

ヒューマノイド・ロ ボット AI とシミュ レーション

ヒューマノイド・ロボティクスおよびAI システム設計の完全ガイド

ヒューマノイド・ロボ ット AI とシミュ レーション

ヒューマノイド・ロボティクスおよび AI システム設計の完全ガイド

First Edition

Gareth Morgan Thomas
Auckland, New Zealand



Published by Burst Books
Auckland, New Zealand

*序文

ヒューマノイドロボティクスの分野は、初期の実験的な機械から、先進的な機構設計、高度な電子工学、人工知能、そして人間とロボットの相互作用を統合した複雑なシステムへと急速に進化してきた。本書『ヒューマノイド・ロボットAIとシミュレーション：ヒューマノイドロボティクスおよびAIシステム設計の完全ガイド』は、学術的厳密さと実践的指針の両立を重視し、ヒューマノイドロボットを理解し、設計し、構築するための包括的な基盤を提供することを目的として執筆された。

本書の動機は、いくつかの変革的技術の融合にある。すなわち、高性能なシミュレーション環境、機械学習およびコンピュータビジョンの進展、そしてリアルタイム制御と意思決定のためのスケーラブルなフレームワークである。これらの技術を確立された工学設計の原理と組み合わせることで、機械的に堅牢であるだけでなく、知的な相互作用と実世界環境における自律的な動作が可能なヒューマノイドロボットを構築することが可能となった。

本書の章構成は、ヒューマノイドロボット開発のライフサイクル全体を反映している。まず、ロボティクスの歴史的および概念的進化を概観し、シミュレーションプラットフォームやソフトウェアエコシステムの統合について述べる。続いて、機械系および電気系のサブシステムを含むヒューマノイドロボットの構造を解説し、ROSやビヘイビアツリーといったソフトウェア基盤へと議論を進める。中盤の章では、モーションプランニング、知覚、強化学習、人間-ロボット相互作用など、実装に重点を置く。後半では、医療、教育、産業分野での応用に加え、倫理、安全性、規制遵守といった重要な観点を取り上げる。最後に、ヒューマノイドロボットを一から構築し、学習させ、実運用に展開するまでを示す完全なケーススタディを提示する。

本書は、ロボティクス分野に入門する学生、AIを物理システムと統合しようとするエンジニア、そして人間-ロボット協調の最前線を探究する研究者など、幅広い読者を対象としている。工学およびプログラミングの基礎的な知識を前提とするが、事例研究、具体例、実践的な洞察を随所に盛り込み、体系的かつ理解しやすい構成としている。

本書の目的は、ヒューマノイドロボットの構築方法を教えることにとどまらず、なぜそれらが作られるのか、そして社会においてどのような役割を果たすのかについて考察を促すことにある。これらの機械が職場、家庭、さらには宇宙探査にまで広がりつつある現在、その能力と影響の双方を理解することは不可欠である。

本書が、次世代のヒューマノイドロボットを形作る人々にとって、技術的な指針であると同時にインスピレーションの源となることを願っている。設計、シミュレーション、人工知能を統合することで、人間の可能性を拡張し、差し迫った課題に取り組み、探究と革新の新たなフロンティアを切り拓くシステムを構想し、構築することができるだろう。

ようこそ『ヒューマノイド・ロボットAIとシミュレーション』へ。

著者について

ガレス・モーガン・トーマスは、複数のSTEM分野にわたる豊富な専門知識を有する技術専門家である。電子工学、ソフトウェア開発、ウェブ開発、プロジェクトマネジメントにおける6つの大学ディプロマに加え、コンピュータネットワーク、CAD、ディーゼル工学、井戸掘削、溶接といった分野の資格を取得し、強固な技術的基盤を築いてきた。

ニュージーランド・オークランドで教育を受けた後、ガレス・モーガン・トーマスはニュージーラ

ンド陸軍に3年間従軍し、その中で規律と問題解決能力を磨いた。長年にわたる技術訓練を経て、現在は科学、技術、工学、数学に対する深い理解を、初学者から上級者までを対象とした専門書シリーズを通じて広く共有することに専念している。

Table of Contents

1	ロボティクスの進化	10
1.1	ロボティクスの歴史	10
1.2	ロボティクスの現代的潮流	14
1.3	ヒューマノイドロボットの台頭	19
1.4	倫理のおよび社会的影響	22
2	NVIDIA ロボティクスエコシステム	26
2.1	Isaac Sim の概要	26
2.2	GROOT を用いたビヘイビアツリーの紹介	29
2.3	ロボティクスにおける Omniverse の役割	31
2.4	NVIDIA SDK の主な特徴	36
3	開発環境の準備	40
3.1	システム要件	41
3.2	Isaac Sim および SDK のインストール	44
3.3	GROOT および Omniverse のセットアップ	46
3.4	最初のステップ：シミュレーション例の実行	52
4	機械的解剖学	55
4.1	ヒューマノイドロボットの骨格構造	56
4.2	アクチュエータとその応用	58
4.3	ヒューマノイドロボットの自由度	63
4.4	ヒューマノイドのバランスに関する課題	65
5	電気部品	68
5.1	センサとその配置	68
5.2	電力システム：バッテリー対配線	71
5.3	マイクロコントローラおよびプロセッサ	75
5.4	通信システム	79
6	機械系と電気系の統合	82
6.1	同期化の課題	82
6.2	効率を考慮した設計	85
6.3	保守に関する考慮事項	88
6.4	一般的な問題のトラブルシューティング	93
7	機械設計のためのツール	98
7.1	CAD ソフトウェアの概要	98
7.2	Omniverse を機械シミュレーションに使用する	102
7.3	ヒューマノイドの実用的設計ヒント	107

7.4	ケーススタディ：ロボットハンドの構築	112
8	材料と製造	115
8.1	ロボティクス用構造材料	116
8.2	3D プリントと CNC 加工	120
8.3	強度と柔軟性のバランス	124
8.4	環境への配慮	127
9	組立とテスト	132
9.1	機械部品の組み立て	132
9.2	フレームワークのストレステスト	137
9.3	機械的故障のトラブルシューティング	143
9.4	実世界における組立ての課題	145
10	電力システム	149
10.1	電源の選択	149
10.2	バッテリー寿命の管理	155
10.3	電気設計における安全性	158
10.4	電力システムの試験とモニタリング	161
11	センサ統合	165
11.1	センサの種類（ビジョン、IMU など）	165
11.2	センサキャリブレーション技術	169
11.3	データ取得と処理	173
11.4	実世界のケーススタディ：障害物検出	178
12	制御システム	183
12.1	モータ制御の基礎	184
12.2	閉ループ対開ループシステム	187
12.3	リアルタイムデータ処理	193
12.4	ヒューマノイドモーション制御における課題	198
13	ロボティクスオペレーティングシステムの導入	203
13.1	ROS および ROS2 の概要	203
13.2	ヒューマノイドロボット向け ROS のセットアップ	207
13.3	基本的な ROS ノードとトピック	211
13.4	ROS ツールによるデバッグ	215
14	ビヘイビアツリーの基礎	218
14.1	ヒューマノイドロボティクスにおける意思決定の基礎	218
14.2	GROOT でのビヘイビアツリーの作成	222
14.3	ツリーのデバッグと最適化	226
14.4	ツリー設計における高度なパターン	230
15	ソフトウェアとハードウェアの統合	234
15.1	ROS を用いたリアルタイム制御	234
15.2	センサおよびアクチュエータとの通信	236
15.3	同期化の課題	240

15.4	ヒューマノイドロボットにおける同期とタイミング	240
15.5	システム統合におけるベストプラクティス	244
16	Isaac Sim の基礎	249
16.1	シミュレーション環境の構築	249
16.2	仮想ツインの構築	253
16.3	物理ベースシミュレーションの機能	260
16.4	シミュレーション問題のデバッグ	262
17	データ収集と学習	267
17.1	合成トレーニングデータの生成	267
17.2	知覚モデルの学習	274
17.3	強化学習の基礎	278
17.4	モデル性能の評価	282
18	実ロボットへの移行	286
18.1	学習済みモデルの転送	286
18.2	シミュレーションと現実のギャップを橋渡しする	290
18.3	実環境でのテスト	297
18.4	継続的なトレーニングサイクル	302
19	コンピュータビジョンの基礎	309
19.1	物体検出と分類	309
19.2	奥行き知覚とステレオビジョン	313
19.3	視覚処理における主要な課題	317
20	ビジョンとロボティクスの統合	321
20.1	ビジョンセンサとハードウェア	321
20.2	ビジョンパイプラインの構築	325
21	Fundamentals of Computer Vision	330
21.1	物体検出と分類	331
21.2	奥行き知覚とステレオビジョン	335
21.3	視覚処理における主要な課題	341
22	ビジョンとロボティクスの統合	344
22.1	ビジョンセンサとハードウェア	345
22.2	ビジョンパイプラインの構築	348
22.3	ケーススタディ：ジェスチャ認識	354
23	高度なビジョン技術	359
23.1	意味的セグメンテーション	359
23.2	ヒューマノイドのためのビジュアル SLAM	363
23.3	事前学習済みモデルの利用	370
23.4	ロボティックビジョンの将来動向	376
24	モーション制御の基礎	381
24.1	運動学と動力学の理解	382
24.2	バランシングアルゴリズム	385

24.3	ケーススタディ：ヒューマノイドの歩行	388
25	高度な運動計画	392
25.1	経路探索アルゴリズム	392
25.2	反応的計画 vs 先行的計画	396
25.3	衝突回避技術	402
26	最適化と課題	406
26.1	動作効率の向上	406
26.2	予期せぬ障害物への対処	409
26.3	複雑なタスクへのモーションプランニングのスケーリング	411
26.4	ヒューマノイド制御におけるよくある落とし穴	419
27	ビヘイビアツリーの拡張	423
27.1	複雑なツリー構造	423
27.2	意思決定へのハイブリッドアプローチ	429
27.3	ケーススタディ：マルチタスクロボット	435
28	協力とチームワーク	440
28.1	複数ロボットの協調	440
28.2	群知能の実装	443
28.3	ケーススタディ：協働ヒューマノイド	452
29	ロボティクスにおける倫理的な行動	456
29.1	意思決定の制約	456
29.2	意図しない挙動の回避	460
29.3	規制および倫理的考慮事項	463
30	HRI の基礎	467
30.1	人間要因の理解	467
30.2	直感的なインタラクションの設計	470
30.3	ケーススタディ：音声コマンドロボット	473
31	高度な相互作用手法	478
31.1	ロボティクスにおける自然言語処理	478
31.2	感情認識	482
31.3	ジェスチャベースの制御	487
32	HRI の応用	489
32.1	医療におけるロボット	489
32.2	教育用ヒューマノイドシステム	493
32.3	障害者支援ロボティクス	497
33	SLAM の基礎	501
33.1	ヒューマノイドの同時位置推定と地図構築を理解する	501
33.2	ヒューマノイドのためのマッピング技術	506
33.3	精度向上のためのセンサ融合	509
33.4	ケーススタディ：屋内ナビゲーション	513
34	経路計画アルゴリズム	518

34.1	グラフベース手法 (A*, ダイクストラ法)	518
34.2	サンプリングベース手法 (RRT, PRM)	522
34.3	パスプランニングへのハイブリッドアプローチ	527
34.4	実世界の例: 障害物回避	533
35	ナビゲーションの課題	537
35.1	動的環境の取り扱い	537
35.2	凹凸のある地形を航行する	541
35.3	センサノイズへの対処	544
35.4	ロバスト性と効率の向上	548
36	強化学習の基礎	551
36.1	重要概念: 状態、行動、報酬	551
36.2	探索と活用のトレードオフ	556
36.3	ロボットのための報酬関数の設計	560
36.4	ケーススタディ: ヒューマノイドのバランス制御	565
37	シミュレーションにおけるロボットの訓練	570
37.1	Isaac Sim を RL 環境に使用する	570
37.2	学習済み挙動のハードウェアへの転送	574
37.3	シミュレーションへの過学習を回避する	578
38	強化学習の応用	584
38.1	ヒューマノイドの歩行と走行	584
38.2	物体操作タスク	588
38.3	強化学習を用いた協調ロボティクス	597
38.4	ロボティクスにおける強化学習の将来動向	600
39	マルチロボットシステムの導入	606
39.1	協働の利点	607
39.2	通信プロトコルの設計	611
39.3	タスク割り当て戦略	614
40	群知能	618
40.1	スワーム・ヒューマノイド・ロボティクスの基礎	618
40.2	分散制御の実装	623
40.3	ケーススタディ: 複数ヒューマノイドによる組立てタスク	628
41	実世界での応用	633
41.1	製造業における協働ロボット	633
41.2	災害対応ロボティクス	637
41.3	協働型ヒューマノイドロボットの未来	640
42	医療におけるロボット	644
42.1	高齢者ケアのための支援ロボット	645
42.2	外科・リハビリテーションにおけるロボット	648
42.3	医療ロボティクスの課題	653
43	教育におけるロボット	658

43.1	ヒューマノイドロボットを用いた STEM 教育	658
43.2	対話的学習体験	661
43.3	ケーススタディ：教室におけるロボット	666
44	産業用ロボット	670
44.1	物流および倉庫におけるヒューマノイド	670
44.2	高度な製造タスク	674
44.3	倫理的および実践的な課題	679
45	機械的トラブルシューティング	682
45.1	一般的な機械故障の特定	683
45.2	保守のベストプラクティス	686
45.3	ケーススタディ：アクチュエータ故障の修理	690
46	ソフトウェアデバッグ	693
46.1	ROS および GROOT を用いたデバッグ	693
46.2	ロボティクスにおける一般的なソフトウェアの落とし穴	697
46.3	性能分析のためのツール	702
47	統合のデバッグ	706
47.1	通信エラーの診断	706
47.2	同期問題の取り扱い	709
47.3	実環境でのテスト	713
48	ロボティクスにおける AI の進歩	717
48.1	制御のための新興 AI モデル	717
48.2	生成 AI とロボティクスの統合	723
48.3	ケーススタディ：予知保全	728
49	宇宙と探査におけるロボティクス	733
49.1	宇宙ミッションにおけるヒューマノイド	733
49.2	ロボティック探査の課題	736
49.3	宇宙ヒューマノイドの将来の可能性	740
50	新たな応用	744
50.1	エンターテインメントおよびゲーミングにおけるロボット	744
50.2	農業における自律型ロボット	750
50.3	ヒューマノイド設計の境界を押し広める	754
51	ゼロから始める	758
51.1	要求と目標の定義	758
51.2	設計および製造プロセスの計画	761
51.3	コストとタイムラインの管理	767
52	段階的な実装	770
52.1	機械構造の設計	770
52.2	制御ソフトウェアのプログラミング	773
52.3	シミュレーションでのロボット訓練	778
52.4	ロボットのテストと展開	785

53	学んだ教訓と今後の方向性	790
53.1	開発中に直面した課題	790
53.2	スケーラビリティの最適化	793
53.3	ヒューマノイドロボティクスの次のステップ	799
	ロボティクスと NVIDIA ツール入門	

1 ロボティクスの進化

1.1 ロボティクスの歴史

本歴史的概観は、今日のヒューマノイドプラットフォームを生み出した工学上の選択をたどる。以下の叙述は、設計要件、制御理論、センシングの進歩を、現代の開発ワークフローに関連する具体的なマイルストーンに結びつける。

歴史を理解することは工学上の目的を果たす：複雑さ、パワー、頑健さの間で繰り返されるトレードオフを明らかにする。初期のオートマタは電氣的駆動なしに運動学とタイミングを探索した。最初の量産アームに始まる産業用マニピュレータは、ペイロード、再現性、単純な軌道制御へ優先順位を移した。ヒューマノイド研究は、二足歩行、全身制御、人間のような操縦を統合設計問題に融合させた。

ヒューマノイド設計を支える重要な技術的事実は以下のように出現した。

- 1950 年代–1960 年代：電気機械的駆動とプログラマブル運動制御が成熟した。産業用ロボットは再現可能なアーム軌道と高トルクモータを優先した。
- 1970 年代：Zero Moment Point (ZMP) 概念は二足の動的安定性を定式化し、有限の足底接触面積を持つロボットで予測可能な歩行制御を可能にした。
- 1973 年：WABOT-1 はヒューマノイドタスクのための全身協調を実証し、研究プラットフォームでビジョンと四肢制御を統合した。
- 1980 年代–1990 年代：サーボ電子機器、マイクロコントローラ、コンパクトセンサの進歩により、多自由度 (DoF) を持つプロトタイプが可能になった。
- 2000 年代：ホンダの ASIMO への発展は、知覚、移動、人間との相互作用能力をスケールで展示した。
- 2010 年代–現在：シリーズ弾性アクチュエータ、可変剛性ジョイント、高性能油圧はボストンダイナミクスの Atlas のような動的俊敏性を重視するシステムに登場した。
- 現在：シミュレーション駆動手法と強化学習は、ハードウェア展開前に歩行と操縦ポリシーを加速する。

制御と設計の観点から、ほとんどの現代ヒューマノイドは、剛体ダイナミクス、接触相互作用、アクチュエータ制約を同時に満たす必要がある。コントローラ設計で使用される一般化ダイナミクスは以下のように残る：

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J_f^\top F \quad (1)$$

ここで $q \in \mathbb{R}^n$ は関節座標を表し、 M は慣性行列、 C はコリオリと遠心力項を集め、 g は重力、 τ はアクチュエータトルク、 J_f は接触ヤコビアン、 F は接触力である。式 (1) は、歴史のアクチュエータと

センサ選択を現代の制御手法に写像する際に依然として中心的である。

設計者は歴史的に3つの結合した制約に対応してきた：

1. 機械的複雑さは状態次元 n を増やし、計算とセンシング要求を高める。
2. アクチュエータ技術はトルク密度と帯域幅を制限し、可能な歩容とペイロードに影響する。
3. センサ品質と配置は接触状態と外部攪乱の可観測性を決定する。

これらの制約は異なる工学領域を生み出した：重いアクチュエータと保守的コントローラを持つ遅く頑健なヒューマノイドと、高度なフィードバックを持つ高帯域幅アクチュエータを用いた俊敏プラットフォーム。

歴史的教訓を実行可能にするために、エンジニアはプラットフォーム特性をカタログ化し、計画と推定ツールチェーンのために状態と制御ベクトルサイズを計算できる。以下の Python リストは簡潔なデータセットと状態ベクトルサイズの単純な計算を示す。この形式は、シミュレータ、ミドルウェア、計算ハードウェアを選択する際の要件キャプチャをサポートする。

コードサンプル 1 歴史的ヒューマノイドプラットフォームのコンパクトデータセット

```
#!/usr/bin/env python3
"""
状態空間サイズ計算を含む
プロダクション対応ロボットカタログ
ROS_2ノードとしても動作可能。
"""

from __future__ import annotations

import argparse
import logging
import sys
from dataclasses import dataclass
from typing import List

# ROS 2 (rclpy) 対応
try:
    import rclpy
    from rclpy.node import Node
    ROS2_AVAILABLE = True
except ModuleNotFoundError:
    ROS2_AVAILABLE = False

# ログ設定
logging.basicConfig(
```

```

        level=logging.INFO,
        format="%(asctime)s_[(levelname)s]_[(name)s:_(message)s",
        datefmt="%Y-%m-%d_%H:%M:%S",
    )
    logger = logging.getLogger("robot_catalog")

```

```

@dataclass(frozen=True, slots=True)
class Robot:
    """ロボット仕様を不変オブジェクトとして保持"""
    name: str
    year: int
    dof: int
    actuation: str

    def state_size(self) -> int:
        """状態ベクトル長_=_関節角_+_関節速度"""
        return 2 * self.dof

```

```

class RobotCatalog:
    """ロボットデータベース"""

    def __init__(self, robots: List[Robot]) -> None:
        self._robots = robots

    @classmethod
    def default_catalog(cls) -> "RobotCatalog":
        return cls(
            [
                Robot("WABOT-1", 1973, 17, "electric"),
                Robot("Unimate", 1961, 6, "electric"),
                Robot("Honda_ASIMO", 2000, 34, "electric"),
                Robot("Boston_Atlas", 2013, 28, "hydraulic"),
            ]
        )

    def summary(self) -> None:
        """標準出力に状態サイズを出力"""
        for r in self._robots:

```

```

        logger.info("%s(%d): state_size=%d", r.name, r.year, r.state_size())

# ROS 2ノード化
if ROS2_AVAILABLE:

    class RobotCatalogNode(Node):
        def __init__(self) -> None:
            super().__init__("robot_catalog")
            self.catalog = RobotCatalog.default_catalog()
            self.timer = self.create_timer(1.0, self.publish_summary)

        def publish_summary(self) -> None:
            self.catalog.summary()

    def main(argv: List[str] | None = None) -> None:
        parser = argparse.ArgumentParser(description="Robot_catalog_state_size_calculator")
        parser.add_argument(
            "--ros2", action="store_true", help="ROS2ノードとして実行"
        )
        args = parser.parse_args(argv)

        if args.ros2 and ROS2_AVAILABLE:
            rclpy.init()
            node = RobotCatalogNode()
            try:
                rclpy.spin(node)
            except KeyboardInterrupt:
                pass
            finally:
                node.destroy_node()
                rclpy.shutdown()
        else:
            RobotCatalog.default_catalog().summary()

if __name__ == "__main__":
    main()

```

数十年にわたり繰り返されるいくつかの工学パターンがある：

- ・モジュラリティは統合リスクを軽減する。センシング、駆動、制御を分離することで段階的検証が可能になる。
- ・エネルギー密度は実用的な運用プロファイルを駆動する。バッテリーは歴史的に連続移動時間を制限した。
- ・接触センシングと力制御はゆっくり成熟したが、操縦と地形交渉にとって決定的になった。
- ・シミュレーション忠実度は転送成功を決定する。ハードウェアのみでチューニングされた初期のコントローラは脆くなった。

運用例は歴史を現在のプロジェクトに結びつける：

- ・倉庫ヒューマノイドは、耐久性と安全性を優先する頑健で低帯域幅のコントローラを必要とする。
- ・災害対応ヒューマノイドは、不均一で予測不能な接触のための動的全身制御と力センシングが必要である。
- ・宇宙ヒューマノイドは放射線硬化電子機器と保守的電力予算を要求し、俊敏性より単純さへ設計選択を移動させる。

工学への影響、トレードオフ、運用上のリスク：

- ・トレードオフ：高 DoF は器用さを改善するが、センシングと制御複雑さを乗算し、故障モードを増やす。
- ・リスク：シミュレーションへの過度の依存は、モデル化されない摩擦、コンプライアンス、センサ雑音のために脆い転送を生む可能性がある。
- ・設計への影響：予想される攪乱を処理するのに十分な帯域幅とマージンを持つアクチュエータとセンサを選択する。
- ・運用上の制約：認証と安全規格はしばしば予測可能で遅い動作を好み、アクチュエータとコントローラ選択に影響を与える。

1.2 ロボティクスの現代的潮流

先ほど概観した歴史的マイルストーンを踏まえ、現代は単一タスクの自動機械から、知覚・計画・全身制御を統合した適応可能で学習機能を持つヒューマノイドへと焦点が移っている。以下の潮流は、ヒューマノイドロボットの設計・シミュレーション・展開において遭遇する具体的なエンジニアリング課題と新興技術を結びつける。

現代的潮流の概要とエンジニアリングの推進要因

- ・知覚駆動型自律性：深層学習の進歩により、ヒューマノイドは物体検出、人間の意図推定、複雑なシーンのセグメンテーションを可能にする。実用的インパクト：知覚モデルはバランスと反応動作を支えるため、組み込みハードウェアで低遅延で動作しなければならない。
- ・シミュレーション優先開発：高忠実度シミュレータがリスクと開発時間を削減。エンジニアはシミュレーションを用いて合成データセットを生成し、全身コントローラを検証し、ハードウェアテスト前に接触を伴う動作を調整する。
- ・学習ベース制御：強化学習（RL）と模倣学習が、動的歩行・操縦のためのモデルベース制御を補

完する。RL は解析的コントローラの導出が非現実的な高次元状態空間で方策を最適化する。

- エッジ・ヘテロジニアスコンピューティング：オンボード GPU と専用アクセラレータ（例：NVIDIA Jetson クラスデバイス）により、畳み込みネットワークと方策推論をオンボードで実行可能にし、オフボードサーバーへの低遅延リンクへの依存を削減する。
- 駆動とコンプライアンス：可変剛性アクチュエータと直列弾性設計は、人間との相互作用時の安全性を向上させ、歩行・跳躍のための機械エネルギーの効率的な蓄積・解放を提供する。
- 多モーダルセンシング・センサ融合：IMU、力・トルクセンサ、触覚アレイ、ビジョンの融合により、単一センサ劣化による故障モードを削減する。
- 人間認識型相互作用と安全性：予測意図モデルと形式的セーフティモニタを動作スタックに統合し、有害な接触を回避しユーザ信頼を維持する。
- クラウドロボティクス・フリート学習：ロボットフリート間での経験とモデル更新の共有が改善を加速するが、セキュアで帯域幅を意識した展開メカニズムが必要である。

技術分析：モデルベースと学習ベースアプローチの融合エンジニアはしばしば剛体ダイナミクスと学習済み残差方策を組み合わせ、既存の物理知識を活用しサンプル効率を向上させる。関節座標 q を持つヒューマノイドに対し、剛体ダイナミクスは

$$[H]\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) + J^T(q)F_{\text{ext}}, \quad (2)$$

と書ける。ここで M は質量行列、 C はコリオリ・遠心力項、 g は重力、 J は接触やコピアン、 F_{ext} は外力である。実用的な制御アーキテクチャは (2) からのモデルベースフィードフォワード項と、補正トルク τ_{learn} を供給する学習済み方策 $\pi_{\theta}(s)$ を用いる。合成指令は

$$[H]\tau_{\text{cmd}} = \tau_{\text{model}} + \alpha \tau_{\text{learn}}, \quad (3)$$

となり、 α は安定性と適応性のバランスを調整するように調整される。

実装パターンとツール

- Sim-to-real パイプライン：シミュレーションでドメインランダム化訓練エピソードを生成し、ハードウェアインザループテストセットで検証し、段階的な方策更新を展開する。ドメインランダム化にはセンサ遅延とアクチュエータ雑音モデルを含める必要がある。
- リアルタイムスタック：バランス・トルク制御の高周波インナーループを、低周波の意思決定層から分離する。インナーループは 500–1000 Hz で動作し、知覚・計画は 10–50 Hz で動作する。
- セーフティエンベロープ：トルク・関節限界・重心制約を強制するモニタリングノードを実装する。違反発生時にはミリ秒単位で安全なフォールバック動作を実行する必要がある。

実用的コード例：PD 制御・剛体モデルからの重力補償・学習済み残差トルクを融合する最小 Python 制御ループ。このスニペットはダイナミクスライブラリ・方策推論・リアルタイム関節状態更新へのアクセスを仮定する。

コードサンプル 2 ハイブリッド PD + 重力 + 学習済み残差制御ループ（概念的）

```
import numpy as np
import rclpy
from rclpy.node import Node
```

```

from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
from std_msgs.msg import Float64MultiArray
import torch
from typing import Optional, Tuple

class ResidualPDController(Node):
    def __init__(self) -> None:
        super().__init__('residual_pd_controller')

        # QoS設定（リアルタイム制御向け）
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        # 関節状態購読
        self.joint_sub = self.create_subscription(
            JointState, '/joint_states', self.joint_callback, qos
        )

        # トルク指令出版
        self.torque_pub = self.create_publisher(
            Float64MultiArray, '/joint_torque_command', qos
        )

        # パラメータ宣言
        self.declare_parameter('kp', 200.0)
        self.declare_parameter('kd', 10.0)
        self.declare_parameter('alpha', 0.6)
        self.declare_parameter('n_joints', 7)

        # 内部状態
        self.q: Optional[np.ndarray] = None
        self.qdot: Optional[np.ndarray] = None
        self.qd: Optional[np.ndarray] = None

        # 学習済みポリシー読み込み（TorchScript）
        policy_path = self.declare_parameter('policy_path', '').value

```

```

self.policy = torch.jit.load(policy_path) if policy_path else None

# タイマー (1 kHz)
self.timer = self.create_timer(0.001, self.control_loop)

def joint_callback(self, msg: JointState) -> None:
    # 関節角度・速度をNumPy配列に変換
    self.q = np.array(msg.position, dtype=np.float64)
    self.qdot = np.array(msg.velocity, dtype=np.float64)

def get_desired_state(self) -> Tuple[np.ndarray, np.ndarray]:
    # 目標関節角度 (外部トピックまたは内部生成)
    if self.qd is None:
        n = self.get_parameter('n_joints').value
        self.qd = np.zeros(n, dtype=np.float64)
    return self.qd, np.zeros_like(self.qd)

def control_loop(self) -> None:
    if self.q is None or self.qdot is None:
        return

    # 動力学計算 (C++バックエンド)
    g = self.compute_gravity(self.q)
    M = self.compute_mass_matrix(self.q)

    # PDフィードバック
    qd, qd_dot = self.get_desired_state()
    kp = self.get_parameter('kp').value
    kd = self.get_parameter('kd').value
    tau_pd = kp * (qd - self.q) + kd * (qd_dot - self.qdot)

    # 学習残差推定
    alpha = self.get_parameter('alpha').value
    tau_learn = self.infer_residual(self.q, self.qdot) if self.policy else 0.0

    # 最終トルク指令
    tau_cmd = tau_pd + g + alpha * tau_learn

    # 指令出版
    msg = Float64MultiArray()

```

```

msg.data = tau_cmd.tolist()
self.torque_pub.publish(msg)

def compute_gravity(self, q: np.ndarray) -> np.ndarray:
    # 実機ではC++ライブラリ呼び出しに置換
    return np.zeros_like(q)

def compute_mass_matrix(self, q: np.ndarray) -> np.ndarray:
    # 実機ではC++ライブラリ呼び出しに置換
    n = len(q)
    return np.eye(n, dtype=np.float64)

def infer_residual(self, q: np.ndarray, qdot: np.ndarray) -> np.ndarray:
    # センサ観測を含む状態ベクトル構築
    obs = self.get_sensor_obs()
    state = torch.tensor(np.concatenate([q, qdot, obs]), dtype=torch.float32)
    with torch.no_grad():
        return self.policy(state).numpy()

def get_sensor_obs(self) -> np.ndarray:
    # 追加センサ（力覚・視覚等）読み取り
    return np.array([], dtype=np.float64)

def main(args=None):
    rclpy.init(args=args)
    node = ResidualPDController()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計トレードオフ、指標、評価エンジニアは以下を用いて性能を定量化すべきである：

1. 安定性マージン（例：キャプチャポイント指標）。
2. エネルギー毎メートルとアクチュエーションサイジングのためのピーク電流。

3. 知覚遅延とエンドツーエンド決定遅延により反応時間を制限。
4. 転移ギャップ：シミュレーションとハードウェア間の性能差。

運用上のリスクと緩和戦略

- ・熱・電力制限：高性能コンピュータとアクチュエータは熱負荷を増大させる。デューティサイクリング、熱モニタリング、保守的電流制限で緩和する。
- ・シミュレーションへの過適合：ランダムイズセンサモデル、実世界キャリブレーション試行、段階的オンハードウェアファインチューニングを用いる。
- ・学習中の安全性：探索を有界アクションセットに制限し、教師あり事前学習とセーフティクリティックを用いる。
- ・通信障害：ネットワークサービス利用不可時に低電力・静的姿勢へ優雅に遷移する動作を設計する。

エンジニアリングへの影響現代のヒューマノイド設計は、敏捷性とエネルギー・熱予算の間でトレードオフを伴う。オンボード GPU 推論を統合すると自律性は向上するが、熱設計要件が増大する。物理ベース制御と学習済み残差を組み合わせることでサンプル複雑度を削減し安定性保証を維持するが、コントローラ間の堅牢なモニタリングと有界仲裁が必要である。堅牢な sim-to-real プラクティスとセーフティモニタは、人間中心環境への運用展開に不可欠である。

1.3 ヒューマノイドロボットの台頭

これまでの歴史的マイルストーンとセンサ主導の動向を踏まえ、本小節ではなぜヒューマノイドプラットフォームが研究・産業の両面で重要性を増しているのかを考察する。これまでの動向を実用的なエンジニアリング要因、制御理論、シミュレーション主導の開発ワークフローに結びつける。

問題定義：ヒューマノイドロボットは人間の環境で頑健に動作し、操縦と移動において人間のアフォーダンスに適合しなければならない。主要な目標は限られた空間への進入、人間用ツールの使用、予測可能な物理的相互作用の提示である。これらの目標を達成するには、力学、センシング、制御、ソフトウェア検証における統合的な問題解決が必要である。

技術分析ではまずコアサブシステムとそれらが課す制御上の課題を分離する。

- ・機械・運動学的複雑性：ヒューマノイドは通常 20–40 の駆動自由度 (DoF) を必要とする。高 DoF は人間のような巧緻性を可能にするが、計画とリアルタイム制御の次元数を増やす。
- ・バランスと全身協調：外部攪乱下での安定性維持は、接触力と重心ダイナミクスをリアルタイムに解決することを要求する。
- ・知覚と意図理解：視覚・触覚知覚は安全な操縦のためのアフォーダンスと接触意図を推定しなければならない。
- ・電力・重量・熱のトレードオフ：アクチュエータとバッテリーが質量・熱バジェットを支配し、耐久性と動的特性に影響する。
- ・ソフトウェアとシミュレーションの忠実度：現実的なデジタルツインはハードウェア試行前のリスクを低減し、物理的に正確なシミュレータが制御イテレーションを加速する。

バランス解析を具体化するため、重心ダイナミクスとゼロモーメントポイント (ZMP) 概念を用い

る。重心質量方程式はロボット重心 (CoM) の線形加速度を外部接触力に関連付ける：

$$[H]m\ddot{r}_G = \sum_i f_i + mg, \quad (4)$$

ここで m は総質量、 \ddot{r}_G は CoM 加速度、 f_i は接触力、 g は重力である。ZMP は接触力・モーメント分布から定義される。制御設計でよく使われる表現は

$$[H]p_{\text{zmp}} = \frac{\sum_i ((r_i \times f_i) + \tau_i)_z}{\sum_i f_{i,z}}, \quad (5)$$

ここで r_i は接触 i の位置、 f_i はその力、 τ_i はそのモーメントである。コントローラは計画運動を p_{zmp} が支持多角形内に留まるように制約し、静的または準静的安定性を保証する。

ヒューマノイドの制御アーキテクチャは、高周波インナーループアクチュエータレギュレータと中周波全身コントローラ、低周波プランナを組み合わせる。典型的なスタック：

1. アクチュエータへのトルクレベルまたは速度レベル PID/インピーダンスコントローラ。
2. モーメントと接触制約を解く全身逆ダイナミクスまたは作業空間コントローラ。
3. 関節またはタスク空間で軌道を生成するモーションプランナ。
4. ビヘイビアツリーまたはステートマシンを用いてタスクをオーケストレートするビヘイビアレイヤ。

シミュレーション主導の実装は開発と安全検証の標準となっている。NVIDIA Isaac Sim は物理的に現実的な接触シミュレーションと知覚のためのドメインランダム化を可能にする。ビヘイビアオーケストレーションは GROOT のようなツリーベースフレームワークで恩恵を受け、モジュラーなタスク記述と安全なフォールバックビヘイビアを促進する。

簡潔な実装スニペットは測定された接触力から平面 ZMP を計算する。このロジックをバランス監視と安全停止トリガのための低遅延モジュールに配置する。

コードサンプル 3 バランス監視のための接触力から平面 ZMP を計算

```
import numpy as np
from typing import Optional, Sequence, Tuple

Array3 = Tuple[float, float, float]

def compute_zmp(
    contact_forces: Sequence[Array3],
    contact_points: Sequence[Array3],
    contact_torques: Sequence[Array3],
    *,
    min_total_force: float = 1e-6,
) -> Optional[Tuple[float, float]]:
    """
```

各接触点の力・位置・トルクから水平面ZMPを計算する。

Parameters

contact_forces: 各接触点の力 (f_x, f_y, f_z) [N]

contact_points: 各接触点の位置 (x, y, z) [m]

contact_torques: 各接触点のローカルトルク (τ_x, τ_y, τ_z) [N·m]

min_total_force: 分母の f_z 合計がこれ以下なら接地なしとみなす

Returns

(px, py): 水平面ZMP位置 [m]. 接地力が小さすぎる場合はNone.

"""

```
forces = np.asarray(contact_forces, dtype=np.float64) # (N,3)
```

```
points = np.asarray(contact_points, dtype=np.float64) # (N,3)
```

```
torques = np.asarray(contact_torques, dtype=np.float64) # (N,3)
```

```
if not (forces.shape == points.shape == torques.shape):
```

```
    raise ValueError("forces/points/torques must have the same shape")
```

```
fz = forces[:, 2]
```

```
denom = fz.sum()
```

```
if abs(denom) < min_total_force:
```

```
    return None # 接地なし
```

```
# 各点でのz軸周りモーメント:  $(r \times f)_z + \tau_z$ 
```

```
mz = np.cross(points, forces)[:, 2] + torques[:, 2]
```

```
# ZMP座標 (水平面)
```

```
px = (points[:, 1] * fz - mz * 0.0).sum() / denom
```

```
py = (-points[:, 0] * fz + mz * 0.0).sum() / denom
```

```
return float(px), float(py)
```

ヒューマノイドプラットフォームを採用する際の実用的エンジニアリング考察：

- ・センサ遅延と帯域幅は全身コントローラのフィードバック帯域幅を直接制約する。
- ・アクチュエータ選択は達成可能なインピーダンス、絶対トルク、熱限界に影響する。
- ・現実的なシミュレーションはハードウェアの剛性・減衰と一致する接触モデルを要求する。
- ・安全要件はコンプライアントビヘイビア、高速転倒検出、エネルギー制限付き駆動を義務付ける。
- ・タスク割当とビヘイビアオーケストレーションは部分的故障下でグレースフルに劣化しなければならない。

設計トレードオフと運用上のリスクは具体的である：

- ・自由度を増やすと操縦は向上するが計算・センシング要件が上昇する。
- ・高トルクアクチュエータは動的タスクを可能にするが重量増とバッテリー寿命減につながる。
- ・積極的なドメインランダムマイゼーションは sim-to-real 転移時間を短縮するが、過度に保守的なポリシーを生む可能性がある。
- ・視覚知覚への過度の依存は遮蔽・照明変化下で脆弱性を導入する。

エンジニアは最大許容転倒確率、アクチュエータの最小デューティサイクル、閉ループ制御の遅延バジェットなど、測定可能な受け入れ基準を優先すべきである。これらのメトリクスがアクチュエータ仕様、センサ選択、シミュレーション忠実度要件を導く。

1.4 倫理のおよび社会的影響

ヒューマノイドの普及と現代の自律性の動向に関する議論を踏まえて、倫理のおよび社会的な問いかけは設計、テスト、展開の決定に反映されなければならない。以下では、これらの問いかけをヒューマノイドシステムに対する工学上の制約、測定可能なリスク、実装可能な緩和策として整理する。

問題の定義と運用上の関連性。ヒューマノイドロボットは人間と共有する空間で動作し、しばしば高い自由度と豊富なセンサスイートを持つ。この組み合わせは、物理的安全性、プライバシーとデータの誤用、社会経済的影響、心理社会的効果という 4 つの運用カテゴリで潜在的な危害を生む。エンジニアはこれらのカテゴリを定量的なリスク指標と制御制約に変換し、知覚、計画、行動モジュールに統合しなければならない。

技術的分析：リスクモデリングと曝露。危害モードの有限集合 i を確率 P_i と重大度 H_i で定義する。モードを横断した期待社会的危害 $E[H]$ は設計・展開時のトレードオフを導く。

$$[H]E[H] = \sum_{i=1}^n P_i H_i \quad (6)$$

ここで、 P_i は自律レベル $a \in [0, 1]$ 、センサ忠実度 s 、対話近接度 p 、運用テンポ r などのシステム変数 x に依存する。エンジニアは P_i をシミュレーションとフィールドデータから学習した関数 $P_i(x)$ としてモデル化できる。導入と曝露は時間とともにスケールし、ロジスティック導入モデルはリスク曝露が非線形に成長することを示す： $A(t) = A_0 / (1 + \exp(-k(t - t_0)))$ 、ここで $A(t)$ は $E[H]$ に乗じて時間経過とともに社会的曝露を生む。

制御理論的な安全性制約。安全性は制御バリア関数 (CBF) で形式化できる。構成 q の連続微分可能な安全関数を $h(q)$ とする。安全集合 $\{q \mid h(q) \geq 0\}$ の前方不変条件は

$$[H]\dot{h}(q, u) + \alpha(h(q)) \geq 0 \quad (7)$$

クラス K 関数 α に対して成り立つ。下位レベルコントローラで CBF を実装することで、高位の自律性があっても物理的安全性を強制する。

実装：測定可能な緩和策と検証。モデルを実行可能な工学項目に変換する：

1. センサデータガバナンス：知覚層でアクセス制御、暗号化ログ、データ最小化を適用する。クラウドサービスに集約センサ出力を共有する際は差分プライバシーを用いる。

2. シミュレーション主導の認証：物理ベースシミュレータ（例：Isaac Sim）を用いて敵対シナリオを実行し、 $P_i(x)$ を推定し、パラメータスイープ下で $E[H]$ を計算する。
3. ランタイムリスク監視：オンラインリスクスコア $R(t)$ を計算し、 $R(t)$ が閾値を超えた際に自律性スロットルを課す。
4. ヒューマンインザループ（HITL）モード：高リスク対話に対して保守的なリモートコントロールフォールバックを維持する。

実用的ランタイム推定器。以下の Python スニペットは、センサ信頼度と人間への近接度を消費し、期待危害を計算し、リスクが閾値を超えた際に自律性を低下させる最小限のリスク推定器を示す。統合時に `get_sensor_confidence` と `get_proximity` を実際の知覚フックに置き換える。

コードサンプル 4 ランタイムリスク推定器と自律性スロットル

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Float32, Bool
from builtin_interfaces.msg import Time
import time
import logging
from typing import Optional

# QoS設定：センサデータ用 BestEffort、ログ用 Reliable
SENSOR_QOS = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=1
)
LOG_QOS = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    history=HistoryPolicy.KEEP_LAST,
    depth=10
)

# パラメータ定義
THRESHOLD = 0.7
AUTONOMY_HIGH = 1.0
AUTONOMY_LOW = 0.2
LOOP_HZ = 50.0 # 50Hz制御ループ
H_PHYSICAL = 10.0 # 重大度重み
```

```

class RiskEstimator(Node):
    def __init__(self) -> None:
        super().__init__('risk_estimator')

        # パラメータ宣言と取得
        self.declare_parameter('threshold', THRESHOLD)
        self.declare_parameter('loop_hz', LOOP_HZ)
        self.threshold = self.get_parameter('threshold').value
        self.timer_period = 1.0 / self.get_parameter('loop_hz').value

        # サブスクライバ：センサ信頼度・近接距離
        self.conf_sub = self.create_subscription(
            Float32, '/perception/confidence', self.conf_callback, SENSOR_QOS)
        self.prox_sub = self.create_subscription(
            Float32, '/perception/proximity', self.prox_callback, SENSOR_QOS)

        # パブリッシャ：自律性レベル・リスク・安全挙動要求
        self.autonomy_pub = self.create_publisher(
            Float32, '/control/autonomy_level', LOG_QOS)
        self.risk_pub = self.create_publisher(
            Float32, '/risk/value', LOG_QOS)
        self.safe_req_pub = self.create_publisher(
            Bool, '/behavior/safe_request', LOG_QOS)

        # タイマー：メインループ
        self.timer = self.create_timer(self.timer_period, self.timer_callback)

        # 内部状態
        self.confidence: float = 1.0
        self.proximity: float = 10.0
        self.autonomy: float = AUTONOMY_HIGH

        # ロガー設定
        self.logger = logging.getLogger('risk_estimator')
        self.logger.setLevel(logging.INFO)

    def conf_callback(self, msg: Float32) -> None:
        self.confidence = max(0.0, min(1.0, msg.data))

```

```

def prox_callback(self, msg: Float32) -> None:
    self.proximity = max(0.01, msg.data) # 0除算防止

def expected_harm(self) -> float:
    # 物理的危害確率を計算
    p_physical = max(0.0, 1.0 - self.confidence) * (1.0 / self.proximity)
    return p_physical * H_PHYSICAL

def timer_callback(self) -> None:
    risk = self.expected_harm()
    if risk > self.threshold:
        self.autonomy = AUTONOMY_LOW
        self.safe_req_pub.publish(Bool(data=True))
    else:
        self.autonomy = AUTONOMY_HIGH
        self.safe_req_pub.publish(Bool(data=False))

    # パブリッシュ
    self.autonomy_pub.publish(Float32(data=self.autonomy))
    self.risk_pub.publish(Float32(data=risk))

    # ログ (throttle 1Hz)
    self.get_logger().info(f'risk={risk:.3f}┐autonomy={self.autonomy}',
                           throttle_duration_sec=1.0)

def main(args=None) -> None:
    rclpy.init(args=args)
    node = RiskEstimator()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

緩和策とガバナンスチェックリスト。エンジニアは以下を実装すべきである：

- 複合現実テストとドメインランダム化シミュレーションで $P_i(x)$ を定量化する。
- 事後インシデント分析のための監査可能なログをシステムが出力するよう計測する。
- CBF 条件 (7) を満たすフォールバック振る舞いを設計する。
- 第三者へのセンサ派生統計公開時に差分プライバシーを適用する。
- 法務・人間要因チームを含む多ステークホルダレビューを展開ポリシーに用いる。

設計上のトレードオフと運用上のリスク。能力と安全性のバランスで実用的なトレードオフが生じる。センサ忠実度 s を上げると知覚誤差は減るがプライバシー曝露が増える。自律性 a を上げるとタスクスループットは向上するが複雑対話での P_i が上昇する。認証とシミュレーション検証は開発コストと時間を増やす。運用上、リスクを緩和するため自律性をダウングレードすると時間臨界タスクでのミッション効果が低下する。

具体的な工学上の影響：

- 倫理的制約を助言的チェックリストではなくハードシステム要求として扱う。
- シミュレーションと被験者実験での広範シナリオテストに予算を割り当てる。
- 自律性を先制的にスロットルできるランタイムリスクモニタを実装する。
- 監査に向けたログとデータガバナンスを設計し、規制審査を想定する。

$E[H]$ を定量化・制御できないと法的責任と公的信頼の喪失が生じる。堅牢な展開には、これらの倫理的制約を制御スタック、検証パイプライン、組織プロセスに統合することが必要である。

2 NVIDIA ロボティクスエコシステム

2.1 Isaac Sim の概要

前述の NVIDIA ツールの概要を踏まえ、Isaac Sim は NVIDIA ロボティクスエコシステム内でヒューマノイドロボットのプロトタイピング、検証、および訓練に使用される主要なシミュレーションエンジンである。以下では、ヒューマノイド開発における Isaac Sim のエンジニアリング上の役割、現実的な挙動を可能にする技術的メカニズム、およびシミュレートされたヒューマノイドに関節レベルのコントローラとセンサを適用するためのコンパクトな実装パターンを説明する。

問題提起。ヒューマノイドロボットには、正確な多体動力学、接触処理、センサ忠実度、スケールなデータ生成が必要である。エンジニアは、以下をサポートする単一プラットフォームを必要とする：バランスと移動のための高忠実度物理、知覚のための現実的なセンサシミュレーション、大規模トレーニングのための GPU 加速実行、ROS/ROS2 および機械学習フレームワークとの相互運用性。

技術分析と主要機能

- 物理とアーティキュレーション：Isaac Sim は Omniverse 上で動作し、アーティキュレーテッドボディと接触のための NVIDIA 物理ライブラリを活用する。剛体動力学、接触解決、摩擦、拘束ソルバーを用いて多関節アーティキュレーションをシミュレートする。アクチュエータレベルの設計では、アーティキュレーション API によって公開される関節トルク、位置、速度状態を扱う。
- シーンフォーマットと拡張性：シーンは USD (Universal Scene Description) ファイルとして作

成・交換される。USD はロボットモデル、環境、センサリグの構成を可能にする。MDL と RTX は視覚ベースの知覚訓練のためのフォトリアリスティックレンダリングを提供する。

- センサと合成データ：Isaac Sim はカメラ、深度、LiDAR、IMU、力・トルクセンサモデルを提供する。これらのセンサは、ハードウェアへの転移学習に適したノイズモデル、キャリブレーションパラメータ、サンプリングレートで構成できる。
- 統合とトレーニング：ネイティブ Python API と Omniverse Kit 拡張機能により、チームは実験をスクリプト化し、PyTorch または TensorFlow と統合し、強化学習（RL）または模倣パイプラインを実行できる。ROS/ROS2 へのブリッジと共有メモリトランスポートにより、学習済みポリシーを実機ヒューマノイドに展開できる。
- スケールとパフォーマンス：GPU 加速と並列ワールドにより、高速なデータ収集とバッチトレーニングが可能になる。決定論的ステップ（設定時）は再現可能な RL 実験を支援する。

関連する動力学と制御の式。アーティキュレーションレベルでは、アクチュエータコマンドは通常、比例微分（PD）則を用いた関節トルク制御のフィードバックから計算される：

$$[H]\tau = K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}), \quad (8)$$

ここで、 τ はトルクベクトル、 q は現在の関節位置、 \dot{q} は関節速度、添字 d は所望の軌道を表す。コンプライアント接触モデルでのシミュレーション安定性のため、陽的時間ステップは支配的な剛性を解決する必要がある。単純な保守的な境界は

$$[H]\Delta t < \sqrt{\frac{m}{k}}, \quad (9)$$

ここで、 m は有効質量、 k は接触またはアクチュエータ剛性である。これは、接触コンプライアンスまたはコントローラ内の仮想インピーダンスを調整する際のタイムステップ選択を指針とする。

実装パターン（実践的）

- USD ヒューマノイドアセットを準備する。CAD または URDF からロボット運動木と慣性パラメータを USD コンポジションにエクスポートする。ファイル名を humanoid.usd とする。
- USD ステージ内でセンサと接地接触材料を、ターゲットハードウェアに合わせて調整した摩擦と反発で構成する。
- Isaac Sim の Python API を使用してステージをロードし、アーティキュレーションハンドルを作成し、(103) を介してトルクを計算し、それらを各物理ステップで適用する制御ループを実行する。
- オプションで ROS ブリッジを介してセンサトピックを公開し、解析のための関節状態をストリーミングする。

Isaac Sim Python バインディングを使用した制御ループの例（説明的、共通 API パターンを使用）。

コードサンプル 5 Isaac Sim: ヒューマノイドをロードし、PD トルクリブを実行し、関節状態を公開

```
import numpy as np
import rclpy
from rclpy.node import Node
```

```

from sensor_msgs.msg import JointState
from omni.isaac.kit import SimulationApp
from omni.isaac.core import World
from omni.isaac.core.articulations import Articulation

# Isaac Sim起動
sim_app = SimulationApp({"headless": False})

# ROS 2初期化
rclpy.init()
ros_node = Node("isaac_pd_controller")
joint_pub = ros_node.create_publisher(JointState, "joint_states", 10)

world = World(stage_units_in_meters=1.0)
world.reset()
world.scene.add_default_ground_plane()

# ロボット読み込み
humanoid = Articulation(prim_path="/World/Humanoid")
world.scene.add(humanoid)
humanoid.initialize()

# PDゲインと目標姿勢
Kp = 150.0
Kd = 3.5
q_des = humanoid.get_joint_positions()
qd_des = np.zeros_like(q_des)

# JointStateメッセージ準備
joint_names = humanoid.dof_names
joint_state = JointState()
joint_state.name = joint_names

# メインループ
for step in range(10000):
    world.step(render=True)
    q = humanoid.get_joint_positions()
    qd = humanoid.get_joint_velocities()

    # PDトルク計算

```

```

tau = Kp * (q_des - q) + Kd * (qd_des - qd)
humanoid.set_joint_efforts(tau)

# ROS 2 配信
joint_state.header.stamp = ros_node.get_clock().now().to_msg()
joint_state.position = q.tolist()
joint_state.velocity = qd.tolist()
joint_pub.publish(joint_state)

rclpy.spin_once(ros_node, timeout_sec=0.0)

ros_node.destroy_node()
rclpy.shutdown()
sim_app.close()

```

エンジニアリングへの影響、トレードオフ、運用上のリスク

- 忠実度 vs 速度：接触剛性またはセンサ忠実度を上げると現実性が向上するが、より小さな Δt とより多くの GPU/CPU リソースが必要になる。下流の使用例（制御チューニング vs データセット生成）に基づいてトレードオフを選択する。
- リアリティギャップ：フォトリアリズムと正確なノイズモデルはギャップを縮小するが、モデル化されていないハードウェア非線形性を排除することはできない。キャリブレーションとドメインランダムマイゼーション実験を計画する。
- 決定論と再現性：決定論的ステップはデバッグと RL 再現性を簡素化する。しかし、決定論的物理を有効にすると並列化または GPU パフォーマンスが低下する可能性がある。
- 安全性と検証：常に最初に保守的で低ゲインのレジームでシミュレーション内でコントローラを検証する。トルクおよび関節速度に対する安全しきい値を持つ段階的テストを通じてハードウェアに転送する。

実践的エンジニアは、Isaac Sim をヒューマノイドアルゴリズムの迅速な反復のための統一環境として扱いながら、ハードウェア転送に影響を与えるシミュレーション仮定を定量化すべきである。

2.2 GROOT を用いたビヘイビアツリーの紹介

Isaac Sim の概要を踏まえ、GROOT は決定論理をランタイム動作に直接マッピングするビジュアル・ノードベースエディタを提供する。本小節では、GROOT 由来のビヘイビアツリーを人間型ロボット制御に活用する方法を説明し、計算コストを示し、Isaac Sim または ROS2 ベースのコントローラへ統合可能なコンパクトな実装例を提供する。

問題設定と工学的的重要性

- 人間型ロボットは、姿勢・歩行・操縦をリアルタイム制約下で調整するため、構造化された反応的な意思決定を必要とする。
- ビヘイビアツリー (BT) は動作ロジックをシーケンシングから分離し、予測可能な成功／失敗セ

マンティクスを提供する。このモジュール性は Isaac Sim でのテストと後の実機展開を簡素化する。

- 工学的目的は、レイテンシと信頼性目標を満たし、安全審査で監査可能な BT を設計することである。

技術分析：BT 要素と実行コスト

- 人間型に用いるコア BT ノードタイプ：
 1. アクションノード：コントローラを呼び出すかアクチュエータコマンドを送信する。
 2. コンディションノード：センサ由来の述語（バランス、接触、視覚）をチェックする。
 3. コンポジットノード：Sequence と Fallback（セレクト）でフローを制御する。
 4. デコレータノード：ガード、タイムアウト、リトライを追加する。
- 共有メモリ（ブラックボード）は robot_pose、zero_moment_point、計画された足踏みリストなどの状態を保持する。ツリーティックを並行スレッドで実行する場合は明示的ロックを用いる。
- ティックレートと計算予算を計画する必要がある。ツリーが周波数 f Hz でティックし、平均してティックごとに N ノードを訪問し、各ノードが CPU 時間 \bar{c} 秒を消費する場合、およその CPU 利用率は：

$$[H]U = \frac{f N \bar{c}}{C_{\text{CPU}}} \quad (10)$$

ここで C_{CPU} は CPU 容量を秒毎秒で表したもの（例：1 コアあたり 1.0）。コントローラジッタやセンサ処理に余裕を残すため、コアあたり $U < 0.6$ を維持する。

人間型動作のための設計パターン

- 反応的バランスフォールバック：シーケンス [CheckBalance, MaintainBalanceAction]；バランスチェック失敗時に RecoveryStep アクションでフォールバック。
- 階層計画：高レベルプランナーが足踏みリストを生成；BT アクションノードが非同期でプランナーを呼び出しブラックボードフラグを待つ。
- タイムアウトとウォッチドッグ：歩行アクションに明示的タイムアウトを付与し、計画された足踏みが無効になった場合の無限ブロッキングを回避。
- 並行実行：Parallel ノードは控えめに使用；コントローラの独立性を保ち、共有アクチュエータには優先仲裁を適用。

実装：シミュレーションプロトタイピング用の最小 Python BT (*py_trees* 使用)

- 以下のリストは、Isaac Sim または ROS2 環境で動作するコンパクトなプロトタイプ BT を示す。アクションノードはシミュレーショントピックとインタフェースし、/humanoid/actuator_cmds にコマンドをパブリッシュし、/humanoid/imu を読み取る。シミュレータ接続時はパブリッシュ/サブスクライバを Isaac Sim API に置き換える。

```
import py_trees
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy

class IsaacSimBTNode(Node):
    def __init__(self):
        super().__init__('isaac_bt_node')
        qos = QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT, history=HistoryPolicy.KEEP_LAST)
        self.create_publisher(rclpy.qos.QoSProfile, 'humanoid/actuator_cmds', qos)
        self.create_subscription(Imu, 'humanoid/imu', self.imu_callback, qos)

    def imu_callback(self, msg: Imu):
        self.latest_imu = msg

    def publish_actuator(self, cmd: dict):
        traj = JointTrajectory()
        traj.joint_names =
```

```

cmd.get('joints', [])point = JointTrajectoryPoint()point.positions =
cmd.get('positions', [])point.velocities = cmd.get('velocities', [])point.effort =
cmd.get('effort', [])point.time_from_start.sec = 0point.time_from_start.nanosec = int(0.02 *
1e9)traj.points.append(point)self.joint_pub.publish(traj)
class CheckBalance(py_trees.behaviour.Behaviour): def init(self, name="CheckBalance", node: IsaacSimBTNode=None): super().init(name)
class MaintainBalance(py_trees.behaviour.Behaviour): def init(self, name="MaintainBalance", node: IsaacSimBTNode=None): super().init(name)
class RecoveryStep(py_trees.behaviour.Behaviour): def init(self, name="RecoveryStep", node: IsaacSimBTNode=None): super().init(name)
def main(): rclpy.init() node = IsaacSimBTNode() root = py_trees.composites.Selector("Root", memory =
False)balance_seq = py_trees.composites.Sequence("BalanceSeq", memory =
True)balance_seq.add_children([CheckBalance(node = node), MaintainBalance(node =
node)])root.add_children([balance_seq, RecoveryStep(node = node)])tree =
py_trees.trees.BehaviourTree(root)tree.setup_with_descendants()rate = node.create_rate(50)50Hz while rclpy.ok():
tree.tick()rclpy.spin_once(node, timeout_sec = 0.0)rate.sleep()rclpy.shutdown()
if __name__ == '__main__': main()

```

実践的統合ノート

- Isaac Sim の同期シミュレーションクロックを使用してティックを物理ステップに合わせる。BT ティック間隔を物理ステップの整数倍に設定する。
- シミュレーションから実機へ移行する際、安全検証とロールバックのため BT にテレメトリフックを組み込む。可能な限りアクションノードの副作用をべき等に保つ。

検証と安全考慮事項

- 転倒緩和など安全上重要な動作では、ガード条件を形式的にモデルチェックする。
- センサドロップアウトをシミュレートし、明示的な故障モードを追加する。ノイズの多い IMU 条件下でリカバリルーチンの平均故障時間を定量化する。

工学への影響、トレードオフ、リスク

- トレードオフ：
 - ティックレートを上げると応答レイテンシは減少するが、式 (10) に従い CPU 負荷とジッタリスクが増加する。
 - 深いツリーはモジュール性を高めるが、最悪ケースのノード走査とデバッグ複雑性が増す。
- 運用上のリスク：
 - 無制限ブロッキングアクションは高優先度安全チェックを停止させる。常にタイムアウトを用いる。
 - 並行アクションからの共有アクチュエータコマンドは競合する可能性がある；集中仲裁またはロックフリーブラックボードパターンを優先する。
- 本番用の人間型では、GROOT 設計のツリーに形式的な安全モニタと保守的なデフォルトフォールバック動作を組み合わせる。

2.3 ロボティクスにおける Omniverse の役割

これらのポイントは、GROOT による振る舞い・タスク構造化と Isaac Sim で可能な物理豊富なシナリオに直接基づいている。Omniverse は共有シーン、アセット、レンダリングインフラを提供し、これらのツールがリアリスティックなヒューマノイド実験で連携動作できるようにする。

Omniverse はヒューマノイドロボティクス研究の統合基盤として機能する。エンジニアはこれを用いて単一の正規シーン表現を作成し、高忠実度アセットを交換し、一貫したジオメトリ、マテリアル、ライティングでマルチフィジックス実験を実行する。ヒューマノイドロボットにとって運用上の利点は具体的で測定可能である：

- デジタルツイン忠実度：CAD ジオメトリ、センサモデル、制御運動学が同じ USD ベースシーンに存在する。これは検証時のシミュレーションとハードウェア間のジオメトリミスマッチを削減する。
- フォトリアリスティックセンサシミュレーション：RTX パストレーシングレンダリングとデノイジングは、現実的な照明とマテリアル応答の下でビジョンスタックの学習データを生成する。
- マルチフィジックス結合：PhysX、Flex、その他のソルバーが共存し、剛体ダイナミクス、ソフトコンタクト、ヒューマノイドの衣服などのクロス相互作用を表現する。
- コラボレーションとライフサイクル管理：Nucleus サーバーが共有 USD ステージをホストし、設計者、制御エンジニア、データサイエンティストが競合するシーンコピーなしに同時にイテレーションできる。
- スケールとオーケストレーション：Omniverse は分散ワーカーとライブ同期をサポートし、RL やシステム同定でのデータ並列学習とマルチインスタンステストを可能にする。

エンジニアは Omniverse をビジュアライザーと再現可能な実験プラットフォームの両方として扱う必要がある。USD ファイルフォーマットは正規変換、ジョイント階層、アセットプロベナンスを確立する。この正規化は、同じソルバーと設定を使用した場合に実験の決定論的再現を促進する。しかし、現実への忠実度はセンサとアクチュエータモデルによって制限される。シミュレーションベース検証で一般的に使用される簡潔な測定モデルは

$$[H]y = h(x) + n, \quad n \sim \mathcal{N}(0, R), \quad (11)$$

であり、ここで x は真の状態、 $h(\cdot)$ はシミュレートされたセンサ観測、 R は雑音共分散である。Omniverse は $h(\cdot)$ にリアリスティックな雑音モデルを組み込むことをサポートし、推定器とコントローラの頑健性に関する制御実験を可能にする。

sim-to-real ミスマッチを定量化することは設計判断に役立つ。一つの測定可能メトリクスは期待観測ギャップ

$$[H]\Delta = \mathbb{E}_{s \sim \mathcal{S}} \|f_{\text{sim}}(s) - f_{\text{real}}(s)\|_2, \quad (12)$$

であり、ここで f_{sim} と f_{real} はシーン状態 s を観測特徴に写像する。エンジニアは以下を通じて Δ を最小化する：

- マテリアルと照明の較正で現実世界の光度応答を一致させること、
- アクチュエータとセンサの確率性モデル化で遅延と帯域幅制限を反映させること、
- テクスチャ、摩擦、質量分布にドメインランダムマイゼーションを適用すること。

実際には、Omniverse は産業・サービスヒューマノイドプロジェクトで使用するワークフローを可能にする：

1. アセット準備：CAD をインポートし、リトポロジーし、MDL マテリアルを適用して正確な反射率を得る。
2. センサとアクチュエータセットアップ：USD ステージにカメラ、IMU、LIDAR、ジョイントアクチュエータをインスタンス化する。
3. 閉ループ実験：ROS または Python で作成した制御方針を実行し、状態とセンサログを収集して解析する。
4. データ生成：RTX レンダリングとランダムイズシーンを用いて、知覚と模倣学習のためのラベル付きデータセットを生成する。
5. ハードウェアインザループ（HIL）：デジタルツインをミラーして実アクチュエータまたは安全コントローラと対話させる。

以下の最小 Python スニペットは、コア Omniverse-Isaac Sim 統合パターンを示す：ヒューマノイド USD をロードし、深度カメラをアタッチし、シンプルなドメインランダムイゼーションを有効にし、ジョイント状態をパブリッシュする。コメントは意図を示すため簡潔に留める。

コードサンプル 6 Minimal Omniverse Isaac Sim loop for a humanoid with depth camera

```
import random
import numpy as np
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import JointState, Image
from geometry_msgs.msg import TransformStamped
import omni.isaac.core.utils.prims as prim_utils
from omni.isaac.core import World
from omni.isaac.core.utils.nucleus import get_assets_root_path
from omni.isaac.core.utils.stage import add_reference_to_stage
from omni.isaac.sensor import Camera
import omni.isaac.core.utils.semantics as semantics
from pxr import Usd, UsdGeom, UsdShade, Sdf

class HumanoidSim(Node):
    def __init__(self):
        super().__init__('humanoid_sim')
        self.world = World(stage_units_in_meters=1.0)
        self._setup_scene()
        self._setup_ros()
        self.episode = 0
```

```

def _setup_scene(self):
    # 地面追加
    self.world.scene.add_default_ground_plane()
    # NucleusからUSD取得
    assets_root_path = get_assets_root_path()
    if assets_root_path is None:
        raise RuntimeError("Nucleusサーバに接続できません")
    usd_path = assets_root_path + "/Isaac/Robots/Humanoid/humanoid.usd"
    add_reference_to_stage(usd_path, "/World/Humanoid")
    self.humanoid_prim = prim_utils.get_prim_at_path("/World/Humanoid")

    # カメラ生成・胸部に固定
    self.camera = Camera(
        prim_path="/World/CameraDepth",
        resolution=(640, 480),
        translation=(0.1, 0.0, 0.3), # 胸部からの相対位置
        orientation=np.array([1, 0, 0, 0])
    )
    self.camera.initialize()
    camera_xform = UsdGeom.Xform.Define(self.world.stage, "/World/Humanoid/root/sp
    self.camera.attach_to_prim(camera_xform.GetPrim())

def _setup_ros(self):
    self.joint_pub = self.create_publisher(JointState, "/humanoid/joint_states", 1
    self.depth_pub = self.create_publisher(Image, "/humanoid/depth", 10)
    self.timer = self.create_timer(0.02, self.timer_callback) # 50Hz

def randomize_materials(self):
    # マテリアルのラフネスをランダム化
    for prim in self.world.stage.Traverse():
        if prim.IsA(UsdShade.Material):
            roughness_attr = prim.GetAttribute("inputs:roughness")
            if roughness_attr:
                roughness_attr.Set(random.uniform(0.2, 0.8))

def timer_callback(self):
    self.world.step(render=True)
    depth = self.camera.get_current_frame()[ "distance_to_camera" ]
    self.publish_depth(depth)
    self.publish_joints()

```

```

def publish_joints(self):
    js = JointState()
    js.header.stamp = self.get_clock().now().to_msg()
    js.name = []
    js.position = []
    for joint_prim in self.humanoid_prim.GetChildren():
        if joint_prim.IsA(UsdGeom.Joint):
            js.name.append(joint_prim.GetName())
            # 簡易：USD attribute から位置取得
            pos_attr = joint_prim.GetAttribute("physics:localPos0")
            if pos_attr:
                js.position.append(pos_attr.Get())
    self.joint_pub.publish(js)

def publish_depth(self, depth: np.ndarray):
    img = Image()
    img.header.stamp = self.get_clock().now().to_msg()
    img.height, img.width = depth.shape
    img.encoding = "32FC1"
    img.step = img.width * 4
    img.data = depth.astype(np.float32).tobytes()
    self.depth_pub.publish(img)

def run_episodes(self, num_episodes=100, steps_per_ep=1000):
    for _ in range(num_episodes):
        self.randomize_materials()
        for _ in range(steps_per_ep):
            rclpy.spin_once(self, timeout_sec=0.0)

def main():
    rclpy.init()
    sim = HumanoidSim()
    sim.run_episodes()
    rclpy.shutdown()

if __name__ == "__main__":
    main()

```

設計トレードオフと運用上のリスクはこれらの機能から直接に続く：

- 計算コスト対忠実度：パストレーシングセンサと複雑コンタクトモデルを有効にするとレイテンシが増加する。検証目的に応じた忠実度を選択する。
- 合成アーティファクトへの過学習：テクスチャの過度なリアリズムは脆い知覚モデルを生む。フォトリアリスティックデータとランダム化変化を組み合わせる。
- 決定論と再現性：GPU 加速ソルバーは非決定性を導入できる。検証ランでは RNG シードを固定しソルバー設定を文書化する。
- ネットワークとコラボレーション制約：Nucleus ベースワークフローは信頼できるストレージと帯域幅に依存する。分散チームのデータ管理方針を計画する。
- HIL における安全性：デジタルツインのミスマッチは安全でないアクチュエータコマンドを引き起こす。初期ハードウェア移行時は常にハードウェアセーフティと保守的制御エンベロープを適用する。

エンジニアは Omniverse を統合ツールチェイン要素として扱うべきである。ヒューマノイドプログラムにとっての主な価値は、統合オーバーヘッドを削減し、知覚と制御開発を加速し、厳格な配備前評価を可能にすることである。ソルバー忠実度、ネットワークアーキテクチャ、データ生成戦略を選択することが、仮想方針を物理ヒューマノイドに移行する際の運用準備と残留リスクを決定する。

2.4 NVIDIA SDK の主な特徴

Omniverse の高忠実度シミュレーションと GROOT のビヘイビア・ツリー原語を基盤に、NVIDIA SDK は仮想テストと実機展開を橋渡しするランタイムとツールを提供する。SDK はヒューマノイドロボットの運用上の要求、すなわち低遅延知覚、高スループット学習、決定論的制御ループ、シームレスな sim-to-real 転送を標的とする。

問題定義. ヒューマノイドロボットは厳しい遅延予算で知覚・計画・制御を同期させる必要がある。知覚モデルはバランスおよび歩行制御器に情報を提供するのに十分な速さで動作しなければならない。学習パイプラインは組込み演算および電力予算を満たすコンパクトなモデルを生成しなければならない。NVIDIA SDK はこれらの工学ニーズを解決するためのアーキテクチャ要素を提供する。

主な技術特徴とその運用上の関連性:

- ハードウェア加速演算:
 - CUDA (Compute Unified Device Architecture) および Tensor Core は知覚・制御で用いられる線形代数カーネルを加速する。混合精度演算 (FP16/INT8) は較正によりモデル精度を保持しながら遅延および電力を削減する。
 - 関連性: TensorRT による INT8 量子化ビジョンモデルはエッジプラットフォームでフレーム当たり遅延を 3-10× 削減し、動的歩行中に高い制御ループレートを実現する。
- 推論およびモデル最適化:
 - TensorRT はレイヤ融合、カーネル自動チューニング、精度較定を実行する。Triton Inference Server は HTTP/gRPC によるスケーラブルなマルチモデルサービングを提供する。
 - 関連性: Triton によりヒューマノイドは単一の Jetson AGX Xavier で知覚・言語モデルをホストしながら、バランスクリティカルモジュールに低遅延ストリームを優先できる。
- センサおよびミドルウェア統合:
 - Isaac ROS パッケージおよびドライバは GPU 加速センサ前処理、カメラ同期、IMU 融合原語

を提供する。ROS2 統合によりトピックベース配線を制御スタックに簡単に組み込む。

- 関連性: 決定論的時間スタンプおよびハードウェア同期画像ストリームはビジュアル・慣性オドメトリにとってクリティカルな motion-to-photon ジッタを削減する。

- 物理および多関節ダイナミクス:

- PhysX および NVIDIA の多剛体ダイナミクス拡張は Isaac Sim で接触およびソフト拘束をモデル化する。正確な接触モデリングはバランスおよび突き飛ばし回復制御器の学習を支援する。
- 関連性: シムと展開で同一接触モデルを用いることで足-地面相互作用のリアリティギャップを削減する。

- 合成データおよびドメインランダムマイゼーション:

- Isaac Sim は手続きシーン生成およびフォトリアリスティックレンダリングによるデータセット合成をサポートする。ドメインランダムマイゼーション API は照明・テクスチャ・センサノイズを変更し一般化を改善する。
- 関連性: ランダム化された合成データでビジョンモジュールを学習すると、センサ較正ドリフトおよび多様な照明に対する頑健性が向上する。

- リアルタイムおよび決定論的実行:

- CUDAstreams、CUDA Graphs、優先スケジューリングはスケジューリングジッタを削減する。JetPack およびコンテナランタイムは決定論的ブートおよびバージョン管理を実現する。
- 関連性: 制御周期デッドライン T_s を満たすには、全パイプライン遅延が有界かつ再現可能であることを保証する必要がある。

遅延予算. ヒューマノイド制御周期 T_s に対し、遅延の合計は T_s を超えてはならない:

$$[H]L_{\text{total}} = L_{\text{capture}} + L_{\text{pre}} + L_{\text{infer}} + L_{\text{post}} + L_{\text{ctrl}} \leq T_s. \quad (13)$$

エンジニアは通常、変動への余裕を確保するため $L_{\text{total}} \leq 0.5 T_s$ を標的とする。

スループット vs. 安定性. バランス補正に必要な閉ループ帯域幅を f_{band} とする。実用的ガイドラインは

$$[H]T_s \leq \frac{1}{10f_{\text{band}}} \quad (14)$$

であり、知覚遅延が制御器に十分な位相余裕を残すようにする。

実装例. 以下の Python スニペットは、GPU 加速推論のためカメラフレームを通信するコンパクトな Triton クライアントを示す。このパターンは Jetson ベースヒューマノイドに展開可能である。展開に合わせてモデルおよびエンドポイント名を置換する。

コードサンプル 7 Triton client for submitting frames to GPU inference server.

```
import os
import time
import logging
from typing import List, Tuple

import cv2
```

```

import numpy as np
import tritonclient.http as httpclient
from tritonclient.utils import InferenceServerException

# ログ設定：INFO以上をコンソールへ
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s_ _%(levelname)s_ _%(message)s",
    datefmt="%H:%M:%S",
)

# 環境変数からサーバアドレス取得（未設定時はlocalhost）
TRITON_URL = os.getenv("TRITON_HTTP_URL", "localhost:8000")
MODEL_NAME = os.getenv("MODEL_NAME", "humanoid_percept")
INPUT_SHAPE = (224, 224)
INPUT_NAME = "input__0"
OUTPUT_NAME = "output__0"

class TritonInference:
    """Triton_ HTTP_クライアントの薄いラッパー"""

    def __init__(self, url: str, model: str) -> None:
        self.url = url
        self.model = model
        self.client = httpclient.InferenceServerClient(url=url, verbose=False)
        # サーバ・モデルが準備できているか確認
        if not self.client.is_model_ready(model):
            raise RuntimeError(f"モデル_ {model}_ が準備できていません")

    def infer(self, img: np.ndarray) -> np.ndarray:
        """1枚の画像を推論し、結果をnumpy配列で返す"""
        inputs = [httpclient.InferInput(INPUT_NAME, img.shape, "FP32")]
        inputs[0].set_data_from_numpy(img)
        outputs = [httpclient.InferRequestedOutput(OUTPUT_NAME)]

        # 同期推論（低遅延）
        try:
            resp = self.client.infer(
                model_name=self.model, inputs=inputs, outputs=outputs
            )

```

```

    )
except InferenceServerException as e:
    logging.error("推論エラー:%s", e)
    raise
return resp.as_numpy(OUTPUT_NAME)

def preprocess(frame: np.ndarray, size: Tuple[int, int]) -> np.ndarray:
    """HWC_BGR->resize->CHW_float32(0-1)->4次元化"""
    resized = cv2.resize(frame, size)
    rgb = cv2.cvtColor(resized, cv2.COLOR_BGR2RGB)
    chw = np.transpose(rgb, (2, 0, 1)).astype(np.float32) / 255.0
    return np.expand_dims(chw, axis=0) # batch次元追加

def main() -> None:
    # カメラ初期化 (ROS2カメラノード使用時はcv2.VideoCaptureを置き換え)
    cap = cv2.VideoCapture(0, cv2.CAP_V4L2)
    if not cap.isOpened():
        logging.error("カメラが開けません")
        return

    # 解像度固定 (処理高速化・帯域削減)
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
    cap.set(cv2.CAP_PROP_FPS, 30)

    triton = TritonInference(TRITON_URL, MODEL_NAME)

    try:
        while True:
            ret, frame = cap.read()
            if not ret:
                logging.warning("フレーム取得失敗")
                time.sleep(0.01)
                continue

            img = preprocess(frame, INPUT_SHAPE)
            result = triton.infer(img)

```

```

        # ここに後処理（分類・検出）を追加
        logging.debug("推論結果_{}_shape=%s".format(result.shape))

    except KeyboardInterrupt:
        logging.info("停止シグナル受信")
    finally:
        cap.release()
        cv2.destroyAllWindows()

if __name__ == "__main__":
    main()

```

設計パターンおよび工学トレードオフ:

- モデル圧縮 vs. 精度: INT8 量子化は遅延およびエネルギーを削減する。しかし、シークリティカルタスクの検出精度を保持するためには量子化対応学習または較定が必要である。
- エッジ vs クラウド: 重いビジョントaskをクラウドインフラにオフロードするとオンボード演算ニーズを削減できる。ネットワーク接続喪失はバランス制御器に許容できない遅延を引き起こすため、クリティカルループはエッジに留めなければならない。
- 混合 CPU/GPU スケジューリング: 行列重い知覚を GPU にオフロードすると CPU を制御ロジックに解放する。不適切なスケジューリングはデータ転送オーバーヘッドを引き起こし GPU 利得を無効化する。CUDA ピンメモリおよびゼロコピーを可能な限り用いる。

運用上のリスクおよび緩和:

- 非決定論的推論時間は安定性を損なう。負荷下でのベンチマークおよび Triton・CUDA Graphs による QoS 強制で緩和する。
- 長期展開中のモデルドリフトは性能を劣化させる。オンライン較定および定期的なドメインランダム化再学習を用いる。
- エッジモジュールのサーマルスロットリングはスループットを削減する。サーマル管理を設計し、保守的性能標的を検討する。

具体的工学インプリケーション:

- 設計の初期段階で式 (370) に従い遅延予算を確立する。Nsight Systems および実ハードウェアで検証する。
- バランスクリティカルな推論およびセンサ融合を決定論的・低遅延パスに配置することを優先する。
- モデル複雑性と制御周波数の間のトレードオフを受け入れ、帯域幅ニーズに対して式 (392) を用いて定量化する。

3 開発環境の準備

3.1 システム要件

これまでの NVIDIA ロボティクススタックおよびシミュレータ機能の概要を踏まえ、本節ではそれらの機能をヒューマノイドロボットプロジェクトで使用する開発マシンおよびデプロイメントホスト向けの具体的な要件に落とし込む。以下のガイダンスはエンジニアリング課題を整理し、制約を分析し、ハイフィデリティシミュレーション、モデル学習、およびハードウェアインザループワークフローとの互換性を確保するための実用的なチェックを提供する。

開発者は次の 3 クラスの要件を満たす必要がある：シミュレーションおよび学習向けの計算・ストレージ、NVIDIA ツールおよび ROS/ROS2 向けのソフトウェア・ドライバ互換性、ロボットハードウェア統合向けのリアルタイム I/O 機能。問題定義：インタラクティブ物理シミュレーション、GPU アクセラレーション感知・制御、ヒューマノイドハードウェアとの決定的通信をサポートするハードウェアおよびシステム構成を選択すること。

技術分析および推奨仕様：

- GPU (必須): Omniverse および Isaac Sim のレンダリングおよび物理アクセラレーションにはハードウェアレイ 트레이シング対応 (Turing 以降) の NVIDIA RTX クラス GPU が必須。計算能力 ≥ 7.5 を要求。実用的ガイダンス：
 - 最小：12 GB VRAM を搭載する単一 GPU (エントリーシミュレーション、小規模シーン)。
 - 推奨：24 GB VRAM (複雑なマルチエージェントシミュレーション、高解像度センサ)。
 - マルチ GPU：スループット重視のモデル学習および並列インスタンス実行には NVLink または PCIe x16 レーンを使用。
- CPU およびメモリ：
 - 最小 CPU：8 物理コア、3.0 GHz 以上。
 - 推奨 CPU：12–16 物理コア (最新 Ryzen/Intel Xeon) で物理ステップおよび通信における CPU 側ボトルネックを回避。
 - RAM：最小 32 GB；並列シミュレーションインスタンスおよび大規模学習データセット向けに推奨 64–128 GB。
- ストレージおよび I/O：
 - プライマリ：OS、SDK、アクティブデータセット用 NVMe SSD 1 TB 以上。
 - スクラッチ：シミュレーションログおよびリプレイ用の追加 NVMe。30–240 Hz でセンサストリームを記録する際は持続 I/O が重要。
 - ネットワーク：標準用途には 1 Gbps；リモート Omniverse サーバーの使用やワークステーション間での大規模データセット共有時には 10 Gbps を推奨。
- オペレーティングシステムおよびドライバ：
 - Ubuntu LTS (20.04 または 22.04) が ROS2、Isaac Sim ツーリング、NVIDIA SDK サポート向けの推奨ホスト OS。
 - 要求される CUDA ツールキットに一致する NVIDIA ドライバをインストール。Isaac Sim および学習フレームワークの要求に応じて CUDA 11.x 以降を使用。

- GPU 対応 Docker コンテナ向けに NVIDIA Container Toolkit をインストール。
- ソフトウェアスタック：
 - Isaac Sim および一般的 ML ライブラリに合わせた管理済み Conda 環境での Python 3.8–3.10。
 - OS に合わせた ROS2 ディストリビューション：例：Ubuntu 22.04 上の ROS2 Humble。
 - Isaac Sim および Omniverse クライアントには互換バージョンの Omniverse ランチャーおよび認証が必要。
- リアルタイムおよびハードウェアインタフェース：
 - 決定的制御ループには *PREEMPT_RT* パッチ済み Linux カーネルまたは専用リアルタイムコントローラを使用。
 - フィールドバス：ヒューマノイドアクチュエータには EtherCAT または CAN が標準。EtherCAT を使用する場合は NIC がリアルタイムイーサネットをサポートしていることを確認。
 - USB およびシリアルポートは OS レベルでパーミッションを持つ必要あり；安全臨界通信には分離された VLAN を優先。

定量的リソース計画：シミュレーションおよび知覚学習に必要な GPU メモリを見積もる。簡易モデル：

$$[H] \text{VRAM}_{\text{required}} \geq V_{\text{base}} + N_{\text{agents}} V_{\text{agent}} + B V_{\text{batch}}, \quad (15)$$

ここで V_{base} はレンダラおよびシーンベースライン、 N_{agents} はヒューマノイド数、 V_{agent} はエージェントあたりの GPU フットプリント (メッシュ、テクスチャ、センサ)、 B は学習バッチサイズ、 V_{batch} はサンプルあたりの GPU メモリである。各項をプロファイリング実行から算出し、十分なヘッドルームを持つ GPU を選択する。

実装：クイックシステムチェックスクリプト

コードサンプル 8 NVIDIA + ROS2 環境向けシステム互換性チェック

```
bash
#!/usr/bin/env bash
set -euo pipefail

# 必要なコマンドの存在チェック
command -v nvidia-smi >/dev/null || { echo "nvidia-smi not found: NVIDIA driver をインストールしてください"; exit 1; }

# GPU 情報表示
echo "=== GPU 情報 ==="
nvidia-smi --query-gpu=name,memory.total,driver_version --format=csv,noheader,nounits

# CUDA Toolkit チェック
if command -v nvcc >/dev/null; then
  echo "=== CUDA Toolkit 情報 ==="
  nvcc --version
else
  echo "CUDA Toolkit is not installed. Please install it to use ROS2 with NVIDIA GPU."
fi
```

```

    echo "nvcc␣not␣found:␣CUDA␣Toolkit␣を確認してください" >&2
fi

# Docker & nvidia-container-toolkit チェック
if command -v docker >/dev/null; then
    echo "===␣Docker␣情報␣==="
    docker --version
    # GPU サポート確認
    if docker info 2>/dev/null | grep -q "Runtimes.*nvidia"; then
        echo "NVIDIA␣Container␣Runtime:␣OK"
        # 最小 GPU テスト
        docker run --rm --gpus all nvidia/cuda:12.2.0-base-ubuntu22.04 nvidia-smi >/dev/null
        && echo "Docker␣GPU␣テスト:␣成功" \
            || { echo "Docker␣GPU␣テスト:␣失敗" >&2; }
    else
        echo "NVIDIA␣Container␣Runtime:␣未検出" >&2
    fi
else
    echo "Docker:␣未インストール" >&2
fi

# ROS 2 チェック
if command -v ros2 >/dev/null; then
    echo "===␣ROS␣2␣情報␣==="
    ros2 --version
    # ROS_DISTRO 環境変数チェック
    [[ -n "${ROS_DISTRO:-}" ]] && echo "ROS_DISTRO:␣$ROS_DISTRO" || echo "ROS_DISTRO:␣未検出" >&2
else
    echo "ROS␣2:␣未検出" >&2
fi

# Python バージョン表示
echo "===␣Python␣情報␣==="
python3 -c "import␣sys,␣platform;␣print(f'Python␣{sys.version_info.major}.{sys.version_info.minor}')"

```

エンジニアリングへの影響、トレードオフ、および運用上のリスク：

- **トレードオフ**：より多くの VRAM およびコアはコストを上昇させるが、シミュレーションから学習までのウォールタイムを短縮し、マルチエージェント実行時のメモリ不足エラーを回避する。
- **運用上のリスク**：ドライバ/CUDA/Isaac Sim バージョンの不一致は非決定的故障を引き起こす；バージョンを固定し、コンテナ化された再現可能イメージを使用する。

- リアルタイムリスク：ハードリアルタイムループに標準カーネルを使用するとパケットロスおよび制御不安定が生じる可能性がある。制御臨界ループをリアルタイムハードウェアにオフロードする。
- セキュリティ/実務上の注意：Omniverse サーバーを実行したりネットワーク経由でデータセットを共有したりすると機密モデルが露出する；VPN、認証、暗号化チャネルを使用する。

これらの要件およびチェックにより、ヒューマノイドシステムに対して予測可能な開発が可能となり、統合サイクルを削減し、コスト、忠実度、運用上の安全性の間のトレードオフが明確になる。

3.2 Isaac Sim および SDK のインストール

前述のハードウェアおよび OS 制約を踏まえ、インストールを試みる前にドライバおよび CUDA の互換性を確認すること。Isaac Sim、Omniverse、SDK のバージョンを GPU ドライバに合わせることで、ランタイムエラーや破損したシミュレーション状態を回避できる。

問題定義: 物理精度の高いヒューマノイドシミュレーションを実行し、合成センサーデータを生成し、ROS または RL パイプラインと統合するために、Isaac Sim および付随する NVIDIA ロボティクス SDK をインストールする。目標は、開発および CI のための再現可能で保守可能な環境を構築し、GUI およびヘッドレス運用のオプションを持つこと。

主要な手順と技術分析:

1. GPU、ドライバ、基本ツールの確認。

- nvidia-smi で NVIDIA ドライバおよび CUDA の可用性を確認。Isaac Sim は最新のドライバに依存しており、不一致はプラグインロードエラーを引き起こす。
- Python および conda をチェック。依存関係の汚染を避けるため専用の conda 環境を使用する。

2. インストール方法の選択。

- デスクトップ開発向け推奨: Omniverse Launcher。管理された Isaac Sim アプリ、アップデート、プラグイン管理を提供。グラフィカル Launcher を使用して、物理、USD スキーマ、RTX 向けの一致する Kit パッケージおよび拡張機能をインストールする。
- CI またはヘッドレスサーバ向け推奨: Isaac Sim 開発者ポータルで提供されるダウンロード可能なヘッドレスアーカイブまたはパッケージ化されたランタイム。ヘッドレスインストールはスクリプト可能で自動トレーニングパイプラインに統合できる。

3. インストールすべき SDK コンポーネントとその理由:

- Isaac Sim (Omniverse Kit アプリ): シミュレーション、USD シーンマネジメント、ロボットアセット。
- Isaac ROS パッケージまたは ROS2 ブリッジ: ロボットコントローラおよび知覚スタックとのリアルタイム統合。
- オンボード知覚モデルをシミュレーション内で実行する場合の推論加速のための TensorRT、cuDNN、CUDA ツールキット。
- オプション: 再現可能な Isaac ROS 環境のための Docker イメージ。

動作確認と簡易サイジング公式:

- ・インストーラまたはアーカイブのダウンロード時間を帯域幅から見積もる:

$$[H]t = \frac{S}{B}, \quad (16)$$

ここで t は秒単位の時間、 S はバイト単位のサイズ、 B はバイト毎秒の帯域幅。10 GiB アーカイブを 100 Mbps リンクでダウンロードする場合、 $t \approx \frac{10 \cdot 2^{30}}{100 \cdot 10^6} \approx 107$ 秒。複数のヒューマノイドアセットおよびテクスチャを管理する際は帯域幅およびストレージを計画する。

実践的なインストールワークフロー（コマンドおよび考慮事項）:

- ・インストール前チェック: ドライバ、ディスク、ユーザ権限。
- ・デスクトップパス: Omniverse Launcher でインストールし、Isaac Sim を起動してサンプルヒューマノイドシーンを選択。
- ・ヘッドレス/CI パス: 提供されたアーカイブおよびスクリプト可能な起動を使用。

自動セットアップスクリプトの例（Linux、ヘッドレス対応）。バージョンおよびプラットフォームに応じてパスおよびパッケージ名を調整。

GPU 存在確認 `if ! nvidia-smi >/dev/null; then echo "NVIDIA driver not found" >2 exit 1 fi`

環境変数のデフォルト値 `ISAAC_SIM_DIR = "ISAAC_SIM_DIR: -/opt/isaac-sim" ISAAC_VENV = "ISAAC_VENV: -humanoid-sim" PYTHON_VER = "PYTHON_VER: -3.9"`

conda 初期化（必要に応じて）`if ! conda >/dev/null; then shellcheck source=/dev/null source "CONDA_EXE" fi`

仮想環境作成（既存ならスキップ）`if ! conda activate "ISAAC_VENV" && > /dev/null; then conda create -y -n "ISAAC_VENV" python = "PYTHON_VER" conda activate "ISAAC_VENV" fi`

依存更新 `python -m pip install -q -U pip setuptools wheel`

Isaac Sim パス確認 `if [[! -x "ISAAC_SIM_DIR/isaac-sim.sh"]]; then echo "IsaacSim not found at ISAAC_SIM_DIR" > 2 exit 2 fi`

ヘッドレス起動（引数があれば渡す）`exec "ISAAC_SIM_DIR/isaac-sim.sh" --headless --no-window@"`

統合ノート:

- ・ヒューマノイドロボットアセットは高解像度メッシュおよび多数の関節自由度を含むことが多い。VRAM が限られた GPU では、テクスチャサイズを削減し RTX デノイジングを無効にしてメモリ不足エラーを防ぐ。
- ・Omniverse 拡張マネージャを使用して omni.physx、omni.kit.viewport、およびセンサープラグインを有効にする。バージョン不一致は拡張インポートエラーを引き起こす。

バージョン互換性チェックリスト:

- ・OS およびカーネルバージョン（Linux: Ubuntu 20.04 が一般的にサポート; デスクトップ向け Windows 10/11）。
- ・NVIDIA ドライババージョン \geq Isaac Sim リリースノートで指定された最小値。
- ・Python バージョン: リリースが対象とする特定の Python マイナーバージョンを使用。
- ・CUDA / cuDNN / TensorRT: 加速推論またはカスタム CUDA カーネルを実行する場合のみ必要。そ

うでなければシミュレーションは GPU レンダリングのみで実行。

ヒューマノイド開発における実装上のリスクおよびトレードオフ:

- GUI (Omniverse Launcher) 対ヘッドレス: GUI はアセット検査およびデバッグを簡素化するが、ヘッドレスはスケーラブルなトレーニングおよび CI に不可欠。両方のワークフローを選択してイテレーション速度と自動化をバランスさせる。
- 忠実度対パフォーマンス: 物理忠実度を上げる (小さなタイムステップ、高サブステップ、ソフトコンタクト) とコンピュートコストが増加。歩容最適化では、高忠実度エピソードをオフラインで実行し、RL データ収集用に忠実度を下げたポリシーロールアウトを実行する。
- ドライバおよび SDK ドリフト: OS またはドライバをパッチすると拡張が黙って破損する可能性がある。バージョンを要求仕様マニフェストで固定し、再現性のためにハードウェア+ソフトウェアハッシュを記録する。

具体的なエンジニアリングへの影響:

- 常にドライバ互換性をデプロイ前に確認してダウンタイムを削減する。
- チーム一貫性のためにコンテナ化するか conda 環境仕様をエクスポートする。
- 複雑なヒューマノイドシーン、センサーログ、データセット生成のために GPU メモリおよびディスクスペースを予算化する。
- 自動トレーニングおよびハードウェアインザループ実験を可能にするためにテスト済みヘッドレス起動スクリプトを維持する。

3.3 GROOT および Omniverse のセットアップ

ハードウェア要件を確認し、Isaac Sim および SDK のインストールが完了したら、次のステップは人間型自律性をサポートするビヘイビアランタイムと可視化レイヤーの統合です。以下の内容は、動作している Omniverse/Nucleus エンドポイントと、同一マシンまたはネットワーク上でアクセス可能な Isaac Sim インスタンスが存在することを前提としています。

GROOT および Omniverse のセットアップは、3つの具体的な目標を持つエンジニアリングプロセスです: GROOT ビヘイビアツリーランタイムをデプロイし、そのランタイムを人間型モデルが存在する OmniverseUSD ステージに接続し、クローズドループ制御要件を満たすためにビヘイビアティックをスケジュールします。以下のサブセクションでは、問題を述べ、技術分析を提供し、実装スケッチを示し、実用的なトレードオフを指摘します。

問題定義

- 目的: GROOT で作成されたビヘイビアツリーを Isaac Sim 内の人間型ジョイントおよびロコモーションコントローラーに、決定的なタイミングと低遅延の状態フィードバックで駆動させる。
- 制約: バランスに必要な制御周波数を維持し、物理タイムステップを尊重し、リアルタイム制御を破壊する CPU オーバランを回避する。

技術分析

1. アセットおよびランタイムポロジ

- Omniverse Nucleus は USD パージョニングおよびアセット同期を提供する。GROOT は Omniverse Kit 内のアプリとして、または USD ステージと通信するヘッドレス Python ランタイムとして動作する。
- 典型的なトポロジ：ローカル Isaac Sim (Omniverse Kit セッション)、GROOT (組み込みまたはリモート)、およびハードウェアインザループ用のオプションの ROS ブリッジ。

2. タイミングおよび安定性制約

- 人間型バランスコントローラーは、システムの最高ダイナミクス f_{\max} に関連する制御周波数を必要とする。ナイキストにより、サンプリング周波数 f_s は $f_s \geq 2f_{\max}$ を満たす必要がある。
- Isaac Sim 物理はタイムステップ dt_{sim} およびシミュレーション周波数 $f_{\text{sim}} = 1/dt_{\text{sim}}$ を使用する。ビヘイビアティックは通常、エイリアシングを回避するために物理タイムステップの整数倍に整列する：

$$[H]f_{\text{beh}} = \frac{f_{\text{sim}}}{n} \quad \text{with} \quad f_{\text{beh}} \geq 2f_{\max}, \quad (17)$$

ここで、 n はビヘイビアティック間のシミュレーションステップ数である。

- n を選択して、 f_{beh} が制御要件を満たしながら CPU 使用率を許容範囲に保つ。

3. 通信遅延および決定性

- 低遅延が重要な場合は、ローカル Omniverse Kit セッションを使用する。
- 分散セットアップの場合、Nucleus への往復遅延を測定し、ビヘイビアタイムアウトにそれを考慮する。

実装チェックリスト

- GROOT をインストール：
 - GUI ワークフロー推奨：Omniverse Launcher 経由で GROOT アプリとしてインストール。
 - ヘッドレスまたは CI：Omniverse パッケージフィードまたは NVIDIA が提供するピン留めされた pip wheel から GROOT Python パッケージを使用。
- バージョン整合を確保：
 - Isaac Sim、Omniverse Kit、および GROOT のバージョンを一致させる。API の不一致はロード失敗またはノードタイプの欠損として現れる。
- Nucleus およびパーミッションを構成：
 - ローカル kit セッションを正しい Nucleus エンドポイントに向け、共有 USD ステージの書き込み/読み取りアクセスを確保。
- ビヘイビアツリーブラックボードキーを USD プリムおよびコントローラトピックにマッピング：
 - ジョイント、フットコンタクトセンサ、IMU、および高レベルコマンドトピックに一貫した命名を使用。

実用的な統合パターン

1. 人間型を含む USD ステージをロード。
2. ロコモーションプリミティブを記述する GROOT ビヘイビアツリーをインスタンス化またはロード。
3. アクションノードをコールバックとして登録し、ジョイントコマンドを公開または制御モードを

変更。

4. 式 (1) から計算された周波数で物理ループと同期してツリーをティック。

実装スケッチ例 (Python) : Isaac Sim とロックステップでティックする、ツリーをロードし、簡単なアクションを登録する最小限のランタイム。プレースホルダを独自の人間型制御 API およびアセットパスに置き換える。

コードサンプル 9 Isaac Sim と統合された最小 GROOT ランタイム

```
import os
import time
import logging
from pathlib import Path
from typing import Dict, Any

import rclpy
from rclpy.node import Node
from rclpy.executors import SingleThreadedExecutor
from rcl_interfaces.msg import ParameterDescriptor, ParameterType

import groot
import carb
import omni.isaac.core.utils.stage as stage_utils
from omni.isaac.core import World
from omni.isaac.core.robots import Robot
from omni.isaac.core.utils.prims import get_prim_at_path
from pxr import Usd, UsdGeom

# ログ設定
logging.basicConfig(level=logging.INFO, format="%(asctime)s_[(levelname)s]_(name)s: %(message)s")
logger = logging.getLogger("HumanoidBTDriver")

class HumanoidBTDriver(Node):
    """ROS2 ノードとして BT を駆動し、Isaac Sim と同期"""

    def __init__(self) -> None:
        super().__init__("humanoid_bt_driver")

        # パラメータ宣言
        self.declare_parameter(
            "behavior_tree_path",
```

```

        ””,
        ParameterDescriptor(
            description=”GROOT_BT ファイルの絶対パス”,
            type=ParameterType.PARAMETER_STRING,
        ),
    )
    self.declare_parameter(
        ”humanoid_prim_path”,
        ”/World/Humanoid”,
        ParameterDescriptor(
            description=”HumanoidのUSD_Primパス”,
            type=ParameterType.PARAMETER_STRING,
        ),
    )
    self.declare_parameter(
        ”simulation_freq”,
        120.0,
        ParameterDescriptor(
            description=”Isaac_Sim物理ステップ周波数[Hz]”,
            type=ParameterType.PARAMETER_DOUBLE,
        ),
    )
    self.declare_parameter(
        ”bt_tick_freq”,
        30.0,
        ParameterDescriptor(
            description=”BT_tick周波数[Hz]”,
            type=ParameterType.PARAMETER_DOUBLE,
        ),
    )

# パラメータ取得
bt_path = (
    self.get_parameter(”behavior_tree_path”)
    .get_parameter_value()
    .string_value
)
if not bt_path or not Path(bt_path).is_file():
    raise FileNotFoundError(f”BTファイルが見つかりません:_{bt_path}”)

```

```

humanoid_path = (
    self.get_parameter("humanoid_prim_path")
    .get_parameter_value()
    .string_value
)
sim_freq = (
    self.get_parameter("simulation_freq").get_parameter_value().double_value
)
bt_freq = (
    self.get_parameter("bt_tick_freq").get_parameter_value().double_value
)

# World取得
self.world = World.instance()
stage = self.world.stage
self.humanoid_prim = get_prim_at_path(humanoid_path)
if not self.humanoid_prim.IsValid():
    raise RuntimeError(f"Primが見つかりません:_{humanoid_path}")

# BT読み込み
self.bt = groot.load_tree(bt_path)
self.bt.register_action("ApplyJointTargets", self.apply_joint_targets)

# 周期設定
self.dt_sim = 1.0 / sim_freq
self.dt_bt = 1.0 / bt_freq
self.next_bt_time = self.world.current_time + self.dt_bt

# タイマー作成 (BT tick用)
self.create_timer(self.dt_bt, self.timer_callback)

logger.info("HumanoidBTDriver初期化完了")

def apply_joint_targets(self, bb: Dict[str, Any]) -> groot.Status:
    """ブラックボードから関節目標を取得しIsaacへ適用"""
    targets = bb.get("joint_targets")
    if targets is None:
        logger.warning("joint_targetsがブラックボードに存在しません")
        return groot.Status.FAILURE

```

```

# ArticulationKernelへ送信
robot = self.world.scene.get_object(self.humanoid_prim.GetName())
if isinstance(robot, Robot):
    robot.set_joint_positions(targets)
    return groot.Status.SUCCESS
logger.error("Robotオブジェクトが取得できません")
return groot.Status.FAILURE

def timer_callback(self) -> None:
    """BT_tickタイマー"""
    self.bt.tick()

def main(args=None):
    rclpy.init(args=args)
    node = HumanoidBTDriver()
    executor = SingleThreadedExecutor()
    executor.add_node(node)
    try:
        executor.spin()
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == "__main__":
    main()

```

設計上のトレードオフおよび運用上のリスク

- ティック周波数を高くすると応答性が向上するが、CPU 負荷が上昇し物理ステップのスキップリスクが増える。ティックレートを利用可能なコアに対してバランスを取る。
- GROOT を同一 Kit プロセス内で実行すると遅延を最小化できるが、一方のアプリでクラッシュが発生するとスタック全体に影響する。
- Nucleus またはリモート GROOT インスタンスを使用する分散セットアップは遅延を追加し決定性を低下させる。安全上重要なタスクにはタイムアウトおよびフェイルセーフビヘイビアを使用する。
- 常に徐々に重いツリーロジックおよびセンサ負荷でテストを行う。ノード実行時間をプロファイリングする。単一のツリーノードが割り当てられた T_{beh} を超えると、タイミングスリップを引

き起こしバランスコントローラを不安定化させる可能性がある。

具体的なエンジニアリングへの影響

- 動的な人間型タスクでは、低レベルコントローラに 100–500 Hz 帯のビヘイビア周波数を指し、高レベル GROOT プランナーはより低いレート（例：10–50 Hz）を使用する。
- Isaac Sim、Omniverse Kit、および GROOT の厳格なバージョンおよび依存関係マトリクスを維持し、ランタイム非互換を回避する。
- Nucleus 接続の喪失または遅延センサ更新を処理するため、ウォッチドッグおよびグレースフルデグラデーションをツリー内に実装する。

3.4 最初のステップ：シミュレーション例の実行

Isaac Sim と NVIDIA SDK がインストールされ、GROOT/Omniverse が設定されたら、次は簡単なヒューマノイドシミュレーションを実行してパイプライン全体を検証する。以下の例では、アセットのロード、物理挙動、センサ出力、基本的な関節空間コントローラを検証することに焦点を当てる。

問題定義と運用上の関連：高度なコントローラの開発や学習データの収集に移る前に、エンジニアはヒューマノイド USD アセットが正しくロードされ、PhysX ベースの動力学が妥当に振る舞い、センサが一貫したデータを公開し、公称コントローラが関節を不安定化させずに動かせることを確認しなければならない。この検証により、知覚、ビヘイビアツリー、強化学習実験へ移行する際のデバッグ時間が削減される。

技術的分析と必要なチェック：

- アセット整合性：USD に関節、衝突ジオメトリ、妥当な質量・慣性が含まれていることを確認する。単位は一貫して（メートル、キログラム）用いる。
- 物理設定：安定性と計算コストを両立する物理タイムステップ dt を選ぶ。接触の多いヒューマノイドシミュレーションでは通常 $dt = 1/240\text{ s} \sim 1/120\text{ s}$ である。
- 制御ループタイミング：サンプリング周波数 $f_s = 1/dt$ は、離散時間 PD 制御を安定にするための最大許容コントローラゲインを決定する。
- センサパイプライン：カメラの内部・外部パラメータ、IMU アライメント、同期されたタイムスタンプを検証する。
- メッセージング：ROS/ROS2 ブリッジまたは Omniverse の publish/subscribe 機構が期待レートで動作することを確認する。

検証に用いる最小限の制御モデル：

- 関節目標位置を指令するための関節空間比例微分（PD）制御を実装する。連続時間トルク則は

$$[H]\tau(t) = K_p(q_{\text{ref}}(t) - q(t)) + K_d(\dot{q}_{\text{ref}}(t) - \dot{q}(t)), \quad (18)$$

で、 K_p, K_d は対角ゲイン行列である。静止目標姿勢では微分項が残留運動を減衰する。タイムステップ dt の離散時間ループでは、 K_d をスケールして臨界減衰に近似する： $K_d \approx 2\sqrt{K_p M_{\text{eq}}}$ 、ただし M_{eq} は近似関節空間慣性である。初期は控えめなゲインを用いる。

ステップバイステップ実装計画：

1. パスとランタイムフラグを定義する：
 - HUMANOID_USD をヒューマノイドの USD ファイルに設定する。
 - 可視化の必要性に応じてヘッドレスまたは GUI 実行を選ぶ。
2. 物理を設定する：
 - dt (物理タイムステップ) とソルバ反復回数を設定する。
 - 繰り返しシナリオ用にコンタクトキャッシングを有効にする。
3. ステージとロボットをロードする：
 - USD ステージをインスタンス化し、ヒューマノイドアーティキュレーションをインポートし、全関節が存在することを確認する。
4. センサをアタッチする：
 - RGB カメラと IMU センサを追加する。カメラ公開レートを 30 Hz、IMU を 200 Hz に設定する。
5. 制御ループを実行する：
 - 関節位置・速度を読み取る。
 - (1) に従い PD トルクを計算する。
 - トルク (または位置指令) を適用する。
6. 出力を検証する：
 - 関節状態トレースと画像をログする。
 - 重心運動と接触力が妥当かをチェックする。
7. 実行後診断：
 - NaN が出ていないか、関節リミットが守られているか、エネルギー増加を検査して不安定を検出する。

以下にコンパクトな Isaac Sim スタイルの Python 例を示す。USD ヒューマノイドのロード、物理の有効化、センサ追加、基本的 PD 制御ループを実演する。実行前に HUMANOID_USD を調整すること。

コードサンプル 10 最小 Isaac Sim 実行：ヒューマノイドロード、簡易 PD 制御、カメラ+IMU

```
# -*- coding: utf-8 -*-
import os
import logging
from pathlib import Path
from typing import Optional

import numpy as np
import carb
import omni.isaac.core.utils.stage as stage_utils
from omni.isaac.core import World
from omni.isaac.core.articulations import Articulation
from omni.isaac.core.utils.nucleus import get_assets_root_path
from omni.isaac.kit import SimulationApp

# ログレベル設定 (Isaac Sim標準)
```

```

logging.basicConfig(level=logging.INFO, format="%(asctime)s_[(levelname)s]_%(name)s: ",
logger = logging.getLogger("HumanoidPD")

# 実行時設定
HUMANOID_USD: Optional[str] = os.environ.get("HUMANOID_USD_PATH")
# 環境変数優先
if HUMANOID_USD is None:
    # Nucleusにサンプルがあれば利用（存在チェック）
    assets_root = get_assets_root_path()
    if assets_root:
        HUMANOID_USD = str(Path(assets_root) / "Isaac/Robots/Humanoid/humanoid.usd")
    else:
        HUMANOID_USD = "/isaac-sim/assets/humanoid.usd" # フォールバック

SIM_OPTIONS = {"headless": False, "width": 1280, "height": 720}
simulation_app = SimulationApp(SIM_OPTIONS)

# World初期化
world = World(stage_units_in_meters=1.0)
stage_utils.add_reference_to_stage(HUMANOID_USD, "/World/Humanoid")
robot = world.scene.add(
    Articulation(
        prim_path="/World/Humanoid",
        name="humanoid",
        position=np.array([0, 0, 1.0]), # 地面にめり込み防止
    )
)

world.reset()
dt = 1.0 / 120.0
world.set_physics_dt(dt)

# PDゲイン（自動チューニングは省略、保守的に設定）
joint_indices = robot.get_active_joints()
dof = robot.num_dof
Kp = np.full(dof, 50.0)
Kd = np.full(dof, 1.0)

# 中立姿勢（USDのdefault位置を初期目標とする）
q_ref = robot.get_joint_positions()

```

```
# 制御ループ
steps = int(10.0 / dt)
log_interval = int(1.0 / (dt * 10))

for i in range(steps):
    world.step(render=True)
    q = robot.get_joint_positions()
    dq = robot.get_joint_velocities()
    tau = Kp * (q_ref - q) - Kd * dq # 要素積で高速化
    robot.set_joint_efforts(tau)

    if i % log_interval == 0:
        com, _ = robot.get_world_pose()
        logger.info(f"step={i:05d} COM_z={com[2]:.3f}")

simulation_app.close()
```

実用的なチェックと期待される診断：

- エネルギー貫性：受動コントローラでは全運動エネルギーと位置エネルギーの和が発散しないはずである。
- 接触応答：接触力がヒューマノイド重量と材質に対して期待範囲内であることを検証する。
- センサ同期：IMU シグネチャを関節加速度およびカメラモーショントラjectoryと相関させる。

エンジニアリング上のトレードオフと運用リスク：

- 高忠実度（小さい dt 、多くのソルバ反復）は CPU/GPU コスト増と引き換えに現実性を向上させる。タスクに応じて dt を選ぶ。
- 攻撃的な PD ゲインは数値不安定や接触チャターを引き起こす；ログベースのスニープでゲインを調整する。
- シミュレーションのセンサモデルはノイズや遅延を簡略化している。ハードウェアへポリシー移行時にこれを考慮する。
- アセットスケールや質量の不整合に注意；単位エラーは非現実的ダイナミクスがよくある原因である。
- ヘッドレス実行はレンダリングによる GPU オーバーヘッドを削減し大規模バッチ実験を可能にするが、ビジュアルデバッグを犠牲にする。

この最初の実行はパイプライン整合性を検証し、統合問題を早期に露呈させる。正常に検証できれば、ヒューマノイドタスクの知覚、ビヘイビアツリー、強化学習へスケールする際の下流リスクが軽減される。

人型ロボットの解剖学

4 機械的解剖学

4.1 ヒューマノイドロボットの骨格構造

前述の機械的概要に続き、本小節ではロボットの骨格を荷重支持、運動、およびセンサ搭載のフレームワークとして考察する。幾何学的運動学、質量分布、および制御・シミュレーションのための実装を結びつける。

明確な問題設定が分析を導く：期待される荷重を支持し、所望のタスク空間ワークスペースを実現し、安定したバランスを許容し、動的運動のための慣性コストを最小化する骨格トポロジーを設計することである。骨格は配線のための簡単なルーティングとアクチュエータおよびセンサの取付点も提供しなければならない。したがってエンジニアは剛性、モジュラリティ、および選択的コンプライアンスをバランスさせる。

技術分析

- 機能モジュール：

1. 胴体と骨盤：主要な荷重経路、胴体慣性要素を統合し、足に対する質量中心を定位する。
 2. 四肢：アクチュエータ関節を持つ直列チェーン；脚は立脚中に地面との制約付き閉リンク相互作用を含むことが多い。
 3. 首と頭：センシングとビジュアルサーボのための低質量、高精度モジュール。
 4. 手：組み込みセンサを持つ巧みなエンドエフェクタ。
- 運動学的トポロジーとフレーム：骨格設計は剛体リンクと関節の運動学的木として自然に表現される。各四肢は直列チェーンとしてモデル化される。制御とシミュレーションのために、同次変換が順運動学を提供する。リンクごとの Denavit-Hartenberg (DH) 補助変換 A_i はリンク $i-1$ から i へ写像する。簡潔に：

$$A_i(\theta_i, d_i, a_i, \alpha_i) = R_z(\theta_i) T_z(d_i) T_x(a_i) R_x(\alpha_i). \quad (19)$$

エンドエフェクタ変換は順序付き積 $T_{0n} = \prod_{i=1}^n A_i$ である。

- 速度と力の写像：ヤコビアン $J(q)$ は関節速度 \dot{q} をエンドエフェクタ空間速度 v へ写像する：

$$v = J(q) \dot{q}. \quad (20)$$

バランスと全身制御のために、重心運動量行列は関節速度を系運動量に関連付け、運動量ベースコントローラに不可欠である。

- 質量分布と動力学：グローバル質量中心 (COM) はバランスとコントローラゲインに影響する。リンクごとの質量 m_i とリンク COM 位置 r_i を共通フレーム内から用いて COM を計算する：

$$r_{\text{COM}} = \frac{1}{M} \sum_{i=1}^N m_i r_i, \quad M = \sum_{i=1}^N m_i. \quad (21)$$

設計者は安定性のため低い胴体 COM を保ちながら、揺れ慣性を低減するため四肢の遠位質量を最小化することを目指す。

設計と実装上の考察

- 関節の選択と配置：
 - 回転関節は大多数の四肢可動部に対してコンパクトでエネルギー効率が良い。
 - 球関節（3 直交回転で実現）は肩と股関節の可動域を可能にするが、アクチュエーションルーティングを複雑にする。
 - 直動関節は胴体高さ調整または線形伸長アクチュエータを除きヒューマノイドでは稀である。
- コンプライアンス：
 - 直列弾性アクチュエーション（SEA）または並列弾性要素は衝撃荷重を低減し力制御を改善する。
 - コンプライアント要素は必要に応じて制御帯域幅を保持するように配置すべきである。
- 構造トポロジーのトレードオフ：
 1. 高モジュラリティ（交換容易）は関節数と配線複雑性を増加させる。
 2. モノコックまたは統合胴体は組立点数を減らす但し修理を複雑にする。
- センサおよび配線との統合：
 - 重いケーブルを骨盤を通してルーティングし遠位慣性ペナルティを回避する。
 - 胴体 COM 近くに IMU を埋め込み、四肢運動からの信号結合を低減する。

実装：運動学的記述からの COM 計算以下の Python スニペットはループ内シミュレーション設計で使用される実用的ツールである。順運動学から得られるリンクごとの質量と局所 COM 位置からロボット COM を計算する。モデルインザループ試験で質量配置変更を迅速に評価するために使用する。

コードサンプル 11 リンクごとの慣性データからヒューマノイド COM を計算。

```
import numpy as np
from typing import Tuple

def compute_global_com(masses: np.ndarray,
                       com_positions: np.ndarray) -> Tuple[float, np.ndarray]:
    """
    全質量と重心を返す
    """
    if masses.ndim != 1 or com_positions.ndim != 2:
        raise ValueError("massesは1次元、com_positionsは2次元で与える")
    if masses.shape[0] != com_positions.shape[0]:
        raise ValueError("massesとcom_positionsの長さが一致しない")
    if masses.min() <= 0:
        raise ValueError("質量は正の値")

    M = masses.sum()
    global_com = (masses[:, None] * com_positions).sum(axis=0) / M
    return float(M), global_com
```

```

if __name__ == "__main__":
    masses = np.array([12.0, 3.5, 2.0, 2.0, 4.0, 1.2, 1.2])
    com_positions = np.array([
        [0.0, 0.0, 0.45],
        [0.0, 0.12, 0.0],
        [0.0, -0.12, 0.0],
        [0.0, 0.12, -0.4],
        [0.0, -0.12, -0.4],
        [0.2, 0.15, 0.1],
        [0.2, -0.15, 0.1],
    ])

    M, global_com = compute_global_com(masses, com_positions)
    print("Total mass:", M)
    print("Global COM(m):", global_com)

```

運用上の影響, トレードオフ, およびリスク

- トレードオフ:

- リンク剛性を高めると位置精度が向上するが, アクチュエータへの衝撃伝達が増加する.
- 遠位質量を追加すると搭載能力が拡大するが, 動的俊敏性が低下しエネルギー消費が増加する.
- DOF を増やすと操縦性が向上するが, 制御が複雑化し故障モードが増加する.

- エンジニアリングリスク:

- COM 配置ミスは単純なバランスコントローラを無効化し, 転倒リスクを増加させる.
- 配線ルーティング不十分は, 繰返し四肢運動下でピンチポイントを生じ故障を引き起こす.
- 高密度胴体内の熱経路を見落とすと, モータおよび電子機器の過熱を引き起こす.

設計者は物理ベースシミュレーションで骨格パラメータを反復し, COM, 慣性, および関節限界に対する感度解析を実施すべきである. これらの手順は性能トレードオフを定量化し, アクチュエータサイジングに情報を提供し, ハードウェア製作前の運用リスクを低減する.

4.2 アクチュエータとその応用

骨格記述により, 関節軸, リンク長, 荷重経路が定まり, アクチュエータの配置と機械的仕事量が決定される. アクチュエータは電気エネルギーと制御指令を関節トルクと線形力に変換するため, それらの運動学的制約とミッションプロファイルを満たす選定が必要である.

問題定義: 必要なトルク, 速度, 帯域, コンプライアンスを達成しつつ, 質量, 熱リスク, 制御複雑さを最小化するアクチュエータを選定・統合する. ヒューマノイドロボットは相反する要求を課す: 支持のための股関節・膝関節での高トルク, 操作作用の手首・手部での高帯域・低慣性, 安全な人間相互作用のための制御されたコンプライアンス.

技術分析。

・アクチュエータのカテゴリと推奨用途：

1. 高減速ギアボックス付きブラシレス DC モータ（例：ハーモニックドライブ）。コンパクトさと高トルク密度が重要な股・膝関節などに使用。効率は良いが反射慣性を増やし、プリロードしないとバックラッシュが生じる。
2. 準直駆動モータ（小減速比）。低慣性と高バックドライバビリティが必要な足首・肩・マニピュレータに使用。
3. シリーズエラスティックアクチュエータ（SEA）。モータと負荷の間にコンプライアント要素（スプリング）を挿入しトルクを計測・衝撃を吸収。脚の衝撃やコンプライアント操作用。
4. 可変剛性アクチュエータ（VSA）。精度とエネルギー効率の両立が必要なタスクのための関節剛性調整。
5. 油圧アクチュエータ。重量級ヒューマノイドや外部荷重タスク向けの高パワー密度。ポンプ・バルブ・油圧制御が必要でシステム複雑さが増す。
6. 空圧。圧縮性のため精密ヒューマノイドでは稀。ソフトロボットエンドエフェクタに有用。

主要なアクチュエータ選定方程式。

・必要関節トルクは動力学・重力・摩擦・安全余裕をカバーしなければならない。単関節について

$$[H]\tau_{\text{req}} = I_{\text{eq}}\ddot{\theta} + \tau_{\text{grav}}(\theta) + \tau_{\text{fric}} + \tau_{\text{margin}}. \quad (22)$$

ここで I_{eq} は関節周りの等価慣性、 $\ddot{\theta}$ は目標角加速度、 τ_{grav} はリンク質量による重力トルク項。

・減速比 G ・効率 η によるモータ-関節マッピング：

$$[H]\tau_{\text{joint}} = \eta G \tau_{\text{motor}}, \quad \omega_{\text{motor}} = G \omega_{\text{joint}}. \quad (23)$$

モータ慣性の反射は G^2 で増大し、制御帯域・バックドライバビリティに影響。動的性能・安全性が重要な場面では低減速比を用いる。

・連続運転時のパワー関係：

$$[H]P = \tau_{\text{joint}}\omega_{\text{joint}}. \quad (24)$$

モータおよび駆動系要素の連続パワーと熱限界を確保する。

評価すべきアクチュエータ性能指標：

- ・トルク密度 (Nm kg^{-1})。
- ・トルク帯域 (Hz) または制御周波数でのボード線図ゲイン。
- ・バックドライバビリティと反射慣性。
- ・熱時定数と連続トルク定格。
- ・位置・トルク分解能；統合エンコーダ・トルクセンサの有無。
- ・ピーク負荷下での機械的コンプライアンスと破損モード。

実装上の考慮事項。

- ・センシング：トルク計測は直接（ひずみゲージまたはトルクセンサ）または SEA・モータ電流から推定。高精度トルク制御には校正済み電流-トルクマッピングと温度補償が必要。
- ・制御モード：

- 位置制御は関節角度を強制。剛性が許容される軌道追従に使用。
- トルク制御は関節トルクを直接調整。力相互作用・コンプライアント動作に必要。
- インピーダンス制御（力と運動の関係を定義する制御概念）は仮想剛性・ダンピングを調整。トルク制御と運動フィードバックを組み合わせで実装。
- アドミタンス制御はその双対：力を入力として運動を生成。アクチュエータのバックドライバビリティとセンサ精度に基づいて選択。
- 制御周波数：モータ電流制御を kHz レートで、外ループトルク／インピーダンス制御を 200-1000 Hz で維持し、安定した接触動力学を実現。
- 機械統合：リンク剛性・ベアリングを設計し、制御ゲインを制限する共振を回避。重いアクチュエータを胴体近くに配置し遠位慣性を低減。

実践的選定手順。

1. 運動タスクから関節トルク・速度エンベロープを Eq. (22) を用いて導出。
2. 候補減速比について Eq. (23) でモータトルク・速度を算出。
3. パワー推定 Eq. (24) とデューティサイクルで熱限界をチェック。
4. 反射慣性とトルク帯域を評価；マニピュレータは低減速比を優先。
5. センシング要件を評価：接触安全が臨界な場所にトルクセンサを統合または SEA を設計。

実装例スニペット：単関節の重力補償付き PD レギュレータによるトルク指令計算。実際にはモデル関数を自前のロボット動力学に置き換える。

コードサンプル 12 1 関節の重力補償 PD トルク指令

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

```
PD+重力補償+慣性FF単関節トルクコマンド発行ノード
ROS2(rclpy)対応、パラメータサーバ経由でゲイン・目標値を動的変更可能。
"""
```

```
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from std_msgs.msg import Float64
from geometry_msgs.msg import Vector3Stamped # theta, theta_dot, theta_ddot_des 受信
import numpy as np
```

```
class SingleJointController(Node):
    def __init__(self):
        super().__init__('single_joint_controller')

        # ---- パラメータ宣言 ----
```

```

self.declare_parameter('Kp', 80.0)
self.declare_parameter('Kd', 2.0)
self.declare_parameter('inertia', 0.05)
self.declare_parameter('gravity_gain', 9.81 * 0.5)

# ---- 購読 ----
self.create_subscription(Vector3Stamped, 'joint_state', self.cb_state, 10)
self.create_subscription(Vector3Stamped, 'target', self.cb_target, 10)

# ---- 配信 ----
self.pub_torque = self.create_publisher(Float64, 'torque_cmd', 10)

# ---- 内部状態 ----
self.theta = 0.0
self.theta_dot = 0.0
self.theta_des = 0.0
self.theta_dot_des = 0.0
self.theta_ddot_des = 0.0

# ---- 制御周期 ----
self.dt = 0.001 # 1 kHz
self.timer = self.create_timer(self.dt, self.control_cycle)

# ---- パラメータ取得ヘルパ ----
def gain(self, name: str) -> float:
    return self.get_parameter(name).get_parameter_value().double_value

# ---- 重力補償・慣性モデル ----
def gravity_torque(self, theta: float) -> float:
    return self.gain('gravity_gain') * np.cos(theta)

def inertia(self, _theta: float) -> float:
    return self.gain('inertia')

# ---- 状態・目標コールバック ----
def cb_state(self, msg: Vector3Stamped):
    self.theta = msg.vector.x
    self.theta_dot = msg.vector.y

def cb_target(self, msg: Vector3Stamped):

```

```

        self.theta_des = msg.vector.x
        self.theta_dot_des = msg.vector.y
        self.theta_ddot_des = msg.vector.z

    # ---- 制御周期 ----
    def control_cycle(self):
        Kp = self.gain('Kp')
        Kd = self.gain('Kd')

        # PD + FF + 重力補償
        tau_pd = Kp * (self.theta_des - self.theta) + Kd * (self.theta_dot_des - self.theta_dot)
        tau_ff = self.inertia(self.theta) * self.theta_ddot_des
        tau_grav = self.gravity_torque(self.theta)
        tau = tau_pd + tau_ff + tau_grav

        # 飽和 (簡易セーフティ)
        tau = np.clip(tau, -10.0, 10.0)

        # 配信
        msg = Float64()
        msg.data = tau
        self.pub_torque.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = SingleJointController()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

技術的影響、トレードオフ、リスク。

- ・高減速比はトルク密度を得るがバックドライバビリティを低下させ反射慣性を増やし、転倒・衝突リスクを高める。
- ・SEA やトルクセンサは安全性を向上させるが位置精度を低下させ複雑さを増す。
- ・油圧システムは高パワーを得られるが流体管理が必要で漏洩時の故障モードが増える。
- ・熱デレーティングと連続パワー制約は、ピークトルク定格よりも持続動作を制限することが多い。
- ・制御戦略はアクチュエータハードウェアと一致させる：トルク性能を持つアクチュエータはコンプライアント HRI と頑健な接触タスクを可能にし、位置専用アクチュエータは適応性を制限する。

設計者は、トルク密度、帯域、コンプライアンス、熱プロファイルをミッション要求と安全要件に対してバランスさせなければならない。

4.3 ヒューマノイドロボットの自由度

前述のアクチュエータ性能と骨格トポロジーは実現可能な動作を決定するが、独立した動作軸を定量化することは設計・制御上不可欠である。ここでは自由度（DoF）を定式化し、タスク要件やヤコビアンに基づく制御との関係を示し、ヒューマノイド設計における配分の実用的チェック法を示す。

自由度はロボットが能動的に指令可能な配置空間の次元を定義する。直列チェーンでは総作動 DoF は各関節 DoF の和となる：

$$[H]N_{\text{act}} = \sum_{i=1}^n \nu_i, \quad (25)$$

ただし $\nu_i \in \{1, 2, 3\}$ は各関節の回転または並進 DoF である。ヒューマノイドでは N_{act} は通常、頭部 N_h 、胴体 N_t 、両腕 $2N_a$ 、両脚 $2N_\ell$ 、手 N_{hand} に分割される。

運用上の関連：タスクは必要な作業空間次元 m を課す。3D 剛体物体の片腕操作では、完全ポーズ制御のため $m = 6$ である。動的バランスと操作を組み合わせた全身タスクでは制約が複合する。冗長性の必要条件は

$$[H]N_{\text{act}} - N_{\text{constraints}} > m, \quad (26)$$

ここで $N_{\text{constraints}}$ は接触と閉ループ運動学的制約を含む。

運動学的写像と冗長性。 $q \in \mathbb{R}^{N_{\text{act}}}$ を作動関節座標とし、 $x \in \mathbb{R}^m$ を作業空間変数（エンドエフェクタポーズ、重心位置など）とする。瞬時運動学的関係は

$$[H]\dot{x} = J(q)\dot{q}, \quad (27)$$

ここで $J(q) \in \mathbb{R}^{m \times N_{\text{act}}}$ はヤコビアンである。 $N_{\text{act}} > m$ のとき冗長性が生じ、二次目的のための内部動作制御が可能となる。逆運動学ではしばしば緩和最小二乗擬似逆行列が用いられる：

$$[H]\dot{q} = J^\top (JJ^\top + \lambda^2 I)^{-1} \dot{x} + (I - J^\dagger J)z, \quad (28)$$

ここで λ は緩和因子、 z は姿勢またはトルク最小化のためのヌル空間指令である。

典型的な配分と工学根拠：

- ・頭部：視線とステレオセンサアライメントのため $N_h = 2-3$ 。2DoF でパン／チルト視線制御で十分；ロールを追加するとカメラ安定化を支援する。

- 胴体：腰部回転と曲げのため $N_t = 1-3$ 。胴体 DoF は到達可能ワークスペースを変え、操作レバレッジを改善する。
- 腕： $N_a = 7$ が一般的（肩 3、肘 1、手首 3）。7DoF では障害物回避と巧みな操作のための腕の冗長動作が可能。
- 脚： $N_\ell = 6$ が標準（股関節 3、膝 1、足首 2）；一部設計では足首ピッチまたはトー作動のため 7 を用いる。脚 DoF は不安定な地形での安定性とコンプライアンスに影響する。
- 手： N_{hand} は広範囲に及ぶ。単純なグリップは 1-2DoF；人型手は巧みさのため 20+DoF を持つことがある。

設計問題の記述：操作・移動・知覚の同時タスクを満たしながら、質量・コスト・制御複雑さを最小化するよう DoF を配分せよ。以下の工学チェックを用いよ：

1. 十分な作業空間ランクを検証：ワークスペース全体で主要タスクに対し $\text{rank}(J) \geq m$ を確保せよ。
2. 冗長性マージンを評価： $r = N_{\text{act}} - m_{\text{task}}$ を二次目的に利用可能なヌル空間次元として計算せよ。
3. アンダーアクチュエーションを評価：浮遊基底のために仮想基底 $\text{DoF} N_b = 6$ を動的定式化に加えよ。全身動力学では $q_{\text{wb}} = [q_b, q]$ ($q_b \in \mathbb{R}^6$) を用いる。

実用的計算と健全性チェックは設計反復で日常的に行われる。以下のリストは、Python でコンパクトな DoF アロケータとヤコビアン次元チェッカを実装したものである。グループ DoF を集計し、タスク実現可能性をチェックし、予想されるヤコビアン形状を計算する。

コードサンプル 13 DoF 配分とヤコビアン次元チェッカ（プロトタイプ）

```
import numpy as np
from typing import Dict, Tuple

# 関節群ごとのDoF定義（実用的なヒューマノイド例）
DOF: Dict[str, int] = {
    'head': 3,
    'torso': 3,
    'arm': 7,
    'leg': 6,
    'hand': 20
}

# 駆動関節総数
N_ACT: int = (
    DOF['head'] +
    DOF['torso'] +
    2 * DOF['arm'] +
    2 * DOF['leg'] +
    DOF['hand']
```

)

タスク次元定義（右腕姿勢6，重心位置3）

```
TASK_DIMS: Dict[str, int] = {  
    'right_arm_pose': 6,  
    'com_position': 3  
}
```

```
M_TASK: int = sum(TASK_DIMS.values())
```

浮遊基底仮想DoF

```
N_BASE: int = 6
```

ヤコビアン形状

```
J_SHAPE: Tuple[int, int] = (M_TASK, N_ACT)
```

冗長性チェック

```
REDUNDANCY: int = N_ACT - M_TASK
```

```
assert REDUNDANCY >= 0, "タスク次元が駆動DoFを超えている"
```

ランク検証用サンプル

```
J_sample: np.ndarray = np.random.randn(*J_SHAPE)
```

```
rank: int = np.linalg.matrix_rank(J_sample)
```

```
assert rank == M_TASK, "ヤコビアンがフルランクでない"
```

制御・ハードウェアへの影響：

- ・冗長性は衝突回避・トルク最適化・特異点回避を可能にするが、リアルタイム逆運動学の計算負荷を増大させる。
- ・高 DoF は多くのアクチュエータ・配線・質量を意味し、エネルギー効率を下げ熱設計要求を高める。
- ・アンダーアクチュエート要素・コンプライアンス・パッシブ関節は制御権限を低下させ、特に動的移動中の安定化を複雑にする。
- ・センサ配置と慣性分布は DoF 配分と相互作用し、胴体 DoF は胴体を再配置することで腕モータトルク要求を低減できる。

工学上のトレードオフと運用上のリスク：

- ・トレードオフ：操作性（腕 DoF 増）とエネルギー予算（アクチュエータ削減）を選び、タスク要求とバッテリー容量を釣り合わせる。
- ・リスク：主要ポーズでのヤコビアンランク不足は制御不能を招き、転倒またはペイロード落下の可能性がある。
- ・実装コスト：追加 DoF は増加した制御帯域・故障検出ロジック・より複雑なキャリブレーション手順を要求する。

4.4 ヒューマノイドのバランスに関する課題

自由度の増加と多様なアクチュエータ選択により、バランス維持のための制御帯域とセンシング要件が高まる。アクチュエータの配置、コンプライアンス、レイテンシは、実行可能な安定化戦略を直接的に制約する。

ヒューマノイドのバランスは運用上の問題である：ロボットの重心（CoM）と角運動量を維持し、接触部が指令された動作を支え、転倒や滑りを回避する。主要な技術的課題は、ハイブリッドでアンダーアクチュエートされた動特性と不完全なセンシングに起因する。制御設計に有用なコンパクトな動特性記述は

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = S^T\tau + J_c^T(q)f_c, \quad (29)$$

である。ここで、 q は一般化座標ベクトル、 M は慣性行列、 C はコリオリ／遠心力項、 g は重力、 S はアクチュエータ選択行列、 τ はアクチュエータトルク、 J_c は接触ヤコビアン、 f_c は接触レンチである。式（29）は二つの運用上の現実を浮き彫りにする：

- ・接触力 f_c は摩擦円錐、支持多角形、一方向法線力によって制約される決定変数である。
- ・ベース運動が直接アクチュエートされない場合、システムはアンダーアクチュエートとなり、バランスは内部トルクの再配分とステップによって達成されなければならない。

設計と制御に現れる二つの標準的バランス指標は以下の通り：

1. ゼロ・モーメント・ポイント（ZMP）。全接地反力 F_z と基準点周りの合成モーメント成分 M_x, M_y に対して、

$$[H]x_{\text{ZMP}} = -\frac{M_y}{F_z}, \quad y_{\text{ZMP}} = \frac{M_x}{F_z}. \quad (30)$$

ZMP は接触安定性チェックを提供する： $(x_{\text{ZMP}}, y_{\text{ZMP}})$ が支持多角形内にあれば、準静的条件下で静的転倒は回避される。

2. キャプチャポイント（CP）と線形倒立振子モデル（LIPM）。CoM 高さ z_c を一定とすると、LIPM はサジタル動特性を

$$[H]\ddot{x} = \omega_0^2(x - x_{\text{ZMP}}), \quad \omega_0 = \sqrt{\frac{g}{z_c}}. \quad (31)$$

と線形化する。CP は $\xi = x + \dot{x}/\omega_0$ である。CP 位置へステップすることで、LIPM 適合動作に対して漸近的バランス回復が得られる。

これらの指標は不可欠であるが限界がある。ZMP は平面接触を仮定し、CoM 周りの角運動量を見捨てる。CP は小角運動量と一定 CoM 高さを仮定する。実用的なコントローラは、これらのモデルを階層最適化で完全動特性制約と組み合わせる：

- ・上位レベル：重心 MPC または足歩プランナが摩擦円錐と到達可能足底配置の下で CoM 軌道と所望接触力を解く。
- ・中位レベル：全身 QP が重心指令を追従しつつ関節限界を尊重するように関節トルク／加速度を計算する。
- ・下位レベル：トルク／電流コントローラがアクチュエータ動特性を補償し内ループ安定性を供給する。

バランス失敗を引き起こす実装上の考慮事項：

- ・センサ雑音とレイテンシ。IMU バイアス、遅延した力／トルク読み、低レートビジョンが古い状態推定値を生む。相補フィルタ、状態推定器（EKFまたはUKF）、タイムスタンプ付きセンサ融合を用いる。
- ・構造コンプライアンス。フレキシブルリンケージと直列弾性アクチュエータは、高周波フィードバックゲインを劣化させ位相余裕をずらすコンプライアントモードを導入する。
- ・接触不確実性。モデル化されていない地形コンプライアンスまたは小さな障害物が支持多角形と実効接触法線を変える。
- ・アクチュエータ限界。トルクまたは速度の飽和が指令された安定化インパルスを阻止し、代わりの回復動作（例：足首戦略の代わりにステップ）を強制する。

多くのコントローラで用いられるコンパクトで実用的な関数は、足底力-トルクセンサと IMU 導出 CoM 加速度から ZMP を推定する。以下のコードは、ローパスフィルタリングと単純なキャプチャポイント予測を備えた堅牢なオンライン計算を示す。

コードサンプル 14 足底 F/T と IMU から ZMP と予測キャプチャポイントを計算

```
import numpy as np
from typing import Dict, Tuple

# 物理定数
g: float = 9.81
z_com: float = 0.85
omega0: float = np.sqrt(g / z_com)
alpha: float = 0.8

# フィルタ状態
_zmp_filtered: np.ndarray = np.zeros(2)

def compute_zmp(
    ft_left: Dict[str, np.ndarray],
    ft_right: Dict[str, np.ndarray],
    imu_accel: np.ndarray,
    foot_positions: Dict[str, np.ndarray],
) -> Tuple[np.ndarray, np.ndarray]:
    """
    ZMP 推定とキャプチャポイント予測
    """
    # 合力・合力モーメント
    F: np.ndarray = ft_left["force"] + ft_right["force"]
```

```

M: np.ndarray = ft_left["moment"] + ft_right["moment"]
Fz: float = F[2] + 1e-6 # ゼロ除算回避

# ZMP 計算
zmp_meas: np.ndarray = np.array([-M[1] / Fz, M[0] / Fz])

# ローパスフィルタ
global _zmp_filtered
_zmp_filtered = alpha * _zmp_filtered + (1 - alpha) * zmp_meas

# CoM 状態（外部推定器から取得想定）
com_pos: np.ndarray = np.zeros(2)
com_vel: np.ndarray = np.zeros(2)

# キャプチャポイント
cp: np.ndarray = com_pos + com_vel / omega0

return _zmp_filtered, cp

```

設計上のトレードオフと運用上のリスク：

- DoF とコンプライアントアクチュエーションを増やすとエネルギー効率と安全な相互作用が向上するが、推定器と制御の複雑さが上がる。
- 高帯域剛性アクチュエータは攻撃的な足首戦略を可能にするが、衝撃力と構造応力を増大させる。
- 保守的計画（支持多角形内の大きな余裕）は堅牢性を向上させるが、雑然環境での可動性を低下させる。
- 単純化モデル（ZMP または LIPM）のみに依存すると、凹凸、コンプライアント、または高度に動的な相互作用タスクで失敗するリスクがある。

エンジニアは堅牢な状態推定、明示的接触モデリング、摩擦およびアクチュエータ制約を課す階層制御を優先すべきである。失敗モードには、遅延トルクによる転倒、過小評価摩擦による滑り、モデル化されていないコンプライアンスによる振動が含まれる。設計時にセンサ雑音、アクチュエータ遅延、地形コンプライアンスに対する感度解析を定量化し、アクチュエータ選択、センサ配置、制御アーキテクチャを導くことが求められる。

5 電気部品

5.1 センサとその配置

機械的な構造とアクチュエータ配置の制約を踏まえた上で、センサの配置は運動学的可動範囲、構造剛性、電力・データの配線を調整しなければならない。適切な配置は制御性能、状態推定精度、運

用上の頑健性に直接結びつく。

問題定義：二足歩行ロボットに対し、機械的パッケージングおよび電磁的制約を尊重しながら、バランス、歩行、操縦、知覚のための正確で低遅延の計測を提供するようにセンサを配置する。主な目的は：

- 制御ループに関連する量を変換誤差最小で計測すること、
- 遮蔽を最小化し十分な視野（FOV）を確保すること、
- 配線、EMI、熱暴露、保守アクセスを管理すること、
- 安全臨界機能のための冗長性を提供すること。

技術的分析と原理

1. フレーム共配置と変換。すべてのセンサ読み取りは融合前に共通のボディフレームへ変換されなければならない。剛体変換記法を用いる：センサ位置は

$$[H]p_s = R_{sb} p_b + t_{sb}, \quad (32)$$

ここで $R_{sb} \in SO(3)$ はボディからセンサ座標への回転、 t_{sb} は並進である。小さな配置誤差 δt と δR はヤコビアンを介して状態推定に伝播する。共分散変換は

$$[H]\Sigma_s = J(\theta) \Sigma_b J(\theta)^\top, \quad (33)$$

ここで $J(\theta)$ は変換の線形化；これは配置不確実性が計測不確実性をどれだけ増大させるかを定量化する。

2. IMU 配置とレバーアーム効果。加速度計は取り付け点での線加速度を計測する。IMU が回転中心からオフセット r の位置にあるとき、回転運動は追加の見かけ上の加速度を生じる：

$$[H]a_s = a_{\text{CoM}} + \alpha \times r + \omega \times (\omega \times r) + g + \eta, \quad (34)$$

α は角加速度、 ω は角速度、 g は重力、 η はセンサノイズ。主要 IMU をロボットの質量中心付近に配置すると ω 依存項が最小化され胴体状態推定のバイアスが低減される。局所関節状態制御には、肢に搭載した IMU がセグメント運動の直接計測を提供できるが、追加のフレーム変換と同期のコストが伴う。

3. ビジョンセンサ幾何と深度精度。ステレオまたは構造化光深度に対し、深度 z は視差 d と $z = fB/d$ で関連し、ここで f は焦点距離、 B はベースラインである。深度不確実性は概ね

$$[H]\sigma_z \approx \frac{z^2}{fB} \sigma_d. \quad (35)$$

したがってベースライン B を増やすと長距離深度精度が向上するが、頭部質量と遮蔽リスクが増大する。胴体上部にカメラを搭載するとシーンカバレッジが向上するが、近接操縦に対して視差とキャリブレーション感度が高まる。

4. 力・トルクおよび触覚センサ。反力は負荷経路の曖昧さを避けるため相互作用インターフェースにできるだけ近く計測すべきである。六軸力・トルクセンサは手首と足首に配置し、直接接触力制御または歩行支持が必要なタスクに対応する。触覚アレイは手のひらと足底に配置し、ゾーンベース多重化で配線密度を削減する。

5. 知覚配置のトレードオフ。障害物回避とナビゲーションには、頭部または胴体上部に広 FOV カメラとライダーを配置し地平カバレッジを最大化する。操縦には手首にハンドアイカメラを追加し、近接高解像度ビューを提供する。ビジョンセンサ間の相互遮蔽を最小限に抑えること。

実装チェックリストと配線考慮事項

- ・剛体搭載は IMU 精度を向上；グローバル状態推定に用いる主要 IMU にはソフトマウトを避ける。
- ・高帯域センサには差動信号、ツイストペア、シールドを用いて EMI を抑制する。
- ・肩・股関節を通るケーブル経路はストレインリリーフを施し擦耗を回避する。
- ・フィールドキャリブレーションと交換のためセンサアクセスパネルを設計する。
- ・時刻同期：IMU、カメラ、F/T センサ間でサブミリ秒アライメントのためハードウェア PPS または PTP を用いる。

実用的アルゴリズム手順：ボディフレーム内の IMU オフセットに対する加速度計読み取り補正。以下のスニペットはボディ角速度と加速度が与えられたときに補正済み加速度を計算する。

コードサンプル 15 IMU 加速度計のレバーアーム効果補正

```
import numpy as np
from typing import Tuple, Union

FloatArray = Union[np.ndarray, Tuple[float, float, float], list]

def accel_correct(a_com: FloatArray,
                  alpha: FloatArray,
                  omega: FloatArray,
                  r: FloatArray) -> np.ndarray:
    """
    加速度計のオフセット補正（重力補正なし）

    Parameters
    -----
    a_com: 形状(3,) の加速度ベクトル [m/s^2]
    alpha: 形状(3,) の角加速度ベクトル [rad/s^2]
    omega: 形状(3,) の角速度ベクトル [rad/s]
    r: 形状(3,) の IMU からセンサ原点へのオフセットベクトル [m]

    Returns
    -----
    np.ndarray
    補正済みセンサ加速度（重力なし） [m/s^2]
    """
    a_com = np.asarray(a_com, dtype=np.float64).flatten()
```

```

alpha    = np.asarray(alpha, dtype=np.float64).flatten()
omega    = np.asarray(omega, dtype=np.float64).flatten()
r        = np.asarray(r, dtype=np.float64).flatten()

if any(v.size != 3 for v in (a_com, alpha, omega, r)):
    raise ValueError("全ての入力は3次元ベクトルである必要があります")

coriolis    = np.cross(alpha, r)                # コリオリ加速度
centripetal = np.cross(omega, np.cross(omega, r)) # 遠心加速度
a_sensor    = a_com + coriolis + centripetal     # 補正後加速度
return a_sensor

```

設計ガイドライン（実践的）

- 主要IMU：胴体質量中心付近に搭載し、グローバルバランス制御の回転アーチファクトを低減。
- 頭部センサ：カメラ、ライダー、マイクをクラスタ化し配線と同期を簡素化するが、頭部慣性を考慮。
- 操縦センサ：触覚と手首 F/T センサをインターフェースに配置；把持改良のためハンドアイカメラを含める。
- 冗長性：臨界センサ（IMU、関節エンコーダ）を複製し相互チェックで故障を検出。
- キャリブレーションアクセス：フィデューシャル可視性と保守アクセスを確保し内部／外部パラメータ再キャリブレーションを実施。

技術的影響、トレードオフ、リスク

- アクチュエータとセンサを共配置するとケーブル長は削減されるが EMI と熱暴露が増大；信号整合性に対する機械的簡潔さをトレード。
- IMU を CoM に搭載するとグローバル推定が向上するが肢振動の直接計測は不可；必要に応じて局所 IMU を追加。
- カメラベースラインを増やすと深度精度が向上するが頭部慣性と転倒リスクが上昇；知覚利得と動的安定性をバランス。
- 時刻同期誤差と搭載不確実性は系統的状態推定バイアスを生じ；厳密なキャリブレーションとハードウェア同期がこれらのリスクを軽減する。

5.2 電力システム：バッテリー対配線

センサの配置決定はハーネスルーティング、コネクタ数、ローカル電力タップを左右するため、バッテリー電源とテザード電源の選択はセンサレイアウトと熱予算を直接的に制約する。以下の分析は、モビリティ、熱・電気性能、運用上の安全性との間の制約付きトレードオフとして工学選択を位置づける。

問題定義. ヒューマノイド設計者はアクチュエータの連続・ピーク電力を満たし、ミッション継続時間を算出し、導体と保護装置をサイズ決めしなければならない。アクチュエータが電力を支配し、知覚、演算、熱サブシステムが定常負荷を追加する。主要な工学上の問い：

- ・歩行、操縦、アイドル時にロボットが要求する連続・ピーク電力は？
- ・ミッションプロファイルに要求される継続時間は？
- ・オンボードエネルギーストレージに対する質量、体積、配置制約は？
- ・テザリングはモビリティ、安全性、電磁干渉にどう影響するか？

技術分析. エネルギー予算から始める。ミッション継続時間 t と平均電力 P_{avg} に対し、要求エネルギーは

$$[H]E_{\text{req}} \approx P_{\text{avg}} t. \quad (36)$$

公称電圧 V_{nom} でのバッテリー容量（アンペア時）は

$$[H]Q_{\text{Ah}} = \frac{E_{\text{req}}}{V_{\text{nom}}}. \quad (37)$$

エネルギーを質量に変換するには比エネルギー e_{spec} (Wh kg^{-1}) を用いる：

$$[H]m_{\text{batt}} = \frac{E_{\text{req}}}{e_{\text{spec}}}. \quad (38)$$

例： $P_{\text{avg}} = 300 \text{ W}$ 、要求継続時間 $t = 1 \text{ h}$ の中型ヒューマノイドは $E_{\text{req}} = 300 \text{ Wh}$ を要する。 $e_{\text{spec}} = 200 \text{ Wh kg}^{-1}$ の Li-ion ではバッテリー質量は 1.5 kg 。動的挙動のピーク電力は平均の 3～10 倍に達することが多く、ピーク電流を電力電子とケーブルサイズ決めに考慮する。

ケーブル電圧降下と導体サイズ決めはテザード・オンボード配電にとって決定的である。導体長 L 、断面積 A 、材料抵抗率 ρ の単導体における直流電圧降下は

$$[H]\Delta V = IR = I \frac{\rho L}{A}. \quad (39)$$

実用的なハーネス設計では、ピーク電流下で ΔV がコンバータ入力許容値内に収まるように A を選ぶ。交流・スイッチング電流では高周波での表皮効果・近接効果を含め、捻り対や編組シールドでループ面積を減らす。

運用・制御への影響：

1. 質量分布：オンボードバッテリー配置は重心と慣性特性に影響。バランス制御器の攪乱を最小化するため骨盤近くに配置。
2. ピーク電力処理：スーパーキャパシタなどのローカルエネルギーバッファを用いてピークを削り、バッテリー C レートストレスを低減。これにより電圧サグを抑えバッテリー寿命を延ばす。
3. 熱管理：大電流経路は発熱する。バッテリーとモータコントローラからシャーシ放熱器への熱経路を設計し、安全温度を維持。
4. 安全性：セルバランシング、過電流、熱遮断を備えたバッテリー管理システム（BMS）を実装。テザードには緊急クイックディスコネクトとスリップリリースを追加し、引っかかり危険を防止。

テザード対バッテリーの実装指針：

- ・テザード電源
 - －利点：事実上無限のエネルギー、オンボード質量減少、保守簡素化、信頼性の高い高ピーク電力供給。
 - －欠点：運用領域制限、ケーブルドラッグ・接触力、絡まりリスク、動作計画で補償が必要な動特性の変化。

- バッテリ電源

- 利点：完全なモビリティ、フィールド展開簡素化、ワークスペースへの制約少。
- 欠点：質量・体積増加、ミッション時間制限、BMS と充電ロジスティクス必要、電力電子の設計複雑性増大。

実用方程式とチェック. 許容電圧降下率 α (電源 V_{nom} に対する) から必要導体断面積を算出：

$$[H]A \geq \frac{I \rho L}{\alpha V_{\text{nom}}}. \quad (40)$$

これをテザード・内部バスの両方の給電線サイズ決めに用いる。低電圧システムでは α を小さく選ぶ：例えば 2%許容降下なら $\alpha = 0.02$ 。

最小ソフトウェアツール. 以下のリストはミッションパラメータを入力としてバッテリ容量、推定質量、必要導体面積を計算する。初期設計段階で迅速なトレードスタディに使用。

コードサンプル 16 ヒューマノイド設計用の簡易バッテリ・導体サイズ決め

```
#!/usr/bin/env python3
"""
バ ッ テ リ ・ ケ ー ブ ル サ イ ズ 計 算 ( ヒ ュ ー マ ノ イ ド ロ ボ ッ ト 用 )
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Final

# 物理定数
RHO_CU: Final[float] = 1.68e-8 # 銅の抵抗率 [ $\Omega \cdot \text{m}$ ]

@dataclass(frozen=True)
class MissionSpec:
    """ ミ ッ シ ョ ン 仕 様 """
    p_avg_w: float # 平均消費電力 [W]
    duration_h: float # 継続時間 [h]
    v_nom_v: float # 公称バス電圧 [V]
    e_spec_wh_kg: float # バッテリ質量エネルギー密度 [Wh/kg]
    l_wire_m: float # 片方向ケーブル長 [m]
    i_peak_a: float # ピーク電流 [A]
    v_drop_ratio: float # 許容電圧降下率 [-]

@dataclass(frozen=True)
class SizingResult:
```

```

"""サイジング結果"""
energy_wh: float
capacity_ah: float
battery_mass_kg: float
min_area_mm2: float

def compute_battery_cable(spec: MissionSpec) -> SizingResult:
    """
    バッテリー容量と導体最小断面積を計算
    """
    energy_wh = spec.p_avg_w * spec.duration_h
    capacity_ah = energy_wh / spec.v_nom_v
    battery_mass_kg = energy_wh / spec.e_spec_wh_kg

    # 許容抵抗値から最小断面積を逆算
    r_max_ohm = (spec.v_drop_ratio * spec.v_nom_v) / spec.i_peak_a
    area_m2 = (RHO_CU * spec.l_wire_m) / r_max_ohm
    area_mm2 = area_m2 * 1e6

    return SizingResult(
        energy_wh=energy_wh,
        capacity_ah=capacity_ah,
        battery_mass_kg=battery_mass_kg,
        min_area_mm2=area_mm2,
    )

def main() -> None:
    spec = MissionSpec(
        p_avg_w=300.0,
        duration_h=1.0,
        v_nom_v=48.0,
        e_spec_wh_kg=200.0,
        l_wire_m=2.0,
        i_peak_a=30.0,
        v_drop_ratio=0.02,
    )

    res = compute_battery_cable(spec)

    print(f"Required energy: {res.energy_wh:.1f} Wh")

```

```

print(f"Battery_capacity:{res.capacity_ah:.2f}Ah_at_{spec.v_nom_v}V")
print(f"Estimated_battery_mass:{res.battery_mass_kg:.2f}kg")
print(f"Minimum_conductor_area:{res.min_area_mm2:.2f}mm²")

if __name__ == "__main__":
    main()

```

工学上の影響とトレードオフ：

- ・高いバス電圧を用いると導体断面積と I^2R 損失が減るが、絶縁・安全性・コンバータ複雑性が増す。
- ・ローカル DC-DC 変換を備えた分散電源アーキテクチャはハーネス質量を減らすが、熱ホットスポットと制御複雑性が増す。
- ・テザードは電力を容易にするが動的負荷を課す；テザード力センシングと動作計画制約を含め、引っかかり・転倒を防止。
- ・バッテリー老化と C レート制限は性能を劣化させる；マージンと予測 BMS テレメトリをもってミッション計画を設計。

システム要求事項に記載すべき運用上のリスク：

- ・ピーク挙動中の電圧サグによるアクチュエータストールまたはコントローラリセット。
- ・換気不良バッテリーの熱暴走；隔離・監視を強制。
- ・テザードの故障または引っかかりによる機械的攪乱と安全インシデント。
- ・高電流スイッチングによる EMI 結合がセンシティブセンサと演算に影響。

5.3 マイクロコントローラおよびプロセッサ

前の小節では、オンボード処理の配置と熱予算に直接影響を与える電力のトレードオフとセンサ配置の制約を確立した。したがって、マイクロコントローラおよびプロセッサの選択は、利用可能な電力、センサ帯域、およびヒューマノイドの物理的レイアウトを考慮する必要がある。

マイクロコントローラ (MCU) およびアプリケーションプロセッサは、ヒューマノイドロボットにおいて異なるが重複する役割を果たす。MCU は、モータの整流、電流制御、エンコーダ復号、安全インターロックなど、決定的で低遅延のタスクを実装する。アプリケーションプロセッサおよびシステムオンチップ (SoC) は、知覚、計画、全身最適化、および調整など、計算負荷の高いタスクを処理する。工学上の課題は、制御、センシング、および計算タスクを、リアルタイムデッドラインを満たしながら、質量、体積、および電力予算を考慮して分割することである。

主要な選択基準：

- ・リアルタイム決定性：マイクロ秒からミリ秒単位の保証を必要とするタスクには、MCU または RTOS 対応コアを選択する。
- ・計算スループット：センサ処理、制御ループ、および知覚スタックに十分な CPU サイクルを確保する。
- ・I/O およびバスサポート：必要なインターフェースには、CAN、EtherCAT、SPI、I2C、UART、およ

びカメラや LiDAR 用の高速 PCIe または USB が含まれる。

- 電力および熱制限：プロセッサの TDP は、利用可能なバッテリーまたはテザー供給およびシャーシ冷却に適合する必要がある。
- 安全機能：ハードウェアウォッチドッグ、ECC メモリ、ハードウェア分離、およびセキュアブート。
- エコシステムおよびツール：ミドルウェアドライバ、SDK（例：NVIDIA Jetson ライブラリ）、およびデバッグツールの可用性。

実用的な工学解析は、タスク要求から始まる。 N 個の駆動関節を f_{loop} Hz で制御するヒューマノイドにおいて、関節制御更新あたり平均 C_{cycle} CPU サイクルを仮定すると、1 秒あたりに必要な生 CPU サイクルは概ね：

$$[H]\text{CPU_cycles_s} \approx N \cdot C_{\text{cycle}} \cdot f_{\text{loop}}. \quad (41)$$

例えば、30 関節、 $f_{\text{loop}} = 1000$ Hz、 $C_{\text{cycle}} = 20,000$ サイクルでは、1 秒あたり約 6×10^8 サイクルとなり、低レベル制御に少なくとも 1 GHz コア 1 個を専用する必要がある。軌道生成、状態推定、およびセンサフュージョンを別コアにオフロードすると、遅延リスクが軽減される。

センサおよびデータ帯域も予算化する必要がある。総センサ帯域は：

$$B = \sum_i r_i \cdot s_i,$$

ここで、 r_i はサンプルレート、 s_i はバイト単位のサンプルサイズである。30 Hz の圧縮ストリームを持つステレオカメラペアはこの帯域を支配し、ビジョン処理を SoC オンボードにするか別ビジョンモジュールにするかに影響を与える。

ヒューマノイドで一般的に使用されるアーキテクチャパターン：

1. 分散型低レベルコントローラ：関節または四肢ごとに 1 個の MCU が高速ループおよび安全を処理し、決定的バス（EtherCAT または CAN-FD）を介して中央プロセッサと通信する。
2. 集中型高レベル計算：マルチコア SoC（例：ARM Cortex-A または NVIDIA Jetson）が ROS2、知覚ネットワーク、およびプランナーを実行する。
3. ハードウェアアクセラレータ：GPU、NPU、または FPGA がビジョン、ニューラル推論、およびカスタム信号処理を加速する。
4. 混合クリティカリティ分割：安全クリティカルタスク用 RTOS、非クリティカル高レベル機能用 Linux；保護境界を介した IPC またはネットワークソケットを使用する。

実装例：関節トルク制御用の割り込み駆動 MCU ループが CAN 経由でコマンドを送信する。スニペットは厳密なタイミング、エンコーダ読取り、PID、およびウォッチドッグ給餌を示す。ペリフェラル呼び出しをプラットフォーム固有の HAL 関数に置き換える。

コードサンプル 17 ARM MCU 上の割り込み駆動トルク制御ループ

```
cpp
#include <cstdint>
#include <atomic>
#include <array>
#include <chrono>
```

```

#include "stm32f4xx_hal.h"           // HAL ベースの移植性
#include "can.h"
#include "encoder.h"
#include "pid.h"

namespace {
constexpr uint32_t CTRL_FREQ_HZ = 1000;           // 1 kHz 周期制御
constexpr uint32_t CAN_TIMEOUT_MS = 2;           // CAN 送信タイムアウト
constexpr uint32_t WD_REFRESH_WIN = CTRL_FREQ_HZ; // ウォッチドッグ更新区間

std::atomic<bool> control_flag{false};           // 割込み→メイン連携
PIDController    pid;                          // ゲインは別途初期化
Encoder          enc;                          // エンコーダ抽象層
CANBus           can;                          // CAN 抽象層

uint32_t loop_cnt = 0;                        // 周期カウンタ
} // namespace

// 1 kHz タイマ更新割込み
extern "C" void TIM1_UP_TIM10_IRQHandler() {
    if (__HAL_TIM_GET_FLAG(&htim1, TIM_FLAG_UPDATE) != RESET) {
        __HAL_TIM_CLEAR_FLAG(&htim1, TIM_FLAG_UPDATE);
        control_flag.store(true, std::memory_order_release);
    }
}

int main() {
    HAL_Init();
    SystemClock_Config();                // クロック 168 MHz 等
    MX_GPIO_Init();
    MX_TIM1_Init();                      // 1 kHz PWM ベースタイマ
    MX_CAN1_Init();
    MX_ADC1_Init();

    pid.init(0.5f, 0.01f, 0.0f);        // Kp, Ki, Kd (例)
    enc.init();
    can.init(1'000'000);                // 1 Mbps
    IWatchdog::init(10);                // 10 ms タイムアウト
}

```

```

HAL_TIM_Base_Start_IT(&htim1);          // 周期割込み開始

while (true) {
    if (control_flag.exchange(false, std::memory_order_acquire)) {
        const int32_t pos = enc.read(); // 高速、低ジッタ
        const int32_t vel = enc.differentiate(pos);
        const float tau = pid.update(static_cast<float>(pos),
                                       static_cast<float>(vel));

        can.send_torque(tau, CAN_TIMEOUT_MS);

        if (++loop_cnt % WD_REFRESH_WIN == 0) {
            IWatchdog::refresh();          // 安全：定期リフレッシュ
        }
    }

    // 空き時間に非クリティカル処理
    can.poll_noncritical();
    Logger::maybe_flush();
}
}

```

設計および統合上の考慮事項：

- 通信決定性：高帯域、低ジッタ駆動には EtherCAT またはリアルタイムイーサネットを使用する。CAN は低帯域駆動ノードで依然として有効である。
- 同期：ハードウェアタイムスタンプおよび PTP/NTP 統合はセンサフュージョン誤差を軽減する。
- 熱スロットリング：CPU および GPU は周波数スケーリングを示す；最悪ケースタイミングは持続負荷下の熱スロットリングを考慮する必要がある。
- EMI およびグラウンディング：プロセッサスイッチングノイズはエンコーダ信号を破損させることがある；アナログフロントエンドを分離し、適切なフィルタリングを適用する。
- 冗長性およびグレースフルデグラデーション：クリティカルセンサを複製し、安全なリムプホーム動作のためのフォールバックコントローラを提供する。

トレードオフおよび運用上のリスク：

- 集中型計算はアルゴリズム開発を容易にするが、故障モードを集中させ、冷却要件を増加させる。
- 分散型 MCU はケーブル長およびジッタを削減するが、ソフトウェア更新およびバージョン管理を複雑にする。
- 高パフォーマンス SoC（例：NVIDIA Jetson）の使用はオンボード学習および知覚を可能にするが、慎重な電力予算化および MCU へのリアルタイムブリッジングを必要とする。
- RTOS および Linux の混在は複雑性を増加させ；不適切な分離はデッドラインミスをリスクさせ、

安全でないアクチュエータコマンドを引き起こす可能性がある。

具体的な工学上の影響：CPU および帯域要件を早期に定量化し、クリティカルリティに従ってタスクをハードウェアにマッピングし、最悪ケース負荷下で熱動作をプロトタイプする。決定的タイミングまたは電力マージンを無視すると、ヒューマノイド動作における運動不安定性、知覚の劣化、および潜在的な安全危険が生じる。

5.4 通信システム

メイン CPU の計算予算を見積もり、アクチュエータの電力余裕を確保した後、通信バックボーンはロボットの熱および EMI 制約を尊重しながら、決定的かつ高帯域でセキュアなリンクを実現しなければならない。以下では、ヒューマノイドロボットのオンボードおよびオフボード通信のための原理的な設計を展開し、制御およびセンシング要件を具体的なネットワーク選択、レイテンシ予算、実装パターンに翻訳する。

問題定義と運用制約

- ・目的：ヒューマノイドの電気アーキテクチャを横断して、センサデータ、アクチュエータコマンド、テレメトリを有界レイテンシ、最小ジッター、EMI に対する耐性をもって確実に配信する。
- ・制約：質量および電力の制限、混在クリティカルティトラフィック（安全臨界低レイテンシ対高帯域知覚）、モバイル無線接続、遠隔運用のためのサイバーセキュリティ要件。

技術分析

- ・トラフィッククラスとマッピング：
 1. 安全臨界制御：関節位置・トルクセットポイント、低レベル状態フィードバック。決定的レイテンシと低ジッターを要求。
 2. 知覚ストリーム：カメラ、LIDAR、深度センサ。高帯域を要求するが、高レイテンシと偶発的ドロップを許容。
 3. テレメトリおよびロギング：非リアルタイム、ベースステーションまたはクラウドへの一括転送。
 4. 管理および診断：ファームウェアアップデートおよびデバッグのためのセキュアチャネル。
- ・ネットワークトポロジ：階層的セグメンテーションを用いる。
 1. 低レベル決定的バス（バス A）：サブミリ秒レイテンシを満たさなければならないアクチュエータおよびセンサのために EtherCAT または CAN FD を採用。EtherCAT は分散クロックで $< 1\text{ ms}$ のサイクルタイムをサポート。
 2. 高帯域バックボーン（バス B）：知覚およびミドルレベル制御の同期のため TSN（Time-Sensitive Networking）付きギガビットイーサネット。
 3. 無線アップリンク：リモートモニタリングのため Wi-Fi 6 または 5G、ファイアウォールおよび VLAN によって安全臨界リンクから隔離。
- ・タイミング、同期、および制御安定性：周波数 f_c で動作する制御ループについて、サンプリング周期は $T_s = 1/f_c$ 。通信レイテンシと処理遅延は T_s より十分小さくすべき；保守的設計目標は

$$L_{\text{total}} \leq 0.1 T_s, \quad (42)$$

ここで L_{total} はセンサ読み取りからアクチュエータ更新までの一往復遅延。離散時間コントロー

- ラの安定性解析では、位相余裕の侵食を避けるため決定的ジッターを T_s の 10%未満に維持する。
- 帯域幅計算：センサレートから必要なバックボーン帯域幅を推定。ステレオ RGB カメラペアについて：

$$B_{\text{cams}} \approx 2 \cdot R_x \cdot R_y \cdot f_{\text{fps}} \cdot b_{\text{pp}} \quad (\text{bits/s}), \quad (43)$$

- ここで $R_x \times R_y$ は解像度、 f_{fps} は毎秒フレーム数、 b_{pp} は圧縮後のピクセルあたりビット数。例：12 bpp 圧縮で 30 Hz の 720p カメラ 2 台は概算 $2 \cdot 1280 \cdot 720 \cdot 30 \cdot 12 \approx 0.66 \text{ Gbit/s}$ 。プロトコルオーバーヘッドおよび追加センサをリンク容量プロビジョニングに考慮。
- 無線リンク計画：目標スループットに対して必要な SNR を Shannon 容量で推定：

$$C = B \log_2(1 + \text{SNR}), \quad (44)$$

フェージングマージンおよび再送回復によるレイテンシ増加を見積もる。遠隔遠隔操作では、屋内運用時は低レイテンシ帯 (sub-6 GHz) を優先。

実装パターン

- セグメンテーションとゲートウェイ：
 1. EtherCAT/CAN FD ネットワークをモータおよびそのドライバの物理的に近くに配置。
 2. 決定的ネットワークをバックボーンへ、レート制限を実施し不正パケットをフィルタし、リンク喪失時に安全状態へ移行するウォッチドッグを実装するハードンドゲートウェイ経由でブリッジ。
 3. 知覚・ロジックホストで ROS2 を実行し、センサおよび制御トピックを優先するよう DDS QoS を設定。
- クロッキングとタイムスタンプ：IEEE 1588 PTP または EtherCAT 分散クロックを使用してノードを同期。NIC でのハードウェアタイムスタンプはジッターを削減しセンサフュージョンの時刻整合を改善。
- セキュリティと安全：
 1. 無線インタフェースを隔離。安全臨界トラフィックを無線経由でルーティングしない。
 2. 相互認証 (TLS または DTLS) および署名付きファームウェアを使用。
 3. ゲートウェイ切断時にフェイルセーフ動作を実装。

実装例：シンプル UDP テレメトリパブリッシャ

- この Python スニペットは、非臨界モニタリングのための軽量テレメトリパブリッシャを示す。タイムスタンプ、シーケンス番号、IMU サンプルをコンパクトなバイナリ UDP パケットに詰める。本番では認証済み DTLS に置き換え、パケットロス戦略を考慮。

コードサンプル 18 UDP telemetry publisher for on-board monitoring

```
#!/usr/bin/env python3
import socket
import struct
import time
```

```

import logging
import os
from typing import Tuple

# ログ設定
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s.%(msecs)03d_[(levelname)s]_(message)s',
    datefmt='%H:%M:%S'
)
logger = logging.getLogger(__name__)

# 環境変数から宛先を読み込み、無ければデフォルト
DEST_IP   = os.getenv('TELEM_DEST_IP', '192.168.1.100')
DEST_PORT = int(os.getenv('TELEM_DEST_PORT', '14550'))
SEND_HZ   = int(os.getenv('TELEM_HZ', '100'))           # 送信周波数
IMU_DEV   = os.getenv('IMU_DEV', '/dev/imu')
# IMUデバイスパス

# IMU読み出しダミー（本番は専用ライブラリに置換）
def get_imu() -> Tuple[float, float, float]:
    """IMUから加速度を取得（ダミー実装）"""
    return (0.0, 0.0, 9.81)

def main() -> None:
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.settimeout(1.0)
    # 送信タイムアウト
    addr = (DEST_IP, DEST_PORT)
    seq  = 0
    dt   = 1.0 / SEND_HZ

    logger.info('UDP_telemetry_started -> %s:%d@%dHz', *addr, SEND_HZ)

    try:
        while True:
            t0 = time.perf_counter()
            t  = time.time()
            try:
                ax, ay, az = get_imu()

```

```

except Exception as e:
    logger.error('IMU_read_fail:%s', e)
    continue

pkt = struct.pack('<Idfff', seq, t, ax, ay, az)
try:
    sock.sendto(pkt, addr)
except socket.error as e:
    logger.warning('sendto_fail:%s', e)

seq = (seq + 1) & 0xFFFFFFFF
# 32bit 循環
elapsed = time.perf_counter() - t0
time.sleep(max(0, dt - elapsed))
except KeyboardInterrupt:
    logger.info('shutting_down')
finally:
    sock.close()

if __name__ == '__main__':
    main()

```

6 機械系と電気系の統合

6.1 同期化の課題

これらの点は、前の小節で扱ったアクチュエータ選択、センサ配置、通信の話題に直接基づいている。以下では、人型システムにおいて高速機械制御と分散エレクトロニクスを組み合わせる際に生じるタイミングとデータアライメントの問題を分析する。

人型ロボットでは、複数のドメインにわたって厳密に結合された制御ループが必要である。アクチュエータ指令、ジョイントエンコーダ、慣性計測ユニット (IMU)、力センサ、高帯域ビジョンストリームを時間的に整列させ、安定したバランス、コンプライアントな相互作用、協調した全身運動を実現しなければならない。したがって同期化の問題定義は、異種ノード間でタイムスタンプと制御動作が相互に一貫し、制御安定性と状態推定精度を保持する遅延とジッタの範囲内に留まることを保証することである。

技術分析

- タイミング誤差の源：

1. プロセッサ間のクロックオフセットとドリフト。クロックドリフトは通常百万分率 (ppm) で規定される。時刻 t 後の最悪ケースのドリフトは $\Delta t \approx D \cdot t / 10^6$ であり、 D は ppm で

ある。

2. Ethernet、CAN、EtherCAT などのバスにおけるネットワーク遅延とジッタ。
3. OS カーネルおよびミドルウェアスタックにおけるソフトウェアキューイング遅延。
4. センササンプリングのミスアライメントと ADC 変換による未知の位相オフセット。

• 制御と推定への影響：

1. センサとコントローラ間のタイムスタンプ誤差 e_t は、周波数 ω で位相遅れ $\phi(\omega) = \omega e_t$ を生じる。コントローラのクロスオーバー ω_c で位相マージン M_ϕ を保持するため、最大許容総遅延 τ_{\max} は

$$\omega_c \tau_{\max} < M_\phi \Rightarrow \tau_{\max} < \frac{M_\phi}{\omega_c}. \quad (45)$$

を満たさなければならない。例えば、 $\omega_c = 2\pi \cdot 50 \text{ rad/s}$ 、 $M_\phi = 30^\circ = 0.524 \text{ rad}$ のトルクレベルコントローラは $\tau_{\max} \approx 1.7 \text{ ms}$ を生じる。同期化誤差はこの境界の小さな割合でなければならない。

2. サンプリングのミスアライメントによるセンサ融合バイアス。位置サンプル $p(t)$ とエンコーダサンプルが e_t ずれると、有限差分速度推定は e_t/T_s に比例した誤差を被り、 T_s はサンプリング間隔である。これはオドメトリとバランス推定器を劣化させる。
3. 協調駆動の危険。同期化されていない四肢が運動プリミティブを実行すると、内部衝突または不安定なトルクを生じる。

実用的な対策とアルゴリズム

• ハードウェアクロック同期：

1. Ethernet 上でサブマイクロ秒同期を実現するため、ハードウェアタイムスタンプを用いた Precision Time Protocol (PTP/IEEE 1588) を使用する。
2. モーションクリティカルなバスについては、決定的なフィールドバス (EtherCAT、時間トリガ拡張付き CAN-FD) を優先する。

• タイムスタンプ規律：

1. 生センササンプルにハードウェアキャチャに可能な限り近いタイムスタンプを付与する。
2. ハードウェアタイムスタンプを非決定的ユーザスレッドで再割り当てせず、ミドルウェアを通じて伝播させる。

• ソフトウェアレベルアライメント：

1. バッファリングと補間によりセンサストリームを共通参照時刻に整列させる。
2. 既知の遅延をモデル化した遅延補償推定器を使用する。

• 頑健性戦略：

1. 受動性または頑健制御手法を用いて制御ループが有界ジッタを許容するように設計する。
2. 同期化が閾値を超えた場合に安全に停止またはトルクオフするウォッチドッグを実装する。

実装例以下の Python スニペットは、IMU とエンコーダメッセージをバッファし、要求されたタイムスタンプに対して線形補間を行う軽量同期器を概説する。この手法は、状態推定または制御モジュールにデータを供給する前にタイムスタンプキューを削減する。

コードサンプル 19 IMU およびエンコーダストリーム用の簡易タイムスタンプバッファ・補間器。

```

import time
from typing import Optional, Dict, Tuple, Any
import numpy as np
from collections import deque
import threading

class SensorBuffer:
    """
    時系列データを保持し、線形補間で任意時刻の値を返すスレッドセーフなバッファ。
    """
    def __init__(self, maxlen: int = 200) -> None:
        self._buf: deque[Tuple[float, np.ndarray]] = deque(maxlen=maxlen)
        self._lock = threading.Lock()

    def append(self, t: float, value: np.ndarray) -> None:
        with self._lock:
            self._buf.append((t, value.copy()))

    def interpolate(self, t_query: float) -> Optional[np.ndarray]:
        with self._lock:
            if len(self._buf) < 2:
                return None
            times = np.fromiter((t for t, _ in self._buf), dtype=float)
            if t_query < times[0] or t_query > times[-1]:
                return None
            idx = int(np.searchsorted(times, t_query))
            t0, t1 = times[idx - 1], times[idx]
            v0, v1 = self._buf[idx - 1][1], self._buf[idx][1]
            alpha = (t_query - t0) / (t1 - t0)
            return (1.0 - alpha) * v0 + alpha * v1

class SyncedSensorManager:
    """
    IMU・エンコーダのバッファを管理し、指定時刻で同期した観測を返す。
    """
    def __init__(self, maxlen: int = 200) -> None:
        self.imu_buf = SensorBuffer(maxlen=maxlen)
        self.enc_buf = SensorBuffer(maxlen=maxlen)

```

```

def on_imu(self, t_hw: float, acc_gyro: np.ndarray) -> None:
    self.imu_buf.append(t_hw, acc_gyro)

def on_encoder(self, t_hw: float, q: np.ndarray) -> None:
    self.enc_buf.append(t_hw, q)

def get_synced(self, t_ref: float) -> Optional[Dict[str, Any]]:
    imu = self.imu_buf.interpolate(t_ref)
    enc = self.enc_buf.interpolate(t_ref)
    if imu is None or enc is None:
        return None
    return {"t": t_ref, "imu": imu, "enc": enc}

```

設計トレードオフと工学への影響

- 遅延 vs 決定性: 高スループット Ethernet は帯域を与えるが、決定性を得るには PTP とハードウェアタイムスタンプが必要。EtherCAT はより厳密なタイミングを与えるが柔軟性を低下させる。
- バッファサイズ vs 応答性: 大きなバッファはジッタを平滑化するが実効遅延を増加させる。式 (45) の τ_{\max} 境界内に追加遅延を抑えるようバッファを選ぶ。
- 複雑さ vs 安全性: 遅延認識推定器と補間を実装するとソフトウェア複雑度が増す。しかし、そうしないと不安定および安全でない動作のリスクがある。
- 運用上のリスク: 同期化されていないセンサは状態推定のゆっくりしたドリフト、突発的のトルクスパイク、またはバランス失敗を引き起こす。同期化メトリクスを監視し、閾値違反時に安全状態を強制することで軽減する。

主要な工学プラクティス

- システムレベル設計時に期待されるコントローラ帯域と式 (45) を用いて同期化精度要件を指定する。
- モーションクリティカルコンポーネントにはハードウェアタイムスタンプと決定的バスを優先する。
- センサ融合および制御スタックの一部としてソフトウェア補間と遅延補償を実装する。
- 同期化の健全性を継続的に監視し、安全なフォールバックを強制する。

これらの対策は、人型ロボットにおけるタイミング起因の不安定確率を低下させる。ハードウェアコストとソフトウェア複雑さと運用安全性および制御性能をトレードする。

6.2 効率を考慮した設計

前小節の同期制約は、アクチュエータとセンサ間のタイミングおよび位相アライメントを強調した。これらの同期の問題はエネルギー使用に直接影響するため、アライメントを改善する設計選択は無駄な仕事と消費を削減する。

効率を考慮した設計では、エネルギー消費をミッションあたり最小化するという工学的問題を構築する必要があるが、動的性能、安全性、信頼性の制約を満たす。運用上の関連性は、産業用ピックア

ンドブレースヒューマノイド、倉庫アシスタント、フィールドサービスロボットに現れ、ここではランタイムと熱限界がタスクスケジュールを支配する。効率は、機械アーキテクチャ、電気パワートレイン、熱管理、および制御アルゴリズム全体で考慮されなければならない。

問題定義. 関節軌道で定義される運動タスクが与えられたとき、トルク、帯域幅、および安定性の制約下で供給総エネルギーを最小化するアクチュエータ構成、伝達比、および制御方針を見つける。実用的な目的はタスクあたりのエネルギーであり、しばしばペイロードと時間で正規化される。

主要な物理関係はエネルギー流れを定義する。関節における機械的瞬時電力はトルクに角速度を乗じたものに等しく、

$$[H]P_{\text{mech}}(t) = \tau(t)\omega(t), \quad (46)$$

DC モータの電気入力概ね

$$[H]P_{\text{elec}}(t) = V(t)I(t) \approx RI^2(t) + k_e I(t)\omega(t), \quad (47)$$

モータ定数 k_e 、巻線抵抗 R 、電流 I を用いる。全伝達効率 η_{tx} と電力電子効率 η_{pe} は電気入力と機械出力を結びつける。有用な性能指標はタスクエネルギー

$$[H]E_{\text{task}} = \int_0^T \frac{P_{\text{mech}}(t)}{\eta_{\text{tx}}\eta_{\text{pe}}} dt + E_{\text{standby}}, \quad (48)$$

であり、ここで E_{standby} は待機電子機器と熱損失をカバーする。

設計戦略とトレードオフ

- アクチュエータ選択:
 - 動的タスクには高トルク対慣性比のモータを優先する。
 - I^2R 損失を削減するため低巻線抵抗を選ぶが、重量と熱質量を考慮する。
 - 直結駆動と減速機付きソリューションを比較する。直結駆動は逆駆動性と回生回収を改善する。減速機システムはモータトルクを削減するが、反映慣性と伝達損失を増加させる。
- 伝達設計:
 - 高潤滑とアライメント品質を持つ低損失ギア（ハーモニックまたは遊星）を用いて η_{tx} を最大化する。
 - 周期的タスクのためにエネルギーを弾性的に蓄え返すシリーズエラスティックアクチュエータ（SEA）を考慮する。
 - タスクが広いトルク-速度エンベロップを持つときは可変ギアまたは準直結駆動を評価する。
- 機械的受動補助:
 - バネまたはコンプライアント要素を統合して重力からの定常トルクを相殺する。
 - 公称姿勢中に必要アクチュエータトルクを最小化するように負荷経路と質量分布を設計する。
- 電気および電力電子:
 - 減速相におけるエネルギー回復のため回生ブレーキを実装する。
 - 高効率 DC-DC コンバータおよび同期モータドライバを用いて変換損失を削減する。
 - センサおよびマイクロコントローラのためのデューティサイクリングを採用し、知覚レイテンシとエネルギー節約のバランスを取る。
- 熱管理:

- 効率的な放熱は低抵抗を維持し、出力制限を防ぐ。
- アクティブ冷却はエネルギー使用を増加させる；制約性能のコストと比較検討する。
- 制御およびスケジューリング:
 - エネルギー認識軌道最適化は制約下で E_{task} を最小化する。
 - 予測制御器は次のタスクを活用して四肢を事前配置し、ピーク電力を削減する。
 - 長期効率のためエネルギーコスト項を持つモデル予測制御（MPC）または強化学習（RL）を用いる。

一般的な工学定式化は制約付き最適化である：

$$[H] \min_{u(t), r} E_{\text{task}}(u, r) \quad \text{s.t.} \quad \begin{cases} \dot{x} = f(x, u), \\ \tau_{\min} \leq \tau(x, u, r) \leq \tau_{\max}, \\ \omega_{\min} \leq \omega(x, u, r) \leq \omega_{\max}, \\ \text{safety and interaction constraints,} \end{cases} \quad (49)$$

ここで $u(t)$ は制御入力、 r は設計パラメータ（ギア比、ばね剛性）、 x は状態である。(49) を解くことは機械サイジングと制御合成を結びつける。

実装チェックリスト（実践的）

1. タスクプロファイルを特徴付ける：速度、ペイロード、デューティサイクル。
2. 関節に必要なトルク-速度エンベロープを計算する。
3. 熱限界にマージンを持たせてエンベロープに一致するモータファミリを選択する。
4. 反映慣性と効率をバランスさせた伝達を設計する。
5. 周期的トルクが存在する場所に受動補助を統合する。
6. モータ損失モデルと現実的なコントローラを用いてエネルギーをシミュレートする。
7. Isaac Sim でのハードウェアインザループまたは高忠実度シミュレーションで検証する。

簡潔な例関数は軌道サンプルから概算タスクエネルギーを計算する。このツールはギア比とモータ選択を比較するトレードオフ研究を支援する。

コードサンプル 20 サンプルされた関節軌道のエネルギーを推定

```
import numpy as np
from typing import Union

Number = Union[int, float]

def estimate_energy(
    torque: np.ndarray,
    omega: np.ndarray,
    dt: float,
    R: Number,
    k_e: Number,
```

```

        eta_tx: float = 0.95
    ) -> float:
        """
    """
    電気エネルギー推定（単位：J）
    """
    """
    # 入力検証
    if not (torque.shape == omega.shape and eta_tx > 0 and dt > 0):
        raise ValueError("Invalid input shape or non-positive dt/eta_tx")

    # 機械パワー
    P_mech = torque * omega

    # 逆起電力と電流
    V_emf = k_e * omega
    I = torque / k_e # k_t == k_e 近似

    # 電気パワー（銅損＋電力変換）
    P_elec = R * I**2 + V_emf * I

    # 伝達効率を考慮した入力パワー
    P_in = P_elec / eta_tx

    # エネルギー積算
    return float(np.sum(P_in * dt))

```

工学への影響とリスク

- ・ギア減速を増加させるとモータ電流は低下するが、反映慣性が増加し敏捷性が低下する可能性がある。このトレードオフは転倒回復とコンプライアンスに影響する。
- ・重量削減を過度に重視すると熱容量が減少し、連続タスクで出力制限が生じる可能性がある。
- ・積極的なセンサデューティサイクリングは電力を節約できるが、知覚レイテンシを増加させ衝突リスクが高まる。
- ・回生は減速が多いタスクで節約をもたらす；戻されたエネルギーを安全に処理するためのトポロジと制御が必要である。
- ・機械および電気ドメインにわたる最適化は、孤立した改善よりも効率を改善する。

設計者はしたがってアクチュエータ、伝達、および制御を共最適化しなければならない。効率目標と堅牢性、安全マージン、保守性をバランスさせ、ヒューマノイドシステムで運用信頼性を確保する。

6.3 保守に関する考慮事項

効率性重視のレイアウト選択とアクチュエータ制御ループの同期によって課せられた制約を踏まえ、保守戦略はアクセシビリティと制御システムの整合性を明示的に調和させる必要がある。以下で

は、ヒューマノイドロボットの状態ベース保全に向けた、運用に焦点を当てた指針、定量的モデル、およびコンパクトな実装パターンを示す。

問題定義。ヒューマノイドロボットは高密度の機械サブシステムと分散電子機器を組み合わせている。保守計画は以下を満たす必要がある：

- ・ サービス中のダウンタイムと安全リスクを最小化すること；
- ・ 同期制御を支えるセンサおよびアクチュエータのキャリブレーションを保持すること；
- ・ 信号経路の劣化や EMI 問題を回避しながら高速診断を可能にすること。

技術的分析。

1. 故障モードと保守性指標。

- ・ 重要部品を特定する：アクチュエータ（モータ、ギアボックス）、ジョイントベアリング、電源ストレージ、高帯域センサ（LiDAR、カメラ）、IMU、通信バス。
- ・ 故障率モデルを用いて点検を優先付けする。多くの電子・機械サブシステムに対して定故障率モデルは初期近似として許容される。信頼度は

$$[H]R(t) = e^{-\lambda t}, \quad (50)$$

ここで λ は故障率、平均故障間隔（MTBF）は

$$[H]MTBF = \frac{1}{\lambda}. \quad (51)$$

- ・ 摩耗支配部品（ベアリング、ハーモニックドライブ）にはワイブル解析を用いる。電子機器では熱サイクルが λ を著しく増大させる。

2. アクセシビリティ対環境保護。

- ・ 設計上のトレードオフ：
 - 着脱式パネルおよびモジュラージョイントは平均修理時間（MTTR）を短縮するが、質量を増やし、侵入保護（IP）等級を低下させる可能性がある。
 - 密封型アクチュエータは過酷環境での寿命を延ばすが、現地サービスを複雑化し、交換部品コストを増大させる。
- ・ 標準化された機械インタフェースおよびキー付き電気コネクタを用いて、正しい再組立を保証し、コネクタ摩耗を最小化する。

3. 電氣的整合性の保守。

- ・ ケーブルルーティングは、重要な電源経路を切断することなく信号ハーネスを分離できるようにする。
- ・ 現地での戦術的はんだ修理を避け、大電流経路には圧着ロックコネクタを用いる。
- ・ EMI 対策：シールド連続性を維持し、バルクヘッド通過にはスルー容量を用いる。緩いシールドまたは劣化したグラウンドリターンは断続的なセンサエラーとして顕在化する。

4. キャリブレーションと診断。

- ・ センサキャリブレーションは観測されたドリフトと制御感度に基づいてスケジュールする。例えば、24 時間で 1°C ドリフトを生じる IMU バイアス不安定性は精密タスクで毎日の自動キャリブレーションを必要とする場合がある。
- ・ アクチュエータおよびセンサのための組込み自己試験（BIST）モードを提供する。BIST は

起動時と、異常検出のためのベースライン署名収集をアイドル期間中にオプションで実行する。

5. 状態ベース保全と予知解析。

- 以下の連続モニタリングを実装する：
 - モータ電流とトルクリップル、
 - ジョイント温度と熱勾配、
 - ベアリングおよびギアボックスの振動スペクトラ、
 - 通信バスでのパケットロスとレイテンシ。
- 帯域幅節約のため、小型エッジ解析パイプラインを用いて特徴量を計算し、ローカルで異常をフラグ付けする。

実装：スケジューリングとコストのトレードオフ。

- 区間 T における期待保守コストは

$$[H]E[C(T)] = C_p \left\lfloor \frac{T}{T_p} \right\rfloor + C_f(1 - R(T)), \quad (52)$$

ここで C_p は予防保全コスト/回、 T_p は予防間隔、 C_f は平均故障コストである。 $E[C(T)]$ を最小化することで実用的な点検周期が得られる。フリートテレメトリからの実証的 λ 推定値を用いてパラメータを調整する。

実用的診断ルーチン（コンパクト Python 例）。コードはモータ振動チャネルのローリング RMS を計算し、RMS が閾値を超えた際に保守アラートを発行する。オンロボットエッジコントローラ向けに軽量に保つ。

コードサンプル 21 振動ベース異常検出のためのエッジルーチン

```
#!/usr/bin/env python3
import logging
import math
import time
from collections import deque
from dataclasses import dataclass
from typing import Deque, Final

import board
import busio
import adafruit_adxl34x # I2C加速度センサ例

# ログ設定
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s.%(msecs)03d_%(levelname)s_%(message)s",
    datefmt="%Y-%m-%d_%H:%M:%S",
```

```

)
logger: Final = logging.getLogger("vibration_monitor")

# 定数
WINDOW: Final = 256          # ローリングウィンドウ長
THRESHOLD_G: Final = 0.75    # RMS閾値[g]
SAMPLE_HZ: Final = 100       # サンプリング周波数[Hz]
SLEEP_SEC: Final = 1.0 / SAMPLE_HZ

# センサ初期化
i2c = busio.I2C(board.SCL, board.SDA)
accel = adafruit_adxl34x.ADXL345(i2c)
accel.range = adafruit_adxl34x.Range.RANGE_16_G # ±16G

@dataclass(slots=True, frozen=True)
class VibrationEvent:
    rms: float
    timestamp: float = time.time()

class VibrationMonitor:
    def __init__(self, window: int = WINDOW, threshold_g: float = THRESHOLD_G) -> None:
        self._buf: Deque[float] = deque(maxlen=window)
        self._threshold: Final = threshold_g

    def _read_accel_magnitude(self) -> float:
        """ADXL345から加速度大きさ[g]を取得"""
        x, y, z = accel.acceleration
        return math.sqrt(x * x + y * y + z * z) / 9.80665 # m/s2 → g

    def _rolling_rms(self) -> float:
        """バッファ内のRMS[g]を計算"""
        n = len(self._buf)
        if n == 0:
            return 0.0
        return math.sqrt(sum(v * v for v in self._buf) / n)

    def _handle_alert(self, rms: float) -> None:
        """閾値超過時のログ出力と保守依頼"""

```

```

        logger.warning("VIBRATION_ALERT_rms=%3f_g", rms)
        # TODO: 保守システムへのREST呼出等を実装
        # trigger_maintenance_workorder()

def run(self) -> None:
    """メインループ"""
    logger.info("Vibration_monitor_started")
    while True:
        self._buf.append(self._read_accel_magnitude())
        if len(self._buf) == self._buf.maxlen:
            rms = self._rolling_rms()
            if rms > self._threshold:
                self._handle_alert(rms)
            time.sleep(SLEEP_SEC)

if __name__ == "__main__":
    try:
        VibrationMonitor().run()
    except KeyboardInterrupt:
        logger.info("Shutting_down")

```

統合とテストポイント。

- モータ相電圧、エンコーダ A/B、IMU SPI ライン、CAN バス終端状態といった主要信号のためのテストヘッダを提供する。これにより非侵襲的なオシロスコープおよびロジックアナライザプロービングが可能となる。
- 診断経路でアクセス可能なバージョンファームウェアおよびハードウェア ID テーブルを実装し、不整合部品更新を防ぐ。

運用上の影響とリスク。

- トレードオフ：
 1. モジュラ性は MTTR を短縮するが、部品数と故障し得る組立インタフェースを増やす。
 2. 密封は腐食環境での寿命を延ばすが、交換コストとサービス時間を増大させる。
 3. オンロボット解析はデータ伝送を減らす、CPU 負荷を増やし、潜在的に熱ストレスを増大させる。
- リスク：
 - 電源システムの保守遅延は熱暴走または安全ハザードを引き起こす可能性がある。
 - 不適切なケーブルルーティングは断続的信号損失を介して同期を劣化させ、前述の同期課題を悪化させる。
 - 度を超えた予防スケジュールはライフサイクルコストを増大させるが、それに見合う信頼性

向上をもたらさない。

具体的なエンジニアリングアクション。

- ・サブシステムごとの保守レベルオブエフォート (LoE) を定義し、MTTR および MTBF 目標にマッピングする。
- ・重要ジョイントに振動および温度計測を装備し、少なくとも複数の故障サイクルにわたってデータを収集して λ を実証的に推定する。
- ・コネクタ、予備部品キット、サービス手順をフィールド技術者向けに標準化する。
- ・まず状態ベース閾値を採用し、次にフリーテレメトリで洗練して不要なサービスを回避する。

これらの対策は、運用ダウンタイムを直接削減し、同期制御性能を保持し、安全リスクを軽減すると同時に、アクセシビリティ、保護、ライフサイクルコストの間のトレードオフを明確に露呈する。

6.4 一般的な問題のトラブルシューティング

前節の保守に関する考察および効率化のための設計では、摩耗パターン、熱限界、制御帯域幅を重要な統合制約として特定した。本小節では、これらの制約が実際のヒューマノイドシステムで相互作用するときに出現する繰り返しの機械・電気故障を診断し修正することに焦点を当てる。

問題定義：統合故障はしばしば断続的な故障、制御性能の低下、または漸進的なドリフトとして現れる。これらの症状は機械・電気インターフェース、すなわちセンサ、アクチュエータ、ハーネス、コネクタ、グラウンド、リアルタイムデータパスに由来する。目的は、エンジニアに構造化されたトラブルシューティングワークフロー、定量的診断、およびヒューマノイドプラットフォームでのファーストパス検出を自動化する軽量コードを提供することである。

一般的な故障モード、診断、および対策

- ・エンコーダの不整合とキャリブレーションドリフト
 - 症状：関節位置誤差が蓄積し、低速動作中に状態推定器が発散する。
 - 原因：緩んだマウント、熱膨張、またはインクリメンタルエンコーダインデックスの喪失。
 - 検出：重力補償後の指令位置と測定位置の間の残差を計算する。オブザーバベースの推定値 $\hat{\theta}$ と残差 $r = \theta_{\text{meas}} - \hat{\theta}$ を用いる。持続的な $|r| > \epsilon$ はキャリブレーションドリフトを示す。
 - 対策：自動エンコーダリホーミングを実行し、機械的ストップ検証と温度補償テーブルを追加する。
- ・アクチュエータの過熱と電流制限トリップ
 - 症状：トルク飽和、リンプモード、またはステップ応答の低下。
 - 原因：過度な摩擦、ストールしたギアボックス、またはアクチュエータの熱性能に不整合な制御ゲイン。
 - 検出：巻線温度と RMS 電流を監視する。時定数 τ_{th} を持つ指数熱モデルを用いて連続熱負荷を推定する。
 - 実装メモ： $T > T_{\text{safe}}$ のときは指令 RMS トルクを削減し、プラスチック変形を防ぐための冷却サイクルをスケジュールする。
- ・配線故障と断続的な短絡
 - 症状：過渡的なセンサドロップアウト、電圧サグ、または IMU チャンネルの EMI。

- 原因：摩耗したハーネス、コネクタの腐食、または不適切なストレインリリーフ。
- 検出：バス電圧とセンササンプリングレートをログする。ドロップアウトを関節動作と関連させる。相互相関を用いて動作相関電気ノイズを検出する。
- 対策：ハーネスを再配線し、フェライトを追加し、シールドバリエーションのコネクタに交換する。
- グラウンドループと EMI
 - 症状：アナログセンサのスプリアス読み値、リミットスイッチの誤トリガ。
 - 原因：複数のグラウンド基準点と大電流スイッチング過渡。
 - 検出：差動プローブを用いてサブアセンブリ間の同相電圧を測定する。
 - 緩和：シングルポイントグラウンドを実装し、コモンモードチョークを追加し、デジタルおよびアナロググラウンドプレーンを分離する。
- 通信層の問題（CAN、EtherCAT、UART）
 - 症状：パケットドロップ、タイムスタンプジッタ、またはコントローラタイムアウト。
 - 原因：バスアービトラージョン問題、リアルタイムタスクのアンダーラン、またはケーブル長／終端問題。
 - 検出：パケット CRC エラー率とエンドツーエンドレイテンシを監視する。周期制御ループでは、ジッタ σ_t がコントローラマージン以下であることを確認する。
 - 対策：リアルタイムタスクを優先し、決定論的バススケジューリングを追加し、トランシーバ終端を検証する。

定量的チェックと簡易式

- 肢体セグメントの重力トルク推定：エンジニアはアクチュエータトルク余力がペイロード下で重力トルクを超えることを検証しなければならない。単リンク肢体の場合、

$$[H]\tau_g(\theta) = mgl \sin(\theta), \quad (53)$$

ここで、 m はセグメント質量、 l は重心距離、 θ は関節角度である。ヒューマノイド用途には安全係数 $\kappa \in [1.5, 2.5]$ で $\tau_{\text{rated}} \geq \kappa \max_{\theta} \tau_g(\theta)$ を検証する。

- サンプリングとエイリアシング制約：センササンプリング周波数 f_s が $f_s \geq 2f_{\text{max}}$ を満たすことを確認する。ここで、 f_{max} は制御対象の機械振動の最高周波数成分である。 f_{max} が不明な場合は、生センサトレースの PSD を調査して f_s を選定する。

実装：自動異常検出器以下はコンパクトな ROS2 スタイル診断ノードである。関節残差とモータ温度の指数移動平均を計算し、持続的な異常でアラートを発行する。

コードサンプル 22 ROS2 診断ノード：関節残差および温度監視

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
from sensor_msgs.msg import JointState
```

```

from diagnostic_msgs.msg import DiagnosticArray, DiagnosticStatus, KeyValue
import threading
import time
from typing import Dict, Optional

# パラメータ記述子（動的再設定対応）
RESID_THRESH_DESC = ParameterDescriptor(
    type=ParameterType.PARAMETER_DOUBLE,
    description='Residual_threshold[rad]',
    read_only=False)
TEMP_THRESH_DESC = ParameterDescriptor(
    type=ParameterType.PARAMETER_DOUBLE,
    description='Temperature_alert_threshold[degC]',
    read_only=False)
ALPHA_DESC = ParameterDescriptor(
    type=ParameterType.PARAMETER_DOUBLE,
    description='EWMA_factor',
    read_only=False)

class DiagnosticsNode(Node):
    def __init__(self):
        super().__init__('joint_diagnostics')
        self.declare_parameter('residual_threshold', 0.05, RESID_THRESH_DESC)
        self.declare_parameter('temperature_threshold', 65.0, TEMP_THRESH_DESC)
        self.declare_parameter('ewma_alpha', 0.02, ALPHA_DESC)

        self._resid_thresh = self.get_parameter('residual_threshold').value
        self._temp_thresh = self.get_parameter('temperature_threshold').value
        self._alpha = self.get_parameter('ewma_alpha').value

        self._sub = self.create_subscription(
            JointState, '/joint_states', self._cb_joint, 10)
        self._diag_pub = self.create_publisher(DiagnosticArray, '/diagnostics', 1)

        self._cmd_pos: Dict[str, float] = {}
        self._ema_temps: Dict[str, float] = {}
        self._lock = threading.Lock()

    # コマンド位置を受け取る（任意）
    self._cmd_sub = self.create_subscription(

```

```

        JointState, '/joint_commands', self._cb_cmd, 10)

# パラメータ変更コールバック
self.add_on_set_parameters_callback(self._on_params)

def _on_params(self, params):
    for p in params:
        if p.name == 'residual_threshold':
            self._resid_thresh = p.value
        elif p.name == 'temperature_threshold':
            self._temp_thresh = p.value
        elif p.name == 'ewma_alpha':
            self._alpha = max(0.0, min(1.0, p.value))
    return rclpy.parameter.SetParametersResult(successful=True)

def _cb_cmd(self, msg: JointState):
    with self._lock:
        for i, name in enumerate(msg.name):
            self._cmd_pos[name] = msg.position[i]

def _cb_joint(self, msg: JointState):
    now = self.get_clock().now()
    diag_array = DiagnosticArray()
    diag_array.header.stamp = now.to_msg()

    with self._lock:
        for i, name in enumerate(msg.name):
            pos = msg.position[i]
            effort = msg.effort[i] if i < len(msg.effort) else 0.0
            temp = self._estimate_temp(name, effort)

            prev = self._ema_temps.get(name, temp)
            ema = self._alpha * temp + (1.0 - self._alpha) * prev
            self._ema_temps[name] = ema

            cmd = self._cmd_pos.get(name, 0.0)
            residual = abs(pos - cmd)

            status = DiagnosticStatus()
            status.name = f'Joint:_{name}'

```

```

        status.hardware_id = name

    # レベル判定
    if ema > self._temp_thresh:
        status.level = DiagnosticStatus.ERROR
        status.message = 'High_motor_temperature'
    elif residual > self._resid_thresh:
        status.level = DiagnosticStatus.WARN
        status.message = 'Large_tracking_residual'
    else:
        status.level = DiagnosticStatus.OK
        status.message = 'OK'

    status.values = [
        KeyValue(key='position', value=f'{pos:.4f}'),
        KeyValue(key='effort', value=f'{effort:.4f}'),
        KeyValue(key='temperature_estimate', value=f'{ema:.2f}'),
        KeyValue(key='residual', value=f'{residual:.4f}')
    ]
    diag_array.status.append(status)

    self._diag_pub.publish(diag_array)

def _estimate_temp(self, name: str, effort: float) -> float:
    # 簡易温度推定: RMS effortに比例
    return 30.0 + 5.0 * abs(effort)

def main(args=None):
    rclpy.init(args=args)
    node = DiagnosticsNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

```

実践的なロボット上トラブルシューティングワークフロー

1. 管理された条件下で故障を再現する。アクチュエータ指令、センサトレース、周囲条件を文書化する。

2. ハードウェアインザループテストでサブシステムを分離する。上腿負荷を除去してトルク源を確認する。
3. 残差と PSD 解析で誤差を定量化する。少なくとも 10 サイクルの故障期間を最大サンプリングレートでログする。
4. 機械アライメント、ハーネス交換、EMI 抑制、ソフトウェアレート安定化を繰り返し適用する。
5. 温度サイクルにわたるストレステストで修正を検証する。

エンジニアリングへの影響、設計のトレードオフ、運用上のリスク

- ・制御ループ帯域幅を上げると応答性は向上するが、センサノイズと配線インピーダンスへの感度が高まる。追加フィルタリングと高分解能センサでバランスを取る。
- ・冗長センシングは単一故障点を減らすが、重量と配線の複雑さが増す。独立したグラウンドで冗長性を設計する。
- ・積極的なアクチュエータサイジングは熱ストレスを軽減するが、質量と電力要求が増す。ペイロード、バッテリー寿命、熱管理を統合的に評価する。
- ・運用上のリスク：未検出の断続的 EMI は安全でない作動を引き起こす可能性がある。保守的なフェイルセーフ状態とウォッチドッグタイマを実装する。

これらのトラブルシューティング手法は、機械系と電気系が密接に相互作用するヒューマノイドロボットにおいて、ダウンタイムを最小化し堅牢性を向上させる。

機械設計の基礎

7 機械設計のためのツール

7.1 CAD ソフトウェアの概要

すでに導入された機械的概念を基に、この小節では人間型キネマティクスおよび構造設計のためにそれらの概念を実装するソフトウェアツールを検討する。CAD の選択が製造可能性、シミュレーション精度、最終的なシステム性能にどのように影響するかを強調する。

エンジニアは質量、バランス、アクチュエータトルクという厳しい制約を満たすように人間型メカニズムを設計する。そのためソフトウェアは以下をサポートしなければならない：

- ・反復的な形状変更のためのパラメトリックモデリング；
- ・動的シミュレーションのための正確な質量・慣性計算；
- ・ジョイントクリアランスおよび動作検討のためのアセンブリレベルキネマティクス；
- ・製造可能な出力（図面，CAM 対応ジオメトリ，シミュレーション用エクスポート）。

問題定義．設計意図を以下を満たす検証済みデジタルモデルに変換せよ：

1. 設計意図をパラメータおよび制約として表現する；
2. FEA，動作シミュレーション，CAM に適したジオメトリを生成する；
3. 制御設計に信頼できる質量・慣性特性を与える。

技術分析．CAD プラットフォームはモデリングパラダイムとエコシステムサポートによって異

なる：

- ・パラメトリックソリッドモデラ（フィーチャベース）：SolidWorks, Creo, Siemens NX. 公差駆動部品, 詳細図面, 設計意図の取り込みに最適.
- ・ダイレクトおよびコンセプトモデリングツール：Onshape, Fusion 360. 初期段階の反復およびクラウドコラボレーションに高速.
- ・サーフェス／NURBS および高度なサーフェス：Rhino, Alias. 人間工学的外装および滑らかな肢カバーに有用.
- ・メッシュおよびスカルプティングツール：Blender, MeshLab. 知覚可視表面に有用だが, 機械用途には変換が必要.
- ・CAE 統合スイート：Ansys, Abaqus 統合またはツール内 FEA（SolidWorks Simulation）による応力, モード, 熱解析.
- ・デジタルツイン対応エクスポート：URDF, SDF, glTF, STEP ファイルへのネイティブサポートまたはプラグインで, Isaac Sim などの下流シミュレータに対応.

質量特性および慣性モデリングは重要である．剛体部品集合に対して，重心は

$$[H]\mathbf{r}_{\text{COM}} = \frac{1}{M} \sum_i m_i \mathbf{r}_i, \quad (54)$$

与えられる．ここで $M = \sum_i m_i$. リンク慣性をその自身の COM からジョイントフレームへ変換するには平行軸定理を用いる：

$$[H]\mathbf{I}_O = \mathbf{I}_{\text{COM}} + m (\|\mathbf{d}\|^2 \mathbf{I}_3 - \mathbf{d}\mathbf{d}^T), \quad (55)$$

ここで \mathbf{d} はリンク COM から目標原点へのベクトルであり， \mathbf{I}_3 は 3×3 単位行列である．

実装上の手順．現実的な人間型設計には，以下のパイプラインに従う：

1. 機能モジュールを定義する：胴体，骨盤，四肢，手，足．アセンブリはモジュール化して保持する．
2. ジョイント基準フレームおよび最小限のファスナ詳細を持つパラメトリックサブアセンブリを作成する．早段階で可動部品を過剰に詳細化しない．
3. CAD で材料および密度を割り当て，リンク質量および慣性を計算する．
4. 干渉および動作検討を実行し，可動範囲および自己衝突回避を確認する．
5. シミュレーションおよび知覚用にエクスポートするジオメトリを簡略化する：小さなフィレットを削除し，詳細ファスナを簡略化された体積に変換する．
6. ジオメトリおよび慣性データを制御用（URDF / SDF）およびシミュレーション用（STEP / glTF）フォーマットの両方にエクスポートする．

エクスポートおよび相互運用性の考慮事項：

- ・正確な CAD 交換には STEP を使用する；テクスチャ付き可視化には glTF または Collada を使用する．
- ・URDF は数値慣性値とコンパクトメッシュ（できれば凸または適切な面数でテッセレートされたもの）を要求する．

- メッシュファイルが水密であり、一貫したスケールおよび単位を持つことを確認し、シミュレーションアーチファクトを回避する。

自動化例. 以下の Python スニペットは、STL ファイルから質量特性を計算し、最小限の URDF 慣性ブロックを出力する。実用的なワークフロー向けの一般的なライブラリを使用する；CAD エクスポートメッシュおよび密度に合わせて適応させる。

コードサンプル 23 メッシュから質量・慣性を計算し URDF 慣性ブロックを書き出す。

```
#!/usr/bin/env python3
"""
URDF_inertial_tag_generator_from_STL_mesh.
STLメッシュからURDF用inertialタグを生成する。
"""

from __future__ import annotations

import argparse
import sys
from pathlib import Path
from typing import Tuple

import numpy as np
import trimesh

def load_mesh(path: Path) -> trimesh.Trimesh:
    """Load triangle mesh from file."""
    if not path.exists():
        raise FileNotFoundError(f"Mesh file not found: {path}")
    mesh = trimesh.load(str(path))
    if not isinstance(mesh, trimesh.Trimesh):
        raise TypeError("Loaded geometry is not a single triangle mesh")
    return mesh

def compute_inertial_params(
    mesh: trimesh.Trimesh,
    density: float,
    origin_offset: np.ndarray,
) -> Tuple[float, np.ndarray, np.ndarray]:
    """
```

質量・重心・慣性テンソルを計算

```
"""
```

```
mass = float(mesh.volume * density)
com = mesh.center_mass # mesh座標系
inertia_com = mesh.moment_inertia * density # 重心周り

# 平行軸定理で原点へ変換
d = origin_offset - com
I_o = inertia_com + mass * (np.dot(d, d) * np.eye(3) - np.outer(d, d))
return mass, com, I_o
```

```
def build_inertial_tag(mass: float, com: np.ndarray, inertia: np.ndarray) -> str:
```

```
    """URDF<inertial>タグ文字列を生成"""
```

```
    return f"""<inertial>
```

```
    <origin xyz="{com[0]:.6f} {com[1]:.6f} {com[2]:.6f}" rpy="0 0 0"/>
```

```
    <mass value="{mass:.6f}"/>
```

```
    <inertia ixx="{inertia[0,0]:.6f}" ixy="{inertia[0,1]:.6f}" ixz="{inertia[0,2]:.6f}"
```

```
    iyy="{inertia[1,1]:.6f}" iyz="{inertia[1,2]:.6f}" izz="{inertia[2,2]:.6f}"/
```

```
</inertial>"""
```

```
def main(argv=None):
```

```
    parser = argparse.ArgumentParser(description="Generate URDF<inertial> from STL")
```

```
    parser.add_argument("mesh", type=Path, help="Path to STL file")
```

```
    parser.add_argument(
```

```
        "--density",
```

```
        type=float,
```

```
        default=2700.0,
```

```
        help="Material density [kg/m^3] (default: 2700 for aluminum)",
```

```
)
```

```
    parser.add_argument(
```

```
        "--origin-offset",
```

```
        type=float,
```

```
        nargs=3,
```

```
        default=[0.0, 0.0, 0.0],
```

```
        metavar=("X", "Y", "Z"),
```

```
        help="Offset from mesh origin to link origin [m]",
```

```
)
```

```
    args = parser.parse_args(argv)
```

```

mesh = load_mesh(args.mesh)
mass, com, inertia = compute_inertial_params(
    mesh, args.density, np.array(args.origin_offset)
)
print(build_inertial_tag(mass, com, inertia))

if __name__ == "__main__":
    main()

```

設計上のトレードオフおよび運用上のリスク：

- 忠実度 vs 性能：高度に詳細な CAD はシミュレーションコストを増大させる。リアルタイム制御テスト用にジオメトリを簡略化する。
- パラメータ化 vs 速度：深いパラメトリックツリーは後段変更を容易にするが、モデル複雑性およびビルド時間を増大させる。
- 慣性精度：質量分布の誤差はバランスコントローラを損ない、不安定な歩容挙動を引き起こす可能性がある。
- エクスポート整合性：誤った単位選択または反転法線は、シミュレーションで誤った質量またはセンサ遮蔽を生じる。
- コラボレーションおよびライセンス：クラウドネイティブ CAD はチームワークフローを加速するが、ベンダーロックインは長期的な自動化パイプラインを妨げる可能性がある。

人間型設計選択への影響：

- コントローラサイジングおよびアクチュエータ選択のために、早段階で正確な慣性モデルを優先する。
- 制御および強化学習用に、簡略化されたメッシュおよび検証済み慣性値を持つ「シミュレーション対応」ブランチを維持する。
- 統合障害を回避するため、単位、基準フレーム、密度仮定を CAD モデルメタデータに文書化する。

7.2 Omniverse を機械シミュレーションに使用する

前の小節では、ロボット構造のための CAD アセンブリを準備・簡略化し、部品階層を保持したままジオメトリをエクスポートする方法を説明した。以下では、それらのエクスポート済み USD アセンブリを Omniverse に取り込み、物理的に正確な機械シミュレーションを実行し、人型サブシステムの設計イテレーションにフィードバックする方法を示す。

Omniverse を機械シミュレーションに使用することは、ハードウェア製作前に人型サブアセンブリがトルク・強度・接触要件を満たすことを検証するという一般的な工学課題に対処する。ワークフローは、CAD ジオメトリをシミュレーション可能な USD シーンに変換し、質量と慣性を割り当て、関節と接触を設定し、再現性のある動的テストを実行する。主な目的は、期待ペイロード下での関節駆

動の検証、接地時の接触力評価、高加速度動作中のピークアクチュエータトルクの推定である。

問題設定とワークフロー概要：

- ・問題：脚または腕の設計が指定動的負荷に耐えつつアクチュエータ制限を満たすことを保証する。
- ・入力：簡略化済み CAD ジオメトリ、公称質量、関節軸定義、意図した運動プロファイル。
- ・出力：関節トルク時系列、接触力・インパルス、重心軌跡、失敗フラグ（トルクまたは応力制限超過）。

技術解析は、Omniverse が NVIDIA PhysX アーティキュレーションを用いてシミュレートする関節剛体動力学から始まる。 n 自由度関節サブシステムの運動方程式は、トルクバランスを

$$[H]M(q)\ddot{q} + C(q, \dot{q}) + g(q) = \tau + J(q)^T f_{\text{ext}}, \quad (56)$$

と表し、ここで M は質量行列、 C はコリオリ・遠心力項、 g は重力、 τ はアクチュエータトルク、 $J^T f_{\text{ext}}$ は外部接触力の写像である。式 (1) は逆動力学チェックと、シミュレートされたトルクをアクチュエータ連続・ピーク定格と比較するための基礎となる。

エンジニアが正確に制御すべき実装詳細：

1. ジオメトリ準備

- ・可能な限り凸型衝突プリミティブを使用。凸包は衝突検出を高速化し接触ジッタを低減する。
- ・レンダリングメッシュとは別に低詳細衝突メッシュを作成。脚・手の衝突頂点数は少なく保つ。
- ・CAD と USD エクスポート間で関節フレームを一致させる。運動モデルで使用したものと同じ局所軸を用いる。

2. 質量・慣性の割り当て

- ・CAD 由来の質量特性を優先。利用不可の場合は材料密度と部品体積から算出。
- ・細長リンクについては、慣性テンソルを解析的または CAD エクスポートで算出。Omniverse はリンク局所フレームでの明示慣性を受け入れる。

3. アーティキュレーションと関節駆動チューニング

- ・PhysX アーティキュレーション駆動を用いて関節コンプライアンス・減衰・最大トルクをモデル化。
- ・駆動剛性・減衰をアクチュエータ帯域幅に一致させ数値不安定性を誘発しないよう調整。

4. 接触パラメータ

- ・摩擦係数、反発係数、接触厚みを設定。人型足地相互作用には、実係数（例：ソール材質に応じ 0.4–1.0）を用いたクーロン摩擦モデルを使用。

5. ソルバとタイムステップ

- ・高加速度テストではソルバサブステップと反復回数を増加。推奨初期値：タイムステップ $dt = 0.002\text{s}$ 、ソルバ反復 4–8 回。厳密制御ループでは $dt \leq 0.001\text{s}$ が必要な場合がある。
- ・精度と CPU/GPU コストをバランス：高忠実度は実時間を増やしリアルタイムテストを制限する。

Omniverse で実行すべき実用的テスト：

- 公称姿勢での静的バランスと重心投影。
- 式 (1) の逆動力学を用いた所定歩行相での動的関節トルク推定。
- インパクトテスト：落下接触によるピーク接触インパルスとアクチュエータトルクスパイク測定。
- 摩擦・接触剛性に対する感度スイープでロバスト性を把握。

コンパクトな Omniverse/Isaac Sim Python 例は、USD 人型をロードし、物理を設定し、関節軌道を指令し、報告アクチュエータトルクと接触力をログする方法を示す。/World/humanoid を自身のシーンパスに置き換える。

コードサンプル 24 Isaac Sim スクリプト：USD ロード、物理 dt 設定、関節指令、トルクログ

```
# Isaac Sim 用の高品質サンプル (ROS 2 非使用)
import os
import signal
import sys
from pathlib import Path
from typing import Dict, List, Optional

import numpy as np
from omni.isaac.core import World
from omni.isaac.core.articulations import Articulation
from omni.isaac.core.utils.extensions import enable_extension
from omni.isaac.kit import SimulationApp

# 必要な拡張を有効化
enable_extension("omni.isaac.core")

# シグナルハンドラ：Ctrl-C でクリーン終了
def _sigint_handler(sig, frame):
    print("\n[INFO] 中断を検出。シミュレータを終了します。")
    SimulationApp.get_app().close()
    sys.exit(0)

signal.signal(signal.SIGINT, _sigint_handler)

# 起動オプション
CONFIG = {
    "headless": False,
    "width": 1280,
    "height": 720,
    "sync_loads": True,
```

```

        "renderer": "RayTracedLighting",
    }

# シミュレーション設定
SIM_DT = 1.0 / 500.0 # 500 Hz
SOLVER_ITER = 8
USD_PATH = Path("/path/to/humanoid.usd") # 実ファイルを指定
ROBOT_PRIM = "/World/humanoid"
HIP_JOINT = "hip_joint"
FOOT_LINK = "right_foot"

# 軌道設定
AMP = 0.3 # rad
FREQ = 0.5 # Hz
DURATION = 2.0 # sec

def load_robot(world: World, usd_path: Path, prim_path: str) -> Articulation:
    """USDをステージに追加してArticulationオブジェクトを返す"""
    if not usd_path.exists():
        raise FileNotFoundError(f"USD not found: {usd_path}")
    world.scene.add_reference_to_stage(str(usd_path), prim_path)
    world.reset() # 読み込み後に必ず reset
    return Articulation(prim_path)

def configure_articulation(art: Articulation) -> None:
    """関節ドライバ設定一括適用"""
    art.set_enabled_self_collisions(False)
    art.apply_drive_settings(
        {
            "leg.*": {"stiffness": 100.0, "damping": 10.0},
            "arm.*": {"stiffness": 80.0, "damping": 8.0},
        }
    )

def main() -> None:
    kit = SimulationApp(CONFIG)
    world = World()

```

```

# 物理設定
world.physics_context.set_time_step(SIM_DT)
world.physics_context.get_physics_view().set_solver_iterations(SOLVER_ITER)

# ロボット読み込み
robot = load_robot(world, USD_PATH, ROBOT_PRIM)
configure_articulation(robot)

# ログ用バッファ
log: List[Dict[str, float]] = []

steps = int(DURATION / SIM_DT)
for i in range(steps):
    t = i * SIM_DT
    q_des = AMP * np.sin(2.0 * np.pi * FREQ * t)
    robot.set_joint_position(HIP_JOINT, q_des)

    world.step(render=True)

# 計測
tau = robot.get_joint_effort(HIP_JOINT) or 0.0
forces = robot.get_contact_forces(FOOT_LINK)
force_mag = np.linalg.norm(forces) if forces is not None else 0.0
log.append({"t": t, "tau": tau, "force": force_mag})

# 簡易ログ出力
for row in log[::50]: # 10 ms 間隔で出力
    print(f"{row['t']:.3f}, torque={row['tau']:.3f}, contact={row['force']:.3f}")

kit.close()

if __name__ == "__main__":
    main()

```

工学における影響とトレードオフ：

- 忠実度対計算：ソルバ忠実度を上げるとトルク予測精度が向上するが計算コストが増加。クリティカルフェーズのみ高忠実度実行を用いる。
- 質量・慣性誤差はトルク推定に直接伝播。アクチュエータの過小選定を避けるため早期に質量特

性を検証。

- ・衝突簡略化は計算を削減するが、非物理的接触点と誤ったモーメントを生じる場合がある。
- ・ソルバチューニングとタイムステップ選択は数値減衰に影響。過剰数値減衰は真の動的特性を隠蔽。
- ・シミュレーションから実機へのギャップ：センサノイズ、ギアバックラッシュ、アクチュエータ電流制限は信頼性の高い移行のため追加モデリングが必要。

設計文書に記載すべき運用上のリスクには、質量特性を検証せずにシミュレーションを過信、足地相互作用の接触モデリング複雑性を無視、アクチュエータ動特性をほぼ完璧と仮定することが含まれる。Omniverse 駆動シミュレーションを反復的に使用し、結果を CAD 調整にフィードバックし、アクチュエータ選定と構造安全のためのマージンを定量化する。

7.3 ヒューマノイドの実用的設計ヒント

Omniverse で検証された運動学と CAD からエクスポートされたジオメトリを基に、これらのヒントはシミュレーション結果と製造可能なハードウェアを橋渡しする。遠位慣性の最小化、アクチュエータを動的負荷にマッチングし、安全で保守可能な配線・センシングを確保することに焦点を当てる。

問題定義. ヒューマノイドの関節は要求トルクと帯域を達成しながら質量を最小化し安全性を維持しなければならない。設計者はタスクレベル要求（歩行速度、ペイロード処理、外乱抑制）をリンク慣性、アクチュエータ選定、機械インタフェースに変換しなければならない。

技術解析 — コア計算と経験則.

- ・質量 m 、重心距離 r の単一回転リンクに対する静的重力トルクを推定：

$$[H]\tau_g = mgr, \quad (57)$$

ただし $g \approx 9.81 \text{ m/s}^2$ 。

- ・平面リンクが角加速度 $\ddot{\theta}$ を受ける場合、回転慣性 I と並進効果を含む。簡潔な近似は

$$[H]\tau_{\text{req}} = I\ddot{\theta} + mr^2\ddot{\theta} + \tau_g. \quad (58)$$

一様な細長ロッドの長さ L に対しては $I = \frac{1}{3}mL^2$ を用いる。

- ・関節固有振動数を計算しアクチュエータ帯域をチェックする。アクチュエータ＋負荷を慣性 J_{eff} 、関節剛性 k としてモデル化し

$$[H]\omega_n = \sqrt{\frac{k}{J_{\text{eff}}}}. \quad (59)$$

制御可能な歩行と外乱抑制のため、アクチュエータ閉ループ帯域は概ね $0.2\text{--}0.5\omega_n$ を目指す。

設計ヒューリスティックと手順.

1. まず遠位質量を削減する。手先の 1 kg はリンク長を通じて必要肩トルクに乘算される。軽量エクソスケルトンスタイルのシェルと胴体近傍のローカル電子機器を優先する。
2. 分散センシングを用いる：IMU は胴体中央に、関節エンコーダは関節軸上に、ロードセルは手首または足部接触面に配置する。これにより状態推定誤差を削減しキャリブレーションを簡素化する。

3. ギア選定のトレードオフ：

- ・高減速（ハーモニック、サイクロイド）はトルク密度を上昇させるがバックドライバビリティを低下させ遊びヒステリシスを追加する。
- ・低減速またはダイレクトドライブはコンプライアンスと力制御を改善するが、より大型のモータと熱設計を要求する。

ギア比 N を選定し、モータトルク T_m が $NT_m \geq \tau_{\text{peak}}(1 + \epsilon)$ を満たし、マージン $\epsilon \in [0.2, 0.5]$ をデューティサイクルと安全性に応じて設定する。

4. コンタクトタスクのために意図的にコンプライアンスを追加する。Series Elastic Actuators (SEA) はモータ慣性をデカップリングし力制御を改善する。剛性 k を設計し、式 (59) を通じて所望の接触帯域を達成する。
5. 熱サイジング：連続モータトルクをミッション平均トルクと比較する。代表的デューティサイクルにわたる RMS トルクを推定しモータ熱限界にマッピングすることで検証する。
6. 配線と保守：ケーブルランを短く保ち、関節近くでストレインリリーフを用い、サービス可能なハーネスアクセスパネルを提供する。コネクタを標準化し交換を簡素化する。

実装例 — 要求トルクマージンと推定固有振動数の計算. コードは静・動トルクを計算しアクチュエータを選定し、マージンと ω_n を報告する。

コードサンプル 25 Simple torque and bandwidth check for a revolute joint.

```
#!/usr/bin/env python3
"""
単リンクアームの必要トルクと自然振動数を計算するプロダクションスクリプト
"""

from __future__ import annotations

import argparse
import json
import logging
import sys
from dataclasses import dataclass
from pathlib import Path
from typing import Final

import numpy as np
import yaml

# ロギング設定
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s",
```

```

        datefmt="%Y-%m-%d_%H:%M:%S",
    )
    logger = logging.getLogger(__name__)

@dataclass(frozen=True)
class LinkParams:
    """リンクパラメータを不変に保持"""
    mass: float          # kg
    com_distance: float  # m
    length: float        # m
    peak_accel: float    # rad/s^2

@dataclass(frozen=True)
class MotorParams:
    """モータ仕様"""
    nominal_torque: float # Nm (連続)
    peak_torque: float    # Nm (瞬間)
    gear_ratio: float

@dataclass(frozen=True)
class JointParams:
    """関節特性"""
    stiffness: float # Nm/rad

# 定数
GRAVITY: Final[float] = 9.81 # m/s^2

def compute_required_torque(link: LinkParams) -> float:
    """
    必要トルクを計算
    """
    # スランダロッド近似による慣性モーメント
    inertia = (1.0 / 3.0) * link.mass * link.length ** 2

    # 重力トルク

```

```

tau_gravity = link.mass * GRAVITY * link.com_distance

# 動的トルク（加速度項）
tau_dynamic = (inertia + link.mass * link.com_distance ** 2) * link.peak_accel

return tau_dynamic + tau_gravity

def compute_natural_frequency(link: LinkParams, joint: JointParams) -> float:
    """
    関節の自然振動数を計算
    """
    inertia = (1.0 / 3.0) * link.mass * link.length ** 2
    j_eff = inertia + link.mass * link.com_distance ** 2
    return np.sqrt(joint.stiffness / j_eff)

def compute_margin(required: float, selected: float) -> float:
    """
    トルクマージンを計算
    """
    if required <= 0:
        raise ValueError("required_torque_must_be_positive")
    return (selected - required) / required

def load_config(path: Path) -> dict:
    """
    YAML/JSON設定ファイルを読み込む
    """
    with path.open("r", encoding="utf-8") as f:
        if path.suffix.lower() in {".yaml", ".yml"}:
            return yaml.safe_load(f)
        return json.load(f)

def main(argv: list[str] | None = None) -> None:
    parser = argparse.ArgumentParser(description="単リンクアームの必要トルク計算")
    parser.add_argument(
        "-c", "--config", type=Path, help="YAML/JSON設定ファイル"
    )

```

```

)
parser.add_argument(
    "-o", "--output", type=Path, help="結果をJSONで出力するファイル"
)
args = parser.parse_args(argv)

if args.config:
    cfg = load_config(args.config)
    link = LinkParams(**cfg["link"])
    motor = MotorParams(**cfg["motor"])
    joint = JointParams(**cfg["joint"])
else:
    # デフォルト値
    link = LinkParams(mass=2.5, com_distance=0.25, length=0.5, peak_accel=10.0)
    motor = MotorParams(nominal_torque=1.2, peak_torque=3.6, gear_ratio=50.0)
    joint = JointParams(stiffness=500.0)

tau_req = compute_required_torque(link)
omega_n = compute_natural_frequency(link, joint)
tau_selected = motor.peak_torque * motor.gear_ratio
margin = compute_margin(tau_req, tau_selected)

result = {
    "required_torque_Nm": round(tau_req, 2),
    "selected_torque_Nm": round(tau_selected, 1),
    "torque_margin": round(margin, 2),
    "natural_frequency_rad/s": round(omega_n, 2),
}

logger.info("tau_req=%2fNm, tau_selected=%1fNm, margin=%2f", tau_req, tau_sel
logger.info("natural_freq=%2frad/s", omega_n)

if args.output:
    with args.output.open("w", encoding="utf-8") as f:
        json.dump(result, f, ensure_ascii=False, indent=2)
    logger.info("結果を%sに保存しました", args.output)

if __name__ == "__main__":
    main()

```

CAD からプロトタイプへ移行する際の実用的チェックリスト：

- ・質量特性を検証：CAD 質量・慣性をエクスポートし手計算と比較する。
- ・シミュレーションで生成されたモーションプロファイルから最悪ケーストルクシナリオを実行する。
- ・期待されるミッションプロファイルを用いてモータの熱デューティサイクルをシミュレートする。
- ・モジュラ関節でプロトタイプを作成し、迅速な交換と段階的改善を可能にする。

エンジニアリングにおける影響とトレードオフ。

- ・遠位質量削減はエネルギー効率と制御帯域を上昇させるが、軽量材料によりコストが増加する可能性がある。
- ・高減速はモータ選定を簡素化するが、バックドライバビリティを低下させ人間安全インタラクションのリスクを増大させる。
- ・コンプライアンスはコンタクト安全性を改善するが、位置精度を低下させ高速歩行制御を複雑化する可能性がある。
- ・トルクまたは熱限界を過小評価するとアクチュエータ飽和、過熱、予期せぬ故障が生じる。

試験中に監視すべき運用上のリスク：

- ・繰返し衝撃に対してライフサイクル定格されていない場合の衝撃負荷下でのギアボックス故障。
- ・連続 RMS トルクがモータ熱限界に接近する際の熱暴走。
- ・フレキシブルケーブリングまたは熱膨張によるセンサドリフト；配線とキャリブレーション手順で軽減する。

設計トレードは初期のハードウェアインザループ試験後に常に再検討すべきである。シミュレーション最適選択は、組立公差、配線ルーティング、未モデル化摩擦を考慮すると調整を要することが多い。

7.4 ケーススタディ：ロボットハンドの構築

前節で議論した実践的な設計ヒューリスティクスと Omniverse 検証ワークフローを基に、本ケーススタディではそれらのツールを用いて人間型マニピュレータに適した五指ロボットハンドを設計・解析・試作する。力、巧緻性、センシング、およびシミュレータ駆動の反復設計との統合をバランスさせるエンジニアリング上の選択に重点を置く。

問題定義と動作制約。家庭内マニピュレーションと軽作業ピック・アンド・プレースを任される人間型のための手首取付け型ハンドを設計せよ。主要要求事項：

- ・指尖把持力：ピンチおよびパワーグリップで各指 10 N 持続。
- ・指速度：完全屈曲 0.25 s。
- ・重量目標：ハンド全体 1.2 kg。
- ・センシング：関節エンコーダ、指尖タクティルアレイ、手のひら 1 軸力・トルクセンサ。
- ・安全性：人との剛性衝突を防ぐ受動コンプライアンス。

技術解析：運動学および駆動トポロジー．ハイブリッドアーキテクチャを選択：近位外転／内転は直駆動，遠位屈曲は腱駆動．これにより遠位モータ質量を削減し，アンダーアクチュエート適応把持を可能にする．2 関節屈曲を持つ各平面指について，指尖力 F_{tip} はヤコビアン転置を介して関節トルクに写像される．関節角 θ_1, θ_2 ，リンク長 l_1, l_2 に対し，平面ヤコビアン $J(\theta)$ は

$$[H]J(\theta) = \begin{bmatrix} -l_1 \sin \theta_1 - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos \theta_1 + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \end{bmatrix}. \quad (60)$$

純粋な法線指尖力 $\mathbf{F} = [F_x, F_y]^T$ 下では，静平衡より関節トルク

$$[H]\tau = J(\theta)^\top \mathbf{F}. \quad (61)$$

指尖法線に沿った単軸法線力では，これは各関節のスカルトルク要求に簡約化される．この写像を用いてモータおよびギアをサイジングする．

駆動サイジングと伝達トレードオフ．モータトルクは関節トルクと伝達損失を克服しなければならない．要求関節トルク τ_{req} ，減速比 N ，ギアボックス効率 η に対し，モータトルクは

$$[H]\tau_{\text{motor}} \geq \frac{\tau_{\text{req}}}{N\eta}. \quad (62)$$

バックドライバビリティを維持しながら，速度とトルクをトレードするよう N を選ぶ．腱駆動アンダーアクチュエート指では，腱力 F_t とプーリ半径 r を定義し，各巻き付き関節に対し関節トルク $\tau_j = r_j F_t$ が得られる．アンダーアクチュエーションはルーティングジオメトリを介して関節トルクを結合させ；その結合を写像行列 A で表現し $\tau = AF_t$ とする．

設計アルゴリズムと検証計画：

1. 人間工学ターゲットから指ジオメトリおよびリンク長を定義．
2. 臨界把持姿勢で (60)–(61) を用いて最悪ケーストルクを計算．
3. ピークトルクおよび連続トルクを熱限界と照合してモータファミリを選定．
4. 伝達を選択：近位関節は低バックラッシュハーモニックドライブ，遠位関節はコンプライアンスのためキャプスタンまたはシース腱ルーティング．
5. 指尖にタクトイルセンサおよびフレックスセンサを統合し，シミュレーションで検証．
6. Omniverse で反復：CAD をインポート，関節定義をアタッチ，腱ルーティングおよび接触シーケンスをシミュレート．

実装スニペット．以下の Python スニペットは法線指尖力下の 2 リンク指に対して要求モータトルクを計算する．モータ選定およびギアボックス選択をパラメータ化するために用いよ．

コードサンプル 26 Compute motor torque for a 2-link finger; use parameters to size motors.

```
import math
from typing import Tuple, List

# パラメータをクラスで管理（保守性向上）
class FingerParams:
    l1: float = 0.05          # リンク1長さ [m]
```

```

l2: float = 0.03          # リンク2長さ[m]
gear_ratio: float = 50.0  # 減速比
efficiency: float = 0.85  # 伝達効率

def compute_jacobian(theta1: float, theta2: float, params: FingerParams) -> List[List[
    """ヤコビ行列を計算"""
    s1 = math.sin(theta1)
    s12 = math.sin(theta1 + theta2)
    c1 = math.cos(theta1)
    c12 = math.cos(theta1 + theta2)

    J = [
        [-params.l1 * s1 - params.l2 * s12, -params.l2 * s12],
        [ params.l1 * c1 + params.l2 * c12,  params.l2 * c12]
    ]
    return J

def compute_motor_torques(
    theta1: float,
    theta2: float,
    F_tip: float,
    params: FingerParams
) -> Tuple[float, float, float, float]:
    """
    関節トルクとモータ必要トルクを返す
    F_tip: 指先に垂直に作用する力[N]
    """
    J = compute_jacobian(theta1, theta2, params)
    Fx, Fy = 0.0, F_tip

    # tau = J^T * F
    tau1 = J[0][0] * Fx + J[1][0] * Fy
    tau2 = J[0][1] * Fx + J[1][1] * Fy

    # 減速機・効率を考慮
    motor_tau1 = abs(tau1) / (params.gear_ratio * params.efficiency)
    motor_tau2 = abs(tau2) / (params.gear_ratio * params.efficiency)

    return tau1, tau2, motor_tau1, motor_tau2

```

```

def main() -> None:
    params = FingerParams()
    theta1 = math.radians(30)
    theta2 = math.radians(20)
    F_tip = 10.0

    tau1, tau2, mt1, mt2 = compute_motor_torques(theta1, theta2, F_tip, params)

    print(f"{tau1=:.3f}, {tau2=:.3f}")    # 関節トルク [Nm]
    print(f"{mt1=:.3f}, {mt2=:.3f}")      # モータ必要トルク [Nm]

if __name__ == "__main__":
    main()

```

材料および指尖設計. 近位リンクには高強度低密度合金を, 遠位リンクおよびパッドにはフレキシブルポリマーを用いる. 指尖タクティルアレイは静電容量型または圧抵抗型センシングと薄いエラストマ皮膚を組み合わせたべきである. 皮膚厚さは接触コンプライアンスを制御し, 力伝達およびセンサ感度に影響する.

制御およびセンシング統合. 階層コントローラを実装:

- 低レベル: モータ毎のトルクまたは電流制御で, 関節レベル位置フォールバックを持つ.
- 中レベル: 把持合成が要求指尖力を計算し, (61) の逆写像またはアンダーアクチュエート最適化を介してアクチュエータ指令に写像する.
- 高レベル: 知覚駆動把持プランナが視覚およびタクティルフィードバックから把持点を供給する.

エンジニアリング上の影響, トレードオフ, およびリスク:

- アンダーアクチュエーションはアクチュエータおよび質量を削減するが, 独立関節制御を制限する. 把持適応を簡素化する.
- 高減速比はトルクを増やすが, バックドライバビリティおよびタクティル応答性を低下させる. ギアボックス選択は人間接触時の安全性に影響する.
- 腱ルーティングは遠位質量を削減するが, 摩擦およびヒステリシスを導入する. キャプスタンまたは低摩擦シース設計が損失を緩和する.
- Omniverse でのシミュレーションは腱干渉および衝突を早期に検出するが, 実機摩擦および皮膚コンプライアンスは実験的チューニングを要する.
- センサ遅延およびアクチュエータ熱限界は持続把持タスクを制約する. 熱放散を設計し, モータ焼損を回避する電流限界を含める.

設計者は巧緻性, 重量, 頑健性をバランスさせ人間型ミッションプロファイルを満たさなければならぬ. モータ熱サイクル, 腱寿命, 皮膚摩擦を実験室試験で検証し, 配備前に妥当性を確認せよ.

8 材料と製造

8.1 ロボティクス用構造材料

先に議論した機械的解剖学およびアクチュエータの決定を基に、材料選択は質量分布、共振挙動、およびインターフェース信頼性を決定する構造基準を固定する。以下では、ヒューマノイドロボットにおける構造材料選択のための基準、計算、および実践的な製作指針を展開する。

材料選択は工学最適化である：歩行、操縦、コンプライアントな相互作用といったタスクに対して、剛性、強度、疲労寿命、および製造可能性の制約を満たしながら質量を最小化する。主要な性能指標は以下を含む：

- ・密度 ρ (kg m^{-3})、これは直接慣性負荷に影響する；
- ・ヤング率 E (GPa)、剛性および固有振動数を制御する；
- ・降伏強度 σ_y (MPa) および靱性、永久荷重および衝撃耐性のため；
- ・比強度 σ_y/ρ および比剛性 E/ρ 、質量制限設計内で有効；
- ・疲労特性（耐久限度または S-N 曲線）、周期的アクチュエータ荷重のため；
- ・環境耐久性（腐食、UV、温度）、および修理可能性。

設計の初期段階で解析チェックを用いて実現可能な材料を範囲指定する。脚や胴体ストラットの細長い部材では、オイラー座屈が圧縮限界を設定する：

$$[H]P_{\text{cr}} = \frac{\pi^2 EI}{(KL)^2}, \quad (63)$$

ここで I は断面二次モーメント、 L は非支持長、 K は有効柱長係数である。 E が高い材料を選べば P_{cr} が増加するが、質量は ρ とともに増加するため、軽量圧縮部材の候補を比較するには E/ρ を評価する。

振動性能については、構造肢のモード周波数を

$$[H]f_n \approx \frac{1}{2\pi} \sqrt{\frac{k}{m}}, \quad (64)$$

で近似する。ここで k は梁に対して EI/L^3 に比例する。 E または断面慣性 I を増やすと k が上昇するが、断面がスケールすれば質量 m も増加する。脚および胴体の第 1 モードの設計目標は、アクチュエータ帯域幅および制御ループ周波数を安全率、典型的には 3-5 倍で上回るべきである。

ヒューマノイド向けの典型的なトレードオフを持つ材料クラス：

- ・鋼（低合金、ステンレス）：高靱性、予測可能な疲労；高密度は局所強度および衝撃耐性が支配的な場所（ジョイントハウジング、マウントプレート）に適する。ねじ切りファスナおよび溶接組立体に適す。
- ・アルミニウム合金（例：6061、7075）：中強度、低密度、加工および押出しが容易。制御に慣性低減が利点となるアームリンクおよび胴体フレームに一般的。
- ・チタン合金（Ti-6Al-4V）：優れた比強度および耐食性。ヒップピンや構造インサートなど高荷重・低質量用途に使用されるが、コストおよび加工困難性が高い。

- 炭素繊維強化ポリマー（CFRP）：繊維方向に傑出した比剛性および比強度、非常に低密度。荷重方向が制御される長尺リンクおよびシェル構造に最適。切り欠き感受性、厚さ方向強度低、複雑な製造（レイアップ、オートクレーブ）に注意。
- マグネシウムおよび高性能ポリマー（PEEK、強化ナイロン）：超軽量ハウジングおよびカバー用；クリープ、モータからの熱効果、および可燃性を確認。
- サンドイッチパネルおよびハニカム：胴体シェルおよび大型パネル向けに質量あたりの高い曲げ剛性；コア-フェース接合およびエッジ補強に注意が必要。

インターフェース設計および製造制約：

- ボルト穴および軸受座は応力を集中させる。局所補強およびねじ切り取り付けには金属が好ましい；複合材では共硬化金属インサートまたは接着ダウエルを用いる。
- 溶接対接着対機械的締結：溶接は鋼および一部のアルミに適すが残留応力を生じる。接着剤は複合材および異種材料接合での荷重伝達を可能にし、疲労分布が良好だが表面処理およびプロセス制御を要する。
- 積層造形はトポロジー最適化および統合機能（導管、ハーネスチャネル）を可能にする。金属 SLM を複雑なチタンまたは鋼ブラケットに用いるが、異方性特性および後処理熱処理を設計に組み込む。
- 共硬化およびオーバーモールドリングは、複合材スキンとポリマーコアを組み合わせ、重い金属補強材なしに衝撃耐性を向上させることができる。

実践的サイジング例：長さ L の軸方向圧縮荷重 P を受ける直線脚ストラットの 2 候補材料を比較する。式 (63) から必要断面慣性を計算し、次に断面積および質量を評価する。二次チェック：座屈安全率、式 (411) による第 1 モード周波数、およびジョイント荷重下での局所ベアリング応力。

以下の Python スニペットは、比剛性および比強度によって候補材料をランク付けし、初期スクリーニング指標を提供する。最終設計では数値特性を $\square\square$ またはデータシート値に置き換える。

コードサンプル 27 比剛性および比強度による材料ランキング

```
#!/usr/bin/env python3
"""
比剛性・比強度でヒューマノイド構造材料をランク付けする CLI ツール
"""

from __future__ import annotations

import argparse
import json
import sys
from dataclasses import dataclass
from pathlib import Path
from typing import Dict, List, Tuple
```

```

# 材料データ型定義
@dataclass(frozen=True)
class Material:
    name: str
    density: float          # kg/m^3
    young_modulus: float    # GPa
    yield_strength: float   # MPa

    @property
    def specific_stiffness(self) -> float:
        """比剛性  $E/\rho$  [m^2/s^2]"""
        return self.young_modulus * 1e9 / self.density

    @property
    def specific_strength(self) -> float:
        """比強度  $\sigma_y/\rho$  [m^2/s^2]"""
        return self.yield_strength * 1e6 / self.density

# デフォルト材料データベース
DEFAULT_MATERIALS: Tuple[Material, ...] = (
    Material("Steel_1045", 7850, 210.0, 530.0),
    Material("Al_7075", 2810, 71.7, 503.0),
    Material("Ti6Al4V", 4430, 114.0, 880.0),
    Material("CFRP", 1600, 120.0, 900.0), # 異方性近似
)

def load_materials(path: Path | None = None) -> Dict[str, Material]:
    """JSONファイルから材料リストを読み込む（未指定時はデフォルトを返す）"""
    if path is None:
        return {m.name: m for m in DEFAULT_MATERIALS}

    with path.open(encoding="utf-8") as f:
        data = json.load(f)
    return {
        name: Material(name, d["density"], d["young_modulus"], d["yield_strength"])
        for name, d in data.items()
    }

```

```

def rank_materials(
    materials: Dict[str, Material],
    key: str = "stiffness",
    top_k: int | None = None,
) -> List[Tuple[str, Material, float, float]]:
    """指定キーで降順ソートし、top_k件返す"""
    key_func = (
        (lambda m: m.specific_stiffness) if key == "stiffness"
        else (lambda m: m.specific_strength)
    )
    ranked = sorted(
        materials.items(),
        key=lambda kv: key_func(kv[1]),
        reverse=True,
    )
    if top_k is not None:
        ranked = ranked[:top_k]
    return [(name, m, m.specific_stiffness, m.specific_strength) for name, m in ranked]

def main(argv: List[str] | None = None) -> None:
    parser = argparse.ArgumentParser(description="構造材料の比剛性・比強度ランキング")
    parser.add_argument("-f", "--file", type=Path, help="材料定義JSONパス")
    parser.add_argument("-k", "--key", choices=["stiffness", "strength"], default="stiffness",
                        help="ランキング指標")
    parser.add_argument("-n", "--top", type=int, help="上位n件を表示")
    parser.add_argument("--json", action="store_true", help="結果をJSON形式で出力")
    args = parser.parse_args(argv)

    materials = load_materials(args.file)
    ranked = rank_materials(materials, key=args.key, top_k=args.top)

    if args.json:
        print(json.dumps([
            {
                "name": name,
                "specific_stiffness": s_stiff,
                "specific_strength": s_str,
            }
        ]))

```

```

        for name, _, s_stiff, s_str in ranked
    ], ensure_ascii=False, indent=2))
else:
    for name, _, s_stiff, s_str in ranked:
        print(f"{name}:  specific_stiff={s_stiff:.2e},  specific_strength={s_str:.2e}")

if __name__ == "__main__":
    main()

```

設計への影響、トレードオフ、および運用上のリスク：

- ・集中荷重経路およびねじ切りインターフェースには金属を用いる；長尺部材および繊維方向を荷重に合わせられる反復荷重部材には複合材を用いる。
- ・CFRP のような高比剛性材料への過度の依存は転倒生存性を低下させる；犠牲要素またはエネルギー吸収構造と組み合わせる。
- ・材料とアクチュエータ間の熱不整合は緩みまたは反りを生じる；接着継手では線膨張係数を合わせる。
- ・製造方法は実現可能な幾何学および公差を決定する；厳密なジョイントフィットおよび軸受面はしばしば後加工を要する。
- ・疲労寿命および検査アクセスは脚およびヒップ組立体に計画する必要がある；アクセス不能な複合材積層板は内部層間剥離を隠蔽できる。

E/ρ および σ_y/ρ を用いた慎重かつ定量的な比較を、座屈およびモードチェックと組み合わせることで実現可能な材料を絞り込む。最終選択には、製造方法、インターフェース設計、および代表的なヒューマノイド運転サイクル下での実証疲労試験を組み込むべきである。

8.2 3D プリントと CNC 加工

構造材料の選択は実現可能な製造方法を直接的に制約するため、前述の材料選定の判断は積層または切削プロセスの適切性を左右する。ここでは、3D プリントと CNC 加工が一般的なヒューマノイド部品にどのように対応するかを分析し、実用的なサイジング、プロセス、実装ルールを提供する。

問題と運用上の関連性。ヒューマノイドロボットは、剛性、軽量、精度、再現性のある表面品質をバランスさせた部品を必要とする。典型的な部品には以下が含まれる：

- ・複雑な幾何学的形状と低コストを必要とする外装カバーやケーブルチャンネル。
- ・高剛性と正確な穴位置を要求するリンケージブラケットやモータマウント。
- ・厳格な公差と適切な表面仕上げを必要とするジョイントハウジングやベアリングシート。

製造方法の選択は、組立てフィット、動的挙動、稼働中の破損モードに影響する。

技術分析：方法の能力と限界。

- ・積層造形（FDM）および槽型光造形（SLA / DLP）
 - 強み：迅速なイテレーション、複雑な内部チャンネル、低セットアップコスト。

- 限界：層間接着による異方性機械特性、一般的な熱可塑性樹脂の低ガラス転移温度、粗い公差（FDM で典型的には $\pm 0.2\text{--}0.5\text{ mm}$ ）。
- 設計上の注意：主要な負荷経路をフィラメントまたは樹脂層方向に整えるよう部品を配置し、引張強度を最大化する。結晶性ポリマーの場合はアニーリング、樹脂の場合は UV 後硬化による後処理で剛性を回復させる。
- 選択的レーザー焼結（SLS）
 - 強み：等方的なナイロン部品、機能的プロトタイプおよび中量生産に適している。
 - 限界：液密性のために含浸が必要な多孔質表面；CAD で補償する必要のある寸法収縮。
- CNC 加工（フライス、旋盤、マルチ軸）
 - 強み：高剛性材料（アルミニウム、鋼、チタン）、厳格な公差（ $\pm 0.01\text{--}0.05\text{ mm}$ が達成可能）、優れた表面仕上げおよびベアリングフィット。
 - 限界：高い個別コスト、工具アクセスおよび固定具による幾何学的制限、複雑形状では長い CAM セットアップ。
- ハイブリッドワークフロー
 - プリントした複雑な形状と CNC 加工したクリティカルな特徴を組み合わせる。例：軽量トルソシェルをプリントし、公差を持たせてベアリングボアを加工する、またはねじ込み金属カラー挿入。

サイジングと簡易機械的チェック。片持ち梁として機能する外部ブラケットでは、設計者はたわみおよび応力限界を確保する必要がある。幅 b 、厚さ t の矩形断面で、長さ L の片持ちに端部力 F を受ける場合、先端の弾性たわみは

$$[H]\delta = \frac{FL^3}{3EI} \quad \text{ただし} \quad I = \frac{bt^3}{12}, \quad (65)$$

たわみ限界 δ_{\max} を満たすために必要な最小厚さは

$$[H]t = \left(\frac{4FL^3}{Eb\delta_{\max}} \right)^{1/3}. \quad (66)$$

製造方向に応じた有効ヤング率 E を用いる。層に対して垂直に負荷がかかる FDM 部品では、保守的に $E_{\text{eff}} \approx 0.6 E_{\text{bulk}}$ を用い、テストで検証するまでこれを維持する。

公差・フィット・運動精度。ベアリングおよびシャフトのフィットはジョイントバックラッシュと制御帯域を決定する。CNC 加工ハブには絶対数値の代わりに公差クラスを指定する；例えば、ジョイントタイプに応じて圧入または遊び用の H7/g6 フィット。プリントベアリングシートではインサート設置を想定し、クリープおよび摩耗を回避するためにヘリコイルまたは金属ブッシュを検討する。

プロセスパラメータと指針。

- FDM 方向：最長負荷ベクトルをフィラメント経路に整える；支持材が負荷面を傷つけるのを避けるためオーバーハングを最小化。
- 層厚およびインフィル：表面仕上げと造形時間のバランスから層厚を選定；インフィルパターンはせん断剛性に影響する。構造表皮にはソリッド外周と、クリティカルブラケットでは $> 80\%$ インフィルを推奨。
- CNC 送りおよび回転数：工具メーカーの表に従う。アルミニウムでは、6,000–10,000 RPM の開始主軸回転数と適切な超硬工具が一般的。外装の表面仕上げ向上には climb milling を用いる。

- 表面および熱的安定性：プラスチックは連続負荷でクリープする；最大連続使用温度を指定し、長時間負荷には金属補強または複合積層を検討。

代表的なヒューマノイド肩リンクの実装チェックリスト：

1. 負荷評価：ピークトルク、連続トルク、転倒時の横荷重。
2. (2) 式を用いて候補厚さを算出。
3. 方法選択：
 - $t > 5 \text{ mm}$ かつ高精度が必要なら CNC アルミニウムを優先。
 - 複雑な内部チャネルまたは迅速なイテレーションが必要なら、SLS または FDM に金属インサートを用いる。
4. 後処理定義：アニーリング（ポリマー）、溶剤平滑（SLA）、または表面処理（アルミニウムの硬質アルマイト）。
5. プロトタイプを作製し、計測し、動的負荷下で検証。

実用的なコードユーティリティ。以下の Python ヘルパーは最小厚さを算出し、たわみおよび要求公差に基づいて製造ルートを提案する。

コードサンプル 28 最小厚さ算出および製造提案

```
"""
厚さ推薦モジュール
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Final

# 材料定数（必要に応じて外部化）
E_ALUMINUM: Final[float] = 69e9          # [Pa]
E_NYLON: Final[float] = 3e9              # [Pa]
E_SLA_RESIN: Final[float] = 2.5e9        # [Pa]

# 製造方法判定閾値（単位：m）
THRESH_CNC: Final[float] = 5e-3          # 5 mm
TOL_CNC: Final[float] = 0.05e-3          # 0.05 mm
TOL_SLS_FDM: Final[float] = 0.2e-3       # 0.2 mm

@dataclass(frozen=True)
class Recommendation:
    """計算結果と製造方法を保持"""
```

```

thickness_m: float
method: str

def recommend_thickness(
    force_N: float,
    length_m: float,
    width_m: float,
    young_mod_Pa: float,
    delta_max_m: float,
    tolerance_m: float,
) -> Recommendation:
    """
    ビームのたわみ制約を満たす最小板厚と製造方法を返す。
    たわみ公式:  $\delta = (F L^3) / (3 E I)$ 、 $I = b t^3 / 12$ を仮定。
    """
    if any(v <= 0 for v in (force_N, length_m, width_m, young_mod_Pa, delta_max_m)):
        raise ValueError("全ての入力は正でなければならない")

    # 最小断面二次モーメントを逆算
    i_min = (force_N * length_m ** 3) / (3 * young_mod_Pa * delta_max_m)
    # 必要板厚
    t_min = ((12 * i_min) / width_m) ** (1.0 / 3.0)

    # 製造方法判定
    if t_min >= THRESH_CNC and tolerance_m <= TOL_CNC:
        method = "CNC_mill_aluminum"
    elif t_min < THRESH_CNC and tolerance_m <= TOL_SLS_FDM:
        method = "SLS_or_FDM_with_metal_inserts"
    else:
        method = "SLA_with_postcure_and_machined_features"

    return Recommendation(thickness_m=t_min, method=method)

# 使用例
if __name__ == "__main__":
    result = recommend_thickness(
        force_N=50.0,
        length_m=0.08,

```

```

width_m=0.02,
young_mod_Pa=E_NYLON,
delta_max_m=1e-3,
tolerance_m=0.2e-3,
)
print(f"min_thickness={result.thickness_m:.4f}μm, method={result.method}")

```

設計上のトレードオフと運用上のリスク。

- ・積層法は重量と複雑さを削減するが、異方的破損と限界熱抵抗をもたらす。寸法不足のプリントジョイントは衝撃で層間剥離を起こす可能性がある。
- ・CNC 加工金属部品は再現性のある運動特性を提供するが、質量とコストを増加させる。重い四肢は制御帯域を劣化させ、アクチュエータ負荷を増大させる。
- ・ハイブリッド組立ては界面と電食の可能性を追加する；シーリングおよび絶縁を計画する。

エンジニアは最終組立体を動的テスト、計測疲労試験、および公差スタック解析で検証しなければならない。ジョイント精度および安全性がクリティカルな場合は製造適性を優先し、高ステークス・高負荷ジョイントには CNC 金属ソリューションを、複雑・低負荷構造および迅速なイテレーションには積層製造を活用する。

8.3 強度と柔軟性のバランス

先行するサブセクションでは、部品の幾何学的および異方性の制約を課す材料選択と製造オプションを確立した。本サブセクションでは、それらの制約を人間型ロボットにおける強度と柔軟性の適切な組み合わせを実現するためのトレードスペース解析と実用的設計ルールに適用する。

ロボティクスの問題定義と工学的重要性。人間型ロボットは、関節トルクと精密な運動学を支えるための剛性リンクと、衝撃耐性、エネルギー蓄積、安全な相互作用のためのコンプライアント要素を必要とする。設計目標は、ピーク負荷下で構造強度基準を満たしながら、頑健性、効率性、または安全性を向上させる制御された柔軟性を提供することである。主要な性能ドライバは、質量、剛性分布、疲労寿命、製造可能性である。以下の解析は、それらのドライバを実行可能な計算と設計パターンに整理する。

技術解析：剛性、強度、および安定性の制約。

- ・曲げ剛性は、横荷重下でのたわみを決定する。長さ L 、ヤング率 E 、断面二次モーメント I の片持ち梁セグメントに力 F が作用した場合の先端たわみは

$$[H]\delta = \frac{FL^3}{3EI}. \quad (67)$$

設計者は I と材料の E を選択し、 δ を制御およびセンサ許容限界内に収める。

- ・強度（許容応力）は断面寸法を設定する。曲げモーメント M を受ける断面の最大曲げ応力は $\sigma_b = Mc/I$ であり、ここで c は外層繊維距離である。必要な安全率を持ち $\sigma_b \leq \sigma_{\text{allow}}$ となるように断面形状を選ぶ。
- ・座屈は細長部材を制限する。オイラー座屈荷重は

$$[H]P_{\text{cr}} = \frac{\pi^2 EI}{(KL)^2}, \quad (68)$$

であり、有効長さ係数 K は端部条件によって決まる。動作中の圧縮荷重に対して P_{cr} を確認する。

- 比剛性 E/ρ と比強度 σ_{allow}/ρ は、剛性・重量および強度・重量のトレードオフを制御する。質量が重要な四肢では、 E/ρ と σ_{allow}/ρ を最大化する。

複合材料とセルラー構造。積層造形と複合材積層は、傾斜剛性と異方性挙動を可能にする。多孔質媒体で一般的なべき法則関係を用いて、設計されたセルラーインフィルの有効弾性率をモデル化する：

$$[H]E_{eff} \approx E_s \phi^n, \quad (69)$$

ここで E_s は固体材料の弾性率、 ϕ は相対密度（インフィル率）、指数 n はセル位相に依存する（通常 $1 \leq n \leq 3$ ）。(69) を用いて、格子セルサイズまたは造形壁厚を変化させることで局所コンプライアンスを調整する。繊維強化部品では、主応力方向に繊維配向を合わせ、必要な場所で曲げ剛性を最大化し、繊維に対して横方向のコンプライアンスを許容する。

設計戦略と実装手順。

1. 動作プロファイルから機械的負荷ケースを定義する。最悪ケースの外乱（押す力、転倒）およびアクチュエータ誘起荷重を含める。
2. 許容たわみと固有振動数制約を設定する。四肢の固有振動数を、アクチュエータまたは環境によって励起される振動数よりも高く保ち、共振を回避する。
3. 比剛性と疲労特性によって候補材料を選定する。アルミニウム 7075、Ti-6Al-4V、炭素繊維積層板、エンジニアリング熱可塑性プラスチック（PA12、PETG、ナイロン）を比較する。
4. 構造を以下に分割する：
 - 一次負荷支持骨格（高 E/ρ 、方向性剛性）。
 - 調整可能なコンプライアントインターフェース（エラストマパッド、積層造形格子、フレクシャ関節）。
 - エネルギー蓄積要素（ねじりスプリング、シリーズ弾性アクチュエータ）。
5. 質量効率の良い負荷経路のためのトポロジー最適化を使用する。最適化された幾何学を CNC または積層造形用にエクスポートし、造形部品では局所インフィル密度を調整する。
6. 線形静解析およびモード FEA で検証し、次いで臨界接合部に対して繰返し疲労シミュレーションおよび物理的疲労試験を実施する。

実用的計算と簡易チェック。以下の Python スニペットは、曲げモーメントおよびオイラー座屈チェックを受ける四肢に必要な矩形梁厚を計算する。数値入力をロボット固有の荷重および長さ置き換える。

コードサンプル 29 曲げおよび座屈下の矩形リンクの簡易サイジングチェック

```
import numpy as np
from typing import Final, Tuple

# 材料・幾何パラメータ
M: Final[float] = 50.0          # N·m
```

```

L: Final[float] = 0.30          # m
b: Final[float] = 0.02          # m
E: Final[float] = 70e9          # Pa
rho: Final[float] = 2700        # kg/m^3
sigma_allow: Final[float] = 250e6 # Pa
K: Final[float] = 1.0           # オイラー係数
P: Final[float] = 200.0         # N

def required_thickness(moment: float, width: float, sigma_max: float) -> float:
    """曲げ応力が許容値以下となる最小板厚を返す"""
    return np.sqrt(6.0 * moment / (width * sigma_max))

def euler_critical_load(width: float, thickness: float, length: float,
                        young: float, k_factor: float) -> float:
    """矩形断面のオイラー座屈荷重"""
    I = width * thickness**3 / 12.0
    return np.pi**2 * young * I / (k_factor * length)**2

def mass_estimate(width: float, thickness: float, length: float, density: float) -> float:
    """見かけ質量（自重は無視）"""
    return width * thickness * length * density

def main() -> None:
    t_req = required_thickness(M, b, sigma_allow)
    Pcr = euler_critical_load(b, t_req, L, E, K)
    m = mass_estimate(b, t_req, L, rho)

    print(f"必要厚さ t={t_req*1000:.2f}mm")
    print(f"オイラー座屈 Pcr={Pcr:.1f}N, 安全率={Pcr/P:.1f}")
    print(f"推定質量 m={m*1000:.2f}g")

if __name__ == "__main__":
    main()

```

運用上重要な設計パターン。

- シリーズ弾性アクチュエータ（SEA）は、アクチュエータ出力インターフェースにコンプライアンスを配置する。SEA はピークトルク要求を低減し、衝撃耐性を向上させ、接触タスク中の制御を簡素化する。関節制御ループにスプリング剛性を考慮する。
- コンプライアントシェル：衝突を吸収するために、剛性内部フレームと柔らかい外層スキンを使用する。調整された減衰材料は反跳エネルギーを低減する。

- ・フレクシャ関節は、小範囲で繰返し可能なコンプライアント動作を提供する。バックラッシュを排除するが応力を集中させ、疲労安全な幾何学と適切な材料選択を必要とする。

試験、検証、および製造に関する注意。

- ・積層複合部品については、主配向に沿って印刷した試験片を試験し、 E_{eff} と疲労寿命を定量化する。異方性は軸外強度を 30–70
- ・CNC 加工金属リンクは等方性挙動を維持するが、第三者加工公差はフィットメントおよびフィレットでの応力集中を変える。疲労亀裂を回避するため最小半径を指定する。
- ・傾斜インフィルについては、スライサパラメータを制御し、再現性のある格子特徴を確保する。プリンタ分解能以下の小セルサイズは一貫性のない機械的特性を生む。

具体的な工学への影響、トレードオフ、および運用上のリスク。

- ・トレードオフ：
 - 質量対剛性：高剛性は通常質量を増加させる；高比剛性材料またはトポロジー最適化された幾何学を選択する。
 - 柔軟性対制御精度：付加コンプライアンスは高周波数帯域幅を低減し、コントローラ複雑性を増加させる。
 - 製造可能性対最適化忠実度：複雑なトポロジー最適化形状は機械加工が高コストとなることがあるが、積層造形では実現可能である。
- ・運用上のリスク：
 - 代表サイクル下で試験しないと、コンプライアント剛性インターフェースで疲労破壊が生じる。
 - 印刷または積層部品での予期しない異方性破壊が、転倒イベント中に脆性破壊を引き起こす。
 - 軽量コンプライアント構造での共振および低減衰が振動を引き起こす。

実用化人間型システムでは、反復試作、標的疲労試験、保守的安全率を優先する。

8.4 環境への配慮

前節で議論した強度と柔軟性のバランス、および 3D 造形と CNC 切削に固有の製造上のトレードオフは、環境選択に直接影響を与える。したがって、材料選定とプロセス計画は、人型ロボットに対して機械的性能、製造適性、およびライフサイクル環境影響をバランスさせる必要がある。

問題定義. 人型ロボットは多様な環境で動作し、長い設計寿命を有する。設計者は、機械的、電氣的、安全要件を満たしながら、ライフサイクル環境影響を最小化しなければならない。主要な環境目標は、エネルギー原単位の削減、温室効果ガス排出の低減、修理容易性の向上、および廃棄時リサイクルの実現である。これらの目標は機械設計における制約となる。

技術解析. 設計初期段階で技術者が適用可能な、定量化可能な環境指標を少数定義する：

- ・エネルギー原単位 E_{emb} (MJ kg^{-1}) および地球温暖化係数 GWP ($\text{kg CO}_2\text{-eq kg}^{-1}$).
- ・リサイクル可能率 R (0–1), 廃棄時に回収可能な材料流の割合.
- ・材料危険度指標 C_{crit} (無次元), 供給リスクおよび紛争鉱物を捉える.

- 毒性/アウトガスリスク T (定性的に数値スケールへ変換).

これらの指標を組み合わせて、最適化およびトレードオフ研究で使用可能な工学目的関数を作成する。1つの有用な正規化コスト関数は

$$[H]J = \alpha \frac{m}{m_{\text{ref}}} + \beta \frac{GWP \cdot m}{GWP_{\text{ref}} \cdot m_{\text{ref}}} - \gamma R + \delta C_{\text{crit}} + \eta T, \quad (70)$$

ここで、 m は部品質量、 $\alpha, \beta, \gamma, \delta, \eta$ はプログラムレベルの持続可能性目標から選ばれる重み係数である。正規化定数 m_{ref} および GWP_{ref} は部品間比較を可能にする。

機械的制約は残る：

- 強度： $\sigma_{\text{allow}}(\text{material}) \geq \sigma_{\text{required}}$.
- 疲労寿命： $N_f(\text{material}, S) \geq N_{\text{mission}}$.
- 剛性： EI (曲げ剛性) は動特性および制御帯域要件を満たす必要がある。
- 製造適性：選択プロセス（造形，切削，板金成形）で実現可能。

材料とプロセスは相互作用する：

- 積層造形（AM）は材料ロスを削減し，トポロジ最適化を可能にし， m を低減する。AM はしばしばポリマーまたは特殊金属粉末を使用する。サポート材料ロス，粉末処理のエネルギー強度，および後処理熱処理を考慮する。
- CNC 切削はチップを生成し，これはしばしばリサイクル可能である。しかし，AM が可能にする部品統合は，ファスナおよび接着剤を削減し， R を向上させる。
- 表面処理（アルマイト，メッキ，塗装）は耐食性を向上させるが，リサイクル性を低下させ，有害化学物質を追加する可能性がある。
- 接着剤およびオーバーモールドは分解を複雑化する。可能な限り，機械的ファスナまたはリバーシブルジョイントを優先する。

設計判断のための実用的指標は，機械性能あたりの比環境影響であり，例えば単位比剛性あたりの $\text{CO}_2\text{-eq}$ ：

$$[H]\Phi = \frac{GWP \cdot m}{E_{\text{bending}}} = \frac{GWP \cdot m}{EI/L}, \quad (71)$$

ここで， L は代表梁長である。低い Φ は，低い環境コストで剛性を提供する材料・プロセス組合せを示す。

実装. 概念設計中に以下の実用的ワークフローを使用する：

1. 機能要件を定義：荷重，疲労寿命，要求剛性，許容質量，電磁両立（EMC）レベル，および熱バジェット。
2. 材料・プロセスデータベースを照会し，既知の GWP ， E_{emb} ， R ， C_{crit} ，および機械的特性を持つ候補を抽出。
3. 各候補に対して (1) から J を計算し，強度，疲労，EMC，および製造適性の制約チェックを適用。
4. 候補をランク付け，トポロジ最適化を繰り返し，制約を保持しながら質量 m を削減する。

候補材料を計算・ランク付けするコンパクトスクリプトを以下に示す。CAD-PLM ワークフローへの統合に適した，簡単なスコアリング手法を示す。

コードサンプル 30 Compute environmental impact score for candidate materials

```
#!/usr/bin/env python3
import json
import logging
from dataclasses import dataclass, field, asdict
from pathlib import Path
from typing import List, Dict, Any

# ログ設定：本番ではファイルへ出力
logging.basicConfig(level=logging.INFO, format="%(asctime)s_%(levelname)s_%(message)s")
logger = logging.getLogger(__name__)

@dataclass(slots=True)
class Material:
    name: str
    mass: float
    GWP: float
    recyclability: float
    crit: float
    tox: float

    def to_dict(self) -> Dict[str, Any]:
        return asdict(self)

@dataclass(slots=True)
class Weighting:
    alpha: float = 1.0
    beta: float = 1.5
    gamma: float = 2.0
    delta: float = 1.0
    eta: float = 1.0

@dataclass(slots=True)
class Reference:
    mass: float = 1.0
    GWP: float = 1.0
```

```

class MaterialRanker:
    def __init__(
        self,
        weights: Weighting = Weighting(),
        reference: Reference = Reference(),
    ) -> None:
        self.weights = weights
        self.ref = reference

    def score(self, mat: Material) -> float:
        """単一材料の環境負荷スコアを算出（小さいほど優位）"""
        w = self.weights
        r = self.ref
        m, G, R, C, T = mat.mass, mat.GWP, mat.recyclability, mat.crit, mat.tox
        return (
            w.alpha * (m / r.mass)
            + w.beta * (G * m / (r.GWP * r.mass))
            - w.gamma * R
            + w.delta * C
            + w.eta * T
        )

    def rank(self, materials: List[Material]) -> List[Material]:
        """スコア昇順でソート"""
        return sorted(materials, key=self.score)

def load_materials(path: Path) -> List[Material]:
    """JSONファイルから材料リストを読み込む"""
    if not path.exists():
        logger.warning("材料DBが見つからないためサンプルデータを使用")
        return _sample_materials()

    with path.open(encoding="utf-8") as f:
        data = json.load(f)
    return [Material(**item) for item in data]

```

```

def _sample_materials() -> List[Material]:
    """サンプルデータ（本番では外部DBへ移行）"""
    return [
        Material("Al7075", 0.8, 8.1, 0.9, 0.2, 0.1),
        Material("CF_recycled", 0.5, 45.0, 0.4, 0.6, 0.2),
        Material("Stainless304", 1.2, 6.0, 0.85, 0.3, 0.15),
    ]

def main() -> None:
    db_path = Path("materials.json")
    materials = load_materials(db_path)

    ranker = MaterialRanker()
    ranked = ranker.rank(materials)

    for mat in ranked:
        logger.info(f"{mat.name}_score={ranker.score(mat):.3f}")

if __name__ == "__main__":
    main()

```

運用上の考慮事項およびリスク。バッテリーおよび電力電子機器は、人型ロボットの多くの環境リスクを支配する。リチウムイオンセルの熱暴走は、環境および安全上の高い影響を持つ。したがって、バッテリー化学および筐体材料に影響を与える設計選択は、ライフサイクル安全に過大な影響を与える。特定のポリマーからのアウトガスは、センサを劣化させ、または無菌環境を汚染する可能性がある。EMI シールドはしばしば銅またはニッケルメッキを使用する；これらの金属は C_{crit} を増加させ、リサイクルを複雑化する。

設計推奨事項およびトレードオフ：

- 分解を考慮した設計を優先し、 R を増加させる。リバーシブルファスナを使用し、異種材料の接着接合を最小化する。
- AM によるトポロジ最適化を使用して質量を削減するが、AM 後処理エネルギーおよび粉末リサイクル率を考慮する。
- 高リサイクル性合金（アルミニウム、鋼）を高質量構造部品に優先する。ただし、比剛性－質量要求が複合材料使用を必要とする場合を除く。
- 複合材料の場合、リサイクル繊維を評価し、廃棄時回収経路を確立する。
- 有害コーティングおよび PFAS 含有処理への依存を削減し、毒性スコアを制限する。
- 材料出自および認証を文書化し、RoHS および REACH のような規制リスクに対処する。

トレードオフは避けられない。 GWP を下げるとコストが増加したり、比剛性が低下したりする可

能性がある。質量を削減すると運用エネルギー効率は向上するが、特定のプロセスでは kg あたりの製造エネルギー強度が増加する可能性がある。これらのトレードオフは、(1) の目的関数および感度解析を使用して定量化する必要がある。

9 組立とテスト

9.1 機械部品の組み立て

この小節では、前述の CAD ベースの運動学的アライメントと材料選択を直接発展させ、それらの設計判断を具体的な組立作業に移行する。焦点は、精度・再現性・保守性がバランス・操作・衝撃荷重におけるロボット性能を決定するヒューマノイドサブシステムの信頼性ある組立にある。

問題定義：構造フレーム・関節・ギアボックス・四肢インターフェースを、期待荷重を維持し運動精度を保ち、校正済みセンサが有効なまま留まるよう組み立てる。主要な工学要件は：

- ・トルク伝達インターフェイスで微小スリップを回避するための制御されたクランプとプリロード；
- ・軸受寿命を維持するための回転関節におけるランナウトと偏心の最小化；
- ・エンドエフェクタ姿勢精度を保証するための文書化された公差スタックアップ。

技術解析は第一原理から出発し、現場で用いられる定量的チェックへと至る。

1. ファスナプリロードとトルク制御ボルト締結はせん断・軸荷重を伝える。適切なプリロードはクランプばらつきを減らし、繰返し荷重による疲労を防ぐ。一般的な技術者見積では、締付トルク T 、プリロード F_p 、公称ボルト径 d 、経験的摩擦係数 K が：

$$T \approx K F_p d. \quad (72)$$

潤滑・乾燥ねじの検証済み表から K 値を用いる。 F_p を計算後、軸応力 $\sigma_a = F_p/A_t$ をボルト降伏に対して照査する。継手面のせん断では、誘発せん断応力 $\tau = F_{\text{shear}}/A_s$ を材料せん断強度と比較する。トルク伝達に対する摩擦非スリップ要件を満たすようプリロードを確保：

$$F_p \geq \frac{M_r}{\mu r}, \quad (73)$$

ここで M_r は伝達モーメント、 μ は摩擦係数、 r は実効半径。

2. プレスフィット・軸受・インターフェランス計算インターフェランスフィットはハブをシャフトに固定し軸受位置を決める。シャフト径と材料ヤング率に基づくインターフェランス量を用いる。細長部材ではオイラー公式で座屈を検討：

$$P_{\text{cr}} = \frac{\pi^2 EI}{(KL)^2}, \quad (74)$$

E はヤング率、 I は断面二次モーメント、 L は非支持長、 K は実効長係数。軸受は組立時に制御された径ランナウトが必要である。プレス作業中はダイヤルゲージでシャフト同心度を測定し、図面に最大許容ランナウトを記載して軸受寿命を保持する。

3. 公差スタックアップと運動精度 CAD 運動モデルから導出されるエンドエフェクタ位置公差を満たすよう組立を行う。区間演算または RSS（平方和平方根）法で最悪ケーススタックアップを

計算：

- 最悪ケース線形：絶対公差の総和；
- 統計：独立公差 t_i に対し $RSS = \sqrt{\sum_i t_i^2}$ 。

生産能力と安全余裕に合わせて手法を選択する。主要 Datum を確立し治具を通じて使用すること。

実践的組立ワークフロー（現場指向）

1. すべてのマーティング面を検査・清掃する。偏心を引き起こすバリを除去。
2. 多品サブアセンブリのアライメントには精密治具を用いる。Datum フィーチャは設計通り自由度を拘束しなければならない。
3. インターフェランスフィットには制御プレスまたは油圧治具を適用する。力対変位曲線を監視し急変を検出する。
4. 校正済みトルクレンチでファスナを締め付け、重要継手では直接張力指示ワッシャまたは超音波プリロード測定を用いる。
5. 機能的ランインを実施：低トルクで期待可動域を回転させ、軸受を座付させ早期に固着を検出する。

実装：実践計算と自動化以下の Python スニペットは、産業用ヒューマノイドアクチュエータの股関節ファスナについて、トルクから目標プリロードを計算し、軸応力をチェックし、せん断余裕を評価する。組立スクリプトに組み込む生産チェックを示す。

コードサンプル 31 股アクチュエータボルトのボルトプリロード・せん断チェック

```
#!/usr/bin/env python3
"""
Production-ready_bolt_preload_and_shear-margin_calculator
ROS2 ノードとしても動作可能（rclpy 使用）
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Final

import rclpy
from rclpy.node import Node
from std_msgs.msg import Float64MultiArray

# ----- 定数 -----
# 鋼の代表強度（Pa）
```

```
STEEL_SHEAR_STRENGTH: Final[float] = 0.6e9 # 0.6 GPa
```

```
# ----- データ構造 -----
```

```
@dataclass(slots=True, frozen=True)
```

```
class BoltSpec:
```

```
    """ボルト仕様を不変保持"""
```

```
    T: float          # 締付トルク [N・m]
```

```
    d: float          # 公称径 [m]
```

```
    K: float          # トルク係数 [-]
```

```
    A_t: float        # 引張応力面積 [m2]
```

```
    A_s: float        # せん断応力面積 [m2]
```

```
@dataclass(slots=True, frozen=True)
```

```
class LoadCondition:
```

```
    """荷重条件"""
```

```
    M_r: float        # 伝達モーメント [N・m]
```

```
    r_eff: float      # 実効半径 [m]
```

```
    mu: float         # 摩擦係数 [-]
```

```
# ----- 計算ロジック -----
```

```
def calc_bolt_areas(d: float) -> tuple[float, float]:
```

```
    """引張・せん断面積を算出"""
```

```
    A_t = math.pi * (d / 2.0) ** 2
```

```
    A_s = 0.7854 * d ** 2 * 0.6
```

```
    return A_t, A_s
```

```
def evaluate_bolt(spec: BoltSpec, load: LoadCondition) -> tuple[float, float, float]:
```

```
    """
```

```
        プリロード・軸応力・せん断余裕を返す
```

```
        Returns:
```

```
        F_p[N], sigma_a[Pa], shear_margin[Pa]
```

```
        """
```

```
        F_p = spec.T / (spec.K * spec.d)
```

```
        # トルク則
```

```
        sigma_a = F_p / spec.A_t
```

```
        # 軸応力
```

```
        tau = (load.M_r / load.r_eff) / spec.A_s
```

```
        # 発生せん断応力
```

```
        shear_margin = STEEL_SHEAR_STRENGTH - tau
```

```
        # 余裕（絶対値）
```

```
return F_p, sigma_a, shear_margin
```

```
# ----- ROS 2 ノード -----
```

```
class BoltCheckNode(Node):
```

```
    def __init__(self) -> None:
```

```
        super().__init__("bolt_check_node")
```

```
        self.pub = self.create_publisher(Float64MultiArray, "~/bolt_diagnostics", 10)
```

```
        timer_period: float = 1.0 # [s]
```

```
        self.timer = self.create_timer(timer_period, self.timer_callback)
```

```
    # パラメータ読取
```

```
        self.declare_parameter("torque_Nm", 25.0)
```

```
        self.declare_parameter("diameter_m", 0.010)
```

```
        self.declare_parameter("torque_coeff", 0.2)
```

```
        self.declare_parameter("moment_Nm", 12.0)
```

```
        self.declare_parameter("radius_m", 0.015)
```

```
        self.declare_parameter("mu", 0.15)
```

```
    def timer_callback(self) -> None:
```

```
        T = self.get_parameter("torque_Nm").value
```

```
        d = self.get_parameter("diameter_m").value
```

```
        K = self.get_parameter("torque_coeff").value
```

```
        M_r = self.get_parameter("moment_Nm").value
```

```
        r_eff = self.get_parameter("radius_m").value
```

```
        mu = self.get_parameter("mu").value
```

```
        A_t, A_s = calc_bolt_areas(d)
```

```
        spec = BoltSpec(T=T, d=d, K=K, A_t=A_t, A_s=A_s)
```

```
        load = LoadCondition(M_r=M_r, r_eff=r_eff, mu=mu)
```

```
        F_p, sigma_a, margin = evaluate_bolt(spec, load)
```

```
    # ログ出力 (日本語)
```

```
        self.get_logger().info(f"プリロード: {F_p:.1 f} N")
```

```
        self.get_logger().info(f"軸応力: {sigma_a/1e6:.2 f} MPa")
```

```
        self.get_logger().info(f"せん断余裕: {margin/1e6:.2 f} MPa")
```

```
    # 診断データ配信
```

```
        msg = Float64MultiArray()
```

```

        msg.data = [F_p, sigma_a, margin]
        self.pub.publish(msg)

# ----- スタンドアロン実行 -----
def main_standalone() -> None:
    """ROS無しで単体実行"""
    # 初期値（股アクチュエータ例）
    T = 25.0
    d = 0.010
    K = 0.2
    M_r = 12.0
    r_eff = 0.015
    mu = 0.15

    A_t, A_s = calc_bolt_areas(d)
    spec = BoltSpec(T=T, d=d, K=K, A_t=A_t, A_s=A_s)
    load = LoadCondition(M_r=M_r, r_eff=r_eff, mu=mu)

    F_p, sigma_a, margin = evaluate_bolt(spec, load)

    print(f"#_プリロード_(N):_{F_p}")
    print(f"#_軸応力_(MPa):_{sigma_a/1e6}")
    print(f"#_せん断応力_(MPa):_{(M_r/r_eff)/A_s/1e6}")
    print(f"#_せん断余裕_(MPa):_{margin/1e6}")

def main() -> None:
    try:
        rclpy.init()
        node = BoltCheckNode()
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        rclpy.shutdown()

if __name__ == "__main__":
    # ROS環境変数があればノード起動、なければスタンドアロン

```

```

if "ROS_DOMAIN_ID" in __import__("os").environ:
    main()
else:
    main_standalone()

```

検査・試験戦略

- ・関節サブアセンブリをモータ駆動し、低速トルクスイープを実施しエンコーダオフセットを記録する。最終閉鎖前にアライメント異常を検出するためにこれらオフセットを用いる。
- ・代表関節に対して繰返し荷重試験を実施し、フレットングまたはプリロード損失を検出する。
- ・接着剤またはプレスフィット使用時は熱サイクルを実施し、差動膨張問題を明らかにする。

設計トレードオフと運用上のリスク

- ・高プリロードは継手剛性と疲労寿命を向上させるが、ボルト降伏リスクを高め分解を困難にする。保守性とバランスを取る。
- ・厳格公差は運動精度を向上させるが、製造コストと不良率を増加させる。
- ・過剰インターフェランスは微小運動を減らすすが、組立時に軸受を過負荷にする可能性がある。
- ・緩い組立管理は制御系誤差を増幅し、歩行不安定またはセンサドリフトを引き起こす。

工学への影響：図面にトルク・プリロード仕様を定義し、組立工程に測定ポイントを含め、検証ログを自動化する。これらの手順は手戻りを減らし、ヒューマノイドの安全性とミッション信頼性を劣化させる現地故障を軽減する。

9.2 フレームワークのストレステスト

組立チェックリストでは、許容嵌め合わせ、締付トルク、およびアライメント公差を設定した。ストレステストは、これらの組立判断を実荷重下で検証し、シャーシ、継手、およびマウントの弱点を明らかにする。

ストレステストは、機械的フレームワークが機能および安全要件を満たすことを確認するため、荷重および摂動を制御して適用するものである。エンジニアは、準静的、動的（衝撃を含む）、および疲労（繰返し）試験という3つの相補的な試験クラスを実施する。各クラスは特定の破損モードに対応し、設計イテレーションに情報を提供する。

問題定義. 人型胴体および脚フレームワークについて、構造部材、溶接部、および締結継手が降伏、座屈、緩み、またはバランスや作動を損なう損傷の蓄積なく、運用荷重に耐えることを実証する。検証すべき運用制約は以下を含む：

- ・エンドエフェクタにおける最大ペイロードを重心安定性を保ちながら維持。
- ・回復動作中のピーク関節トルク。
- ・リンク部材および締結具に疲労を誘発する繰返し歩容サイクル。
- ・アクチュエータおよび外部摂動によって励振される振動モード。

技術解析. ミッションプロファイルから必要な荷重ケースを導出することから始める。例えば、押圧回復イベントは足部に反力を生じる。軸力および曲げ荷重を受ける簡略化肢部材を表現し、座屈お

よび von Mises 基準をチェックする：

$$[H]P_{cr} = \frac{\pi^2 EI}{(KL)^2}, \quad (75)$$

ここで E はヤング率、 I は断面二次モーメント、 L は有効長、 K は柱有効長係数である。適用圧縮力 P を P_{cr} と比較して座屈リスクを評価する。

延性部材には von Mises を用いて合成応力を計算する：

$$[H]\sigma_{vm} = \sqrt{\sigma_x^2 + \sigma_y^2 + \sigma_z^2 - \sigma_x\sigma_y - \sigma_y\sigma_z - \sigma_z\sigma_x + 3\tau_{xy}^2 + 3\tau_{yz}^2 + 3\tau_{zx}^2}. \quad (76)$$

設計目標： $\sigma_{vm}/\sigma_{yield} \leq FOS^{-1}$ を、破損の影響に応じて通常 1.5~3 の安全率 (FOS) で維持する。

疲労評価は、S-N データが利用可能な場合に使用する。高サイクル疲労には Basquin 関係を用いる：

$$[H]\sigma_a = \sigma'_f (2N)^b, \quad (77)$$

ここで σ_a は応力振幅、 N は破損までのサイクル数、 σ'_f は疲労強度係数、 b は疲労指数である。ミッションデューティサイクルを臨界応力集中における等価サイクルに変換して寿命を予測する。

試験計測および治具. 以下のセンサおよび配置を用いる：

- ・高応力位置および溶接トウにひずみゲージを配置し、局所ひずみを直接測定。
- ・足部インターフェースに 3 軸ロードセルおよびカスタム治具を用いて制御荷重を適用。
- ・骨盤および胸部に慣性計測装置 (IMU) を配置し、衝撃時の角加速度を記録。
- ・加速度計およびレーザ振動計をモーダル試験および減衰推定に使用。
- ・アクチュエータのトルクおよび位置エンコーダを用いて閉ループ荷重制御を実施。

試験プロトコルテンプレート. 最悪ケース運動を繰り返し実行する再現可能なシーケンスを実装する：

1. 静的荷重シーケンス. 荷重を段階的に増加させ、線形ひずみ応答および弾性回復を検証。ひずみ対荷重を記録し、剛性を算出。
2. モーダルスイープ. 計測インパクトハンマまたは振動機で構造を励振し、固有振動数および減衰比を特定。
3. 動的摂動. 胴体中心に横インパルスを加えて突き飛ばしをシミュレート。ピーク関節トルクおよび足部反力を取得。
4. 疲労サイクリング. 代表歩容サイクルを所定サイクル数または破損基準超過まで実施。
5. 環境コンディショニング. 熱サイクル後に臨界試験を繰り返し、材料および締結具劣化を明らかにする。

自動化および安全インターロックは不可欠である。閉ループアクチュエータ制御を用いて試験装置の過荷重を回避する。以下の Python スニペットは、ひずみをログし閾値超過で停止する最小自動トルクリン試験を示す。ControllerAPI および SensorAPI はハードウェアインターフェースに置換する。

コードサンプル 32 Automated torque-ramp with threshold stop

```
#!/usr/bin/env python3
import argparse
```

```

import csv
import logging
import os
import time
from dataclasses import dataclass
from math import isfinite
from pathlib import Path
from typing import List, Optional, Protocol

import numpy as np

# ----- #
# ハードウェア抽象インターフェース（本番環境では実ドライバに置換）
#
# ----- #

class ControllerInterface(Protocol):
    def enable_safety_lock(self) -> None: ...
    def disable_safety_lock(self) -> None: ...
    def set_torque(self, joint_id: int, torque: float) -> None: ...
    def state(self) -> "JointState": ...

class SensorInterface(Protocol):
    def read_strain(self) -> np.ndarray: ...
    def read_force(self) -> np.ndarray: ...

@dataclass
class JointState:
    pos: float
    vel: float
    eff: float

# ----- #
# 本番用実装
#
# ----- #

class TorqueRampConfig:
    def __init__(self) -> None:
        self.torque_max: float = 50.0 # Nm

```

```

self.torque_step: float = 1.0 # Nm/sec
self.strain_threshold: float = 2000e-6 # micro-strain
self.dt: float = 0.2 # 指令更新周期
self.joint_id: int = 2
self.output_dir: Path = Path(os.environ.get("RAMP_LOG_DIR", "."))

```

```
class TorqueRampTest:
```

```

    def __init__(
        self,
        controller: ControllerInterface,
        sensors: SensorInterface,
        cfg: TorqueRampConfig,
    ) -> None:
        self.ctrl = controller
        self.sns = sensors
        self.cfg = cfg
        self.logger = logging.getLogger(self.__class__.__name__)

```

```
# 緊急停止
```

```

def _emergency_stop(self, joint_id: int) -> None:
    self.ctrl.set_torque(joint_id, 0.0)
    self.ctrl.disable_safety_lock()
    self.logger.error("緊急停止：ひずみ閾値超過")

```

```
# ログ書き出し
```

```

def _log_sample(self, writer: csv.writer, torque: float) -> None:
    strain = self.sns.read_strain()
    force = self.sns.read_force()
    state = self.ctrl.state()
    writer.writerow(
        [
            time.time(),
            torque,
            *strain,
            *force,
            state.pos,
            state.vel,
            state.eff,
        ]
    )

```

```

# メインループ
def run(self) -> None:
    self.ctrl.enable_safety_lock()
    torque = 0.0
    log_path = self.cfg.output_dir / f"torque_ramp_{int(time.time())}.csv"
    with log_path.open("w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(
            ["timestamp", "command_torque"]
            + [f"strain_{i}" for i in range(len(self.sns.read_strain()))]
            + [f"force_{i}" for i in range(len(self.sns.read_force()))]
            + ["pos", "vel", "eff"]
        )
        while torque <= self.cfg.torque_max:
            self.ctrl.set_torque(self.cfg.joint_id, torque)
            time.sleep(self.cfg.dt)
            self._log_sample(writer, torque)
            if np.any(np.abs(self.sns.read_strain()) > self.cfg.strain_threshold):
                self._emergency_stop(self.cfg.joint_id)
                raise RuntimeError("Strain_threshold_exceeded")
            torque += self.cfg.torque_step
        self.ctrl.set_torque(self.cfg.joint_id, 0.0)
        self.ctrl.disable_safety_lock()
        self.logger.info(f"試験完了：ログ={log_path}")

# ----- #
# ハードウェアスタブ（本番では削除）
#
# ----- #

class FakeController:
    def enable_safety_lock(self) -> None: pass
    def disable_safety_lock(self) -> None: pass
    def set_torque(self, _: int, __: float) -> None: pass
    def state(self) -> JointState: return JointState(0.0, 0.0, 0.0)

class FakeSensors:
    def read_strain(self) -> np.ndarray: return np.zeros(4)
    def read_force(self) -> np.ndarray: return np.zeros(3)

```

```

# ----- #
# エントリーポイント
#
# ----- #

def main() -> None:
    logging.basicConfig(level=logging.INFO)
    parser = argparse.ArgumentParser()
    parser.add_argument("--joint", type=int, default=2, help="対象ジョイントID")
    parser.add_argument("--max-torque", type=float, default=50.0, help="最大トルク [Nm]")
    parser.add_argument("--step", type=float, default=1.0, help="トルク増分 [Nm]")
    parser.add_argument("--strain-limit", type=float, default=2000e-6, help="ひずみ閾値")
    parser.add_argument("--log-dir", type=Path, default=Path("."), help="ログ保存先")
    args = parser.parse_args()

    cfg = TorqueRampConfig()
    cfg.joint_id = args.joint
    cfg.torque_max = args.max_torque
    cfg.torque_step = args.step
    cfg.strain_threshold = args.strain_limit
    cfg.output_dir = args.log_dir
    cfg.output_dir.mkdir(parents=True, exist_ok=True)

    test = TorqueRampTest(FakeController(), FakeSensors(), cfg)
    test.run()

if __name__ == "__main__":
    main()

```

データ解析および合格／不合格指標. 各試験に対して以下を算出：

- ひずみ対荷重の傾きから弾性剛性。
- 除荷後の残留変形。
- 疲労荷重前後のモーダル振動数変化。
- 混合振幅には Miner 則を用いた累積損傷指標。

閾値を設定：

1. 降伏回避：主要構造部材について最大 $\sigma_{vm} < 0.6\sigma_{yield}$
2. 座屈余裕：細長比の高い圧縮部材で $P/P_{cr} < 0.5$ 。
3. 疲労寿命：予測サイクル N_{life} がミッションサイクルを安全率 2 で上回る。

4. 締結具緩み：熱および振動コンディショニング後のトルク損失 < 10%。

設計および運用への影響。ストレステストは、質量を安全に削減できる部位および補強が必要な部位を定量化する。また、剛性と衝撃吸収、安全率とシステム質量の間のトレードオフも明らかにする。運用上のリスクには、未検出疲労亀裂、締結具疲労、動的荷重下でのモータ過電流が含まれる。対策として、標的補強、臨界継手への冗長センシング、進行的構造劣化をフラグする自動ランタイム監視が挙げられる。

9.3 機械的故障のトラブルシューティング

組立検査とフレームワーク応力試験を踏まえ、トラブルシューティングは観察された症状を測定可能な故障モードに変換することに焦点を当てる。実用的な診断には、定量化された閾値、再現可能な試験、修理前の安全な隔離手順が必要である。

明確な問題文から始める：症状、動作条件、最近の出来事を記述する。例：「左膝は5時間歩行後、20 N・m 負荷下で予期しないコンプライアンスを示す。」これを測定可能な問いに翻訳する：

1. 故障は構造的、運動学的、または駆動系関連か？
2. 制御された入力下で再現するか？
3. どのセンサーと受動検査が原因を局在化できるか？

推奨診断ワークフローは修理までの時間を短縮し、不要な分解を回避する。

- ・電源を遮断し関節をロックしての視覚・触覚検査。
- ・低リスク駆動下での計測付き機能試験で故障を再現。
- ・定量的測定と傾向ログ記録。
- ・コントローラーを無効化または部品交換によりサブシステムを隔離。
- ・的確な修正処置を適用し試験を繰り返す。

一般的な機械的故障カテゴリと測定可能な指標：

- ・締結部緩み：隙間、トルク損失、断続的騒音。トルクレンチまたは超音波ボルトスキャナーでプレロードを測定。
- ・軸受劣化：摩擦増加、発熱、または遊び。軸受寿命関係式を用いて残存寿命を推定。
- ・ギアボックスバックラッシュと摩耗：小振幅振動試験中のヒステリシス角。
- ・構造亀裂：染料浸透または超音波スキャンで可視化される亀裂進展。
- ・ミスアライメントとバインディング：モータ電流増加、不均一エンコーダ読み値、スティックスリップ挙動。

交換優先順位付けに軸受寿命を使用。玉軸受では、基本定格寿命回転数 L_{10} は

$$[H]L_{10} = \left(\frac{C}{P}\right)^p, \quad (78)$$

で満たされ、玉軸受では $p = 3$ 、 C は動定格荷重、 P は等価動荷重である。軸回転数 n (rev/min) で時間に換算：

$$[H]L_{10h} = \frac{L_{10}}{60n}. \quad (79)$$

L_{10h} がミッション時間に近づいたら、緊急交換ではなく計画的な予防交換をスケジュールする。

小さな符号付きトルクスイープを指令しエンコーダヒステリシスを測定してバックラッシュを定量化。ヒステリシス角を

$$\Delta\theta_{\text{hyst}} = \theta_+ - \theta_-,$$

と定義し、 θ_+ は正トルク時の角度、 θ_- は同じ大きさの負トルク時の角度である。 $\Delta\theta_{\text{hyst}} > \theta_{\text{thresh}}$ ならギアボックス摩耗をフラグし、歯形とクリアランスを検証する。

アクチュエータトルク、関節剛性、観測たわみを関連付け、構造たわみと制御誤差を区別する。線形剛性 k_{joint} でモデル化された関節では、

$$\tau = k_{\text{joint}} \Delta\theta + \tau_{\text{fric}},$$

で、 τ は指令トルク、 $\Delta\theta$ は弾性回転角、 τ_{fric} は摩擦トルクである。剛性推定のため整理すると：

$$k_{\text{joint}} = \frac{\tau - \tau_{\text{fric}}}{\Delta\theta}.$$

ベースラインと現在の試験間で k_{joint} が低下すると、緩んだ構造接続または材料降伏を示す。

計測付き試験は不可欠である。同期ログに以下を含める：

- 関節エンコーダ位置と速度。
- モータ電流と電圧。
- 電流-トルクマップによるトルク推定値。
- 全身過渡荷重用 IMU 加速度。
- 足部または手首部の力-トルクセンサ読み値。

自動化スクリプトが再現可能な試験と即時異常検出を簡素化する。以下のリストは、小振幅関節スイープを実行しデータをログし、ヒステリシス角を計算する最小限の Python ルーチンを示す。

コードサンプル 33 関節ヒステリシス試験とログ（簡略版）

```
# connect to robot API (placeholder)
robot = connect_robot() # establish comms
joint = "knee_left"      # target joint
amp = 0.05                # small rad amplitude
rate = 100                # Hz logging

# perform sweep: positive then negative torque
robot.set_mode('torque') # low-level control mode
robot.command_torque(joint, +0.5) # hold small positive torque
time.sleep(0.5) # settle
theta_pos = robot.read_encoder(joint) # measure angle

robot.command_torque(joint, -0.5) # hold small negative torque
time.sleep(0.5) # settle
theta_neg = robot.read_encoder(joint) # measure angle
```

```
# compute and log hysteresis
delta_theta = theta_pos - theta_neg
log = {'joint': joint, 'theta_pos': theta_pos, 'theta_neg': theta_neg, 'delta_theta': delta_theta}
save_log(log) # persistent storage
print("Delta_theta(rad):", delta_theta)
```

connect_robot() をプラットフォーム API に置き換える。本番コードでは安全インターロックと電流制限を追加する。

実用的測定許容差は誤検出を減らす。推奨閾値：

- バックラッシュ： $\theta_{\text{thresh}} = 0.5\text{--}1.0$ 度、精密膝関節用。
- 軸受温度上昇：定常状態 $\Delta T > 20^\circ\text{C}$ 以上の環境温度上昇は潤滑またはプレロード故障を示す。
- モータ電流：同じ動作プロファイルでベースラインより持続的に $>10\%$ 増は機械的ドラッグを暗示。

隔離技術：

1. アクチュエータドライバを交換し電子系異常を排除。
2. 手動で関節をバックドライブし摩擦と制御アーチファクトを分離。
3. 疑わしい伝達機構を校正済み試験負荷に交換。
4. モーダルハマー試験を用い緩んだ接続と共振シフトを検出。

安全および運用制約：

- 分解前は常に電源を遮断し機構をロックする。
- 計測付き試験では初期段階デバッグで安全ケージとテザー付きロボットを使用する。
- 比較診断用にバージョン管理されたベースラインログを維持する。

エンジニアリングへの影響、トレードオフ、リスク：

- 予防交換はミッション故障を減らす但し保守費用を増やす。
- 厳密なプレロードと低バックラッシュは制御を改善するが摩擦と発熱リスクを高める。
- 自動診断は修理を高速化するがサンプリングが粗いと断続的構造故障を見落とす可能性がある。
- 電氣的症状を機械的と誤って帰属すると繰返し故障のリスクがある。

9.4 実世界における組立ての課題

前節で機械的故障の分離と応力試験による構造マージンの検証に焦点を当てたが、本小節では優れた設計が実際に失敗する組立て固有の障害を考察する。焦点は運用上のものであり、ヒューマノイドロボットの組立てにおいて、公差、治具、ケーブル、ヒューマンファクターがどのように相互作用して故障を生むかを示す。

問題提示：多自由度ヒップ・ニーモジュールを、生産および現場条件下で運動精度、トルク伝達、センサライメント要件を満たすよう組立てる。実際の組立てでは4つの課題クラスに直面する：幾何公差の累積、インターフェースおよびコネクタのばらつき、ケーブルおよびハーネスの管理、プロセスの再現性。

技術分析：

- 公差累積。嵌合インターフェースの寸法誤差はリンク間で累積し運動精度を低下させる。最悪ケース（WC）累積は絶対公差を合計する：

$$[H]\Delta_{WC} = \sum_{i=1}^n t_i, \quad (80)$$

ここで t_i は個々の部品公差である。統計的（root-sum-square, RSS）累積は独立した製造誤差をモデル化する：

$$[H]\Delta_{RSS} = \sqrt{\sum_{i=1}^n t_i^2}. \quad (81)$$

安全上重要なジョイントでは保証されたクリアランスのために (80) を用いる。較正計画のための公称運動ばらつきを予測するには (81) を用いる。

- IMU およびエンコーダのミスアライメント。角度ミスアライメントは推定姿勢およびジョイント空間運動学にバイアスを生じる。小角近似により一次の姿勢誤差はミスアライメント角 θ に比例する：測定誤差 $\approx \theta$ ラジアン。胴体への IMU 取付け誤差は、較正中に補償されない場合、全身状態推定ドリフトを生じる。
- 締結およびプリロードのばらつき。ボルトトルクからプリロードへの変換は

$$[H]T = KFd, \quad (82)$$

に従い、トルク T 、プリロード F 、公称径 d 、ナット係数 K を伴う。 K のばらつき（潤滑、表面粗さ）はプリロードの散在を引き起こす。プリロード不足または過剰はベアリング寿命およびジョイントコンプライアンスを変化させる。

- ケーブルルーティング、摩擦、摩耗。太もものハーネスでは、導管の詰まりまたは鋭い曲げがケーブルドラッグトルクおよびストレインリリーフ負荷を変化させる。動的ケーブル摩擦はヒップアクチュエータに寄生トルクを加え、制御安定性を劣化させる。
- 熱膨張および材料ミスマッチ。アルミニウムと鋼の締結材間の差動膨張は、代表的操作温度変動にわたりプリロードおよびアライメントを変化させる。線膨張は $\Delta L = \alpha L \Delta T$ であり、ここで α は線膨張係数である。
- コネクタおよびサプライヤのばらつき。コネクタロット間のわずかな寸法差は断続的な電気接触、EMI 感受性、組立て時間の増加を引き起こす。

実装戦略：

1. ヒューマノイドのための組立て設計（DFA）

- 単一の基準面から伝播するデータスキームを指定する。インターフェースで自由度を拘束するために幾何公差記入（GD&T）を用いる。
- 繰返し可能なサブアセンブリ位置決めのための運動学的ロケータ（3点接触）を導入する。

2. 治具およびツーリング

- ベアリングプリロードおよびエンコーダインデクシングのための精密組立て治具を用いる。治具は必要な人間の判断を減らし、スループットを向上させる。
- トレーサビリティのためのトレーサブルログを備えたトルク制御ドライバを実装する。トレーサビリティのために各締結具に印加されたトルクを記録する。

3. 較正治具および測定

- 非本質的自由度をロックし、レーザトラッカまたは光学 CMM でジョイント軸ミスアライメントを測定する較正治具を作成する。
- 直交軸を励起するモーションシーケンスを用いて IMU 姿勢較正を自動化し、センサ対ボディ回転を解く。

4. ハーネス設計およびストレインリリーフ

- 制御された曲げ半径とアクチュエータ出口でのケブラー強化ストレインリリーフポイントを備えたチャンネルにケーブルをルーティングする。
- ハーネスが回転スリーブを通過する場所には低摩擦ライナーを用いる。

5. プロセス制御および検査

- 主要寸法に対して統計的プロセス制御 (SPC) を適用する。プロセス能力指数 C_{pk} が閾値を下回る場合、部品をリジェクトまたは再加工する。
- エンドオブラインファンクショナルテストを用いる：閉ループトルクスweep、エンコーダインデックス検証、IMU 静止バイアス測定。

実践例：膝ジョイント組立て

- 問題：仕様を超えるエンコーダリング振れが制御ジッタを引き起こす。
- 分析：ベアリングボア同心性およびシャフト振れを測定し、(81) で合成誤差を定量化し、次いで加工公差の絞り込みまたはコンプライアントカップリングの追加を決定する。
- トレード：より厳しい加工は単体コストを増加させる；コンプライアントカップリングは伝達剛性を低下させ、制御帯域に影響を与える。

コード実用：組立てエンジニアのための迅速公差集約および意思決定支援。

コードサンプル 34 公差累積計算機

```
#!/usr/bin/env python3
"""
Tolerance_stack-up_analyzer_for_encoder_mounting_interface.
Computes_worst-case_and_RSS_(Root-Sum-Square)_tolerance_stack-up.
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Final, List

# 許容差仕様 (mm)
@dataclass(frozen=True)
class ToleranceSpec:
    bearing_seat: float
```

```

    spacer: float
    shaft_runout: float

    def to_list(self) -> List[float]:
        return [self.bearing_seat, self.spacer, self.shaft_runout]

# 組立許容基準 (mm)
MAX_RUNOUT: Final[float] = 0.15 # エンコーダ最大許容振れ
RSS_THRESHOLD_RATIO: Final[float] = 1.0 / 3.0 # RSS警告閾値比率

def compute_stack_up(spec: ToleranceSpec) -> tuple[float, float]:
    """
    最悪値とRSS積み重ねを計算
    Returns (worst_case, rss)
    """
    tolerances = spec.to_list()
    worst_case = sum(tolerances)
    rss = math.sqrt(sum(t * t for t in tolerances))
    return worst_case, rss

def evaluate_stack_up(worst_case: float, rss: float) -> None:
    """
    積み重ね結果を評価し必要に応じて警告
    """
    print(f"Worst-case: {worst_case:.3f} μm")
    print(f"RSS: {rss:.3f} μm")

    if rss > MAX_RUNOUT * RSS_THRESHOLD_RATIO:
        print("Action: tighten tolerances or add compliant coupling")

def main() -> None:
    # サプライヤデータに応じて調整
    spec = ToleranceSpec(bearing_seat=0.10, spacer=0.08, shaft_runout=0.12)
    worst_case, rss = compute_stack_up(spec)
    evaluate_stack_up(worst_case, rss)

if __name__ == "__main__":
    main()

```

運用上の影響、トレードオフ、リスク：

- 公差を厳しくすると組立てトラブルシューティングは減少するが、製造コストおよびリードタイムが増加する。重要インターフェースに対して標準化された検査を実装することでバランスを取る。
- 熟練した手作業組立てへの過度の依存はばらつきおよびスケーリングリスクを増加させる。繰返し性を向上させるために治具およびトルク制御ソーリングに投資する。
- 組立て誤差を補償するためのコンプライアンス追加は堅牢性を向上させるが、制御剛性および帯域幅を低下させる。
- 不適切なハーネスルーティングおよびコネクタ選択は、動的負荷下で潜在的な電気故障を引き起こす。フルジョイントレンジおよび既知のケーブル疲労サイクルに対応できるようハーネスを設計する。
- 機械公差を緩和しようとするほど較正の複雑さが増加する。各較正ステップは故障モードおよび現場保守負担を追加する。

具体的指針：データ駆動設計を優先し、測定により組立てステップを計測化し、最悪ケースおよび統計モデルの両方を用いて公差影響を定量化する。これらの対策は現地故障を減少させ、製造コスト、制御性能、信頼性間のトレードオフを明確にする。

ロボティクスのための電気工学

10 電力システム

10.1 電源の選択

前節では、ヒューマノイドロボットにおけるバッテリーと有線電源を対比し、モビリティと連続電源のトレードオフを示した。本節では、ヒューマノイドに実用的な電源を選ぶための厳密な工学レベルの手法を展開し、電気的要求を機械的および運用上の制約に結びつける。

問題定義. ヒューマノイドロボットは、センサと計算のための連続ベースライン電力に加え、移動と操作用の過渡的なピーク電力を供給しなければならない。設計目標は、運用時間、ピーク電流、安全性、重量、パッケージングの制約を満たしながら運用上のリスクを最小化する電源（またはハイブリッド組合せ）を選択することである。

技術的分析. システム要求を電氣的プリミティブに変換することから始める：

1. ベースライン負荷: センサ、コントローラ、オンボード計算。例: NVIDIA クラスのオンボードコンピュータが消費する $P_{\text{comp}} = 60 \text{ W}$ 。
2. アクチュエータ負荷: モータはトルク τ と角速度 ω を供給する。モータ 1 台あたりの機械的電力は $P_m = \tau\omega$ である。全モータのピーク機械電力を合計し、その後アクチュエータドライブと伝達効率 η_{drive} を含める。
3. ピークおよび平均電力バジェット: ミッションプロファイルに対して P_{avg} を、最悪ケースの機動に対して P_{peak} を定義する。

バッテリーサイジングへの変換. 必要なバッテリー容量（アンペア時）は

$$[H]C_{\text{Ah}} = \frac{P_{\text{avg}} t_{\text{runtime}}}{V_{\text{nom}} \eta_{\text{sys}}}, \quad (83)$$

で与えられる。ここで V_{nom} は公称パック電圧、 t_{runtime} は所望のミッション時間、 η_{sys} は変換損失を集約した効率である。ピーク電流チェックには

$$[H]I_{\text{peak}} = \frac{P_{\text{peak}}}{V_{\text{nom}}}, \quad (84)$$

を用いる。 I_{peak} がバッテリーの最大連続およびパルス放電限界（C レートで表される）を下回ることを確認する： $I_{\text{max}} = C_{\text{rate}} \times C_{\text{Ah}}$

内部抵抗 R_{int} を用いた電圧サグを考慮する。大電流時、端子電圧は $V_{\text{term}} = V_{\text{oc}} - IR_{\text{int}}$ となる。コントローラおよびモータドライバが最悪ケースの V_{term} に耐えることを確保する。

電源オプションと工学への影響：

- ・リチウムイオンバッテリーパック：一般的な電気化学オプションの中で最高の重量エネルギー密度。数時間の自律性を必要とする無拘束ヒューマノイドに使用する。セルバランシング、熱暴走リスク、機械的配置による安定性維持を考慮する。
- ・スーパーキャパシタ：踏み切りや衝撃時の短時間バーストの吸収または供給に高い出力密度。ハイブリッドアーキテクチャでバッテリー C レートストレスを軽減しサイクル寿命を延長するために使用する。
- ・燃料電池：長時間ミッションに高いエネルギー密度。空気供給、熱管理、過渡ピークを補うためのハイブリッド化が必要。
- ・有線テザー：エネルギー貯蔵重量を排除するが、モビリティを制限し転倒危険を引き起こす。実験室およびテスト環境に最適。
- ・ハイブリッドシステム：バッテリー＋スーパーキャパシタまたはバッテリー＋燃料電池。ハイブリッドはピークイベントを高出力コンポーネントに割り当てることでバッテリーストレスとサイズを削減する。

実装チェックリスト. 防御可能な選択のために以下の手順に従う：

1. モータトルク-速度曲線およびミッションプロファイルから P_{avg} と P_{peak} を測定または推定する。
2. I^2R 損失を最小化しモータドライババスに一致するように V_{nom} を選ぶ。典型的なヒューマノイドパックは電流を管理可能にするため 24–48 V を使用する。
3. 式 (83) を通じて C_{Ah} を計算する。経年変化および温度効果に対する予備容量マージン（20–30%）を加える。
4. 式 (84) を用いて C レートおよび R_{int} 制約を検証する。
5. セル監視、バランシング、熱管理、安全インターロックのためのバッテリー管理システム（BMS）を設計する。
6. バランスエンベロープ内に重心を保ち、冷却のための熱経路を割り当てるために機械的配置を繰り返す。

現実的な例. $P_{\text{avg}} = 200 \text{ W}$ 、短時間プッシュ時のピーク電力 $P_{\text{peak}} = 1500 \text{ W}$ 、所望運用時間 $t_{\text{runtime}} = 2 \text{ h}$ 、公称電圧 $V_{\text{nom}} = 48 \text{ V}$ 、システム効率 $\eta_{\text{sys}} = 0.9$ のヒューマノイドを考える。容量を計算：

$$[H]C_{\text{Ah}} = \frac{200 \cdot 2}{48 \cdot 0.9} \approx 9.26 \text{ Ah}. \quad (85)$$

ピーク電流要求は $I_{\text{peak}} = 1500/48 \approx 31.25 \text{ A}$ である。12 Ah にサイズされたパックでは、ピーク引き

出しはおおよそ $2.6C$ に相当し、多くの高放電リチウムイオンセルにとって許容範囲である；それでも繰り返しバーストが発熱を引き起こす場合はスーパーキャパシタバッファを含めるべきである。

実用的なコード. 以下の Python スニペットは、必要な容量を計算し、C レートをチェックし、ピーク電流時の電圧サグを推定する。

コードサンプル 35 ヒューマノイドミッションプロファイル用バッテリーサイジング計算機

```
"""
Production-ready battery sizing utility for humanoid robots.
ROS2 ノードとしても動作可能 (rclpy 対応)。
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Final

# ROS 2 利用時のみインポートを試みる
try:
    import rclpy
    from rclpy.node import Node
    ROS2_AVAILABLE: Final[bool] = True
except ModuleNotFoundError:
    ROS2_AVAILABLE = False

# 定数は型付きで一元管理
class BatteryConsts:
    """リチウムイオン電池に関する定数"""
    LI_ION_NOMINAL_V: Final[float] = 3.7 # V
    MIN_CELL_V: Final[float] = 3.0 # 過放電保護閾値
    MAX_CELL_V: Final[float] = 4.2 # 満充電時

    @dataclass(slots=True, frozen=True)
    class BatterySizingResult:
        """計算結果を不変オブジェクトで返す"""
        c_required_ah: float
        i_peak_a: float
        implied_c: float
```

```

v_term_peak: float
cells_series: int
can_handle_peak: bool

def battery_size(
    p_avg_w: float,
    p_peak_w: float,
    t_hours: float,
    v_nom_v: float,
    eta_sys: float,
    cell_capacity_ah: float,
    cell_c_rate: float,
    r_int_per_pack_ohm: float,
) -> BatterySizingResult:
    """
    必要なパック容量とセル構成を推定する。
    負値・ゼロ除算はValueErrorで早期リジェクト。
    """
    # 入力値の簡易バリデーション
    if p_avg_w <= 0 or p_peak_w <= 0 or t_hours <= 0 or v_nom_v <= 0:
        raise ValueError("平均・ピーク電力、時間、電圧は正でなければならない")
    if not (0 < eta_sys <= 1):
        raise ValueError("システム効率は0<η≤1")
    if cell_capacity_ah <= 0 or cell_c_rate <= 0 or r_int_per_pack_ohm < 0:
        raise ValueError("セル仕様が不正")

    # 必要容量（有効容量÷効率）
    c_req = (p_avg_w * t_hours) / (v_nom_v * eta_sys)

    # ピーク電流
    i_peak = p_peak_w / v_nom_v

    # パック全体としての暗黙Cレート
    implied_c = i_peak / c_req if c_req > 0 else math.inf

    # ピーク負荷時の終端電圧（実効電圧）
    v_term = v_nom_v - i_peak * r_int_per_pack_ohm

    # 直列セル数（切り上げで安全側に）

```

```

cells_in_series = max(1, math.ceil(v_nom_v / BatteryConsts.LI_ION_NOMINAL_V))

# ピーク電流をセルCレートで満たせるか
can_handle_peak = (cell_c_rate * cell_capacity_ah) >= i_peak

return BatterySizingResult(
    c_required_ah=c_req,
    i_peak_a=i_peak,
    implied_c=implied_c,
    v_term_peak=v_term,
    cells_series=cells_in_series,
    can_handle_peak=can_handle_peak,
)

# -----
# ROS 2ノード化（オプション）
# -----
if ROS2_AVAILABLE:

    class BatterySizingNode(Node):
        def __init__(self) -> None:
            super().__init__("battery_sizer")
            self.declare_parameters(
                namespace="",
                parameters=[
                    ("p_avg_w", 200.0),
                    ("p_peak_w", 1500.0),
                    ("t_hours", 2.0),
                    ("v_nom_v", 48.0),
                    ("eta_sys", 0.9),
                    ("cell_capacity_ah", 3.0),
                    ("cell_c_rate", 10.0),
                    ("r_int_per_pack_ohm", 0.05),
                ],
            )
            self.timer = self.create_timer(1.0, self.run_sizing)

        def run_sizing(self) -> None:
            p_avg = self.get_parameter("p_avg_w").value

```

```

        p_peak = self.get_parameter("p_peak_w").value
        t_hours = self.get_parameter("t_hours").value
        v_nom = self.get_parameter("v_nom_v").value
        eta = self.get_parameter("eta_sys").value
        cap = self.get_parameter("cell_capacity_ah").value
        crate = self.get_parameter("cell_c_rate").value
        r_int = self.get_parameter("r_int_per_pack_ohm").value

    try:
        result = battery_size(p_avg, p_peak, t_hours, v_nom, eta, cap, crate,
                               self.get_logger().info(f"{result}"))
    except ValueError as e:
        self.get_logger().error(f"Invalid params: {e}")

# -----
# 単体実行 (python battery_sizer.py)
# -----
if __name__ == "__main__":
    if ROS2_AVAILABLE and rclpy.ok():
        rclpy.init()
        node = BatterySizingNode()
        try:
            rclpy.spin(node)
        except KeyboardInterrupt:
            pass
        node.destroy_node()
        rclpy.shutdown()
    else:
        # スタンドアロンテスト
        result = battery_size(
            p_avg_w=200,
            p_peak_w=1500,
            t_hours=2,
            v_nom_v=48,
            eta_sys=0.9,
            cell_capacity_ah=3.0,
            cell_c_rate=10,
            r_int_per_pack_ohm=0.05,
        )

```

```
print(result)
```

設計のトレードオフと運用上のリスク. 主なトレードオフは以下を含む：

- 重量対運用時間: より高い重量エネルギー密度は質量を削減するが、しばしばコストと熱イベントのリスクを増大させる。
- 電圧選択: より高い電圧は電流と損失を下げるが、絶縁と安全性を複雑にする可能性がある。
- ハイブリッド化の複雑さ: スーパーキャパシタはバッテリストレスを軽減するが、質量と制御の複雑さを追加する。
- 熱および EMI 効果: 大電流は熱および電磁干渉を生じ、センサおよび計算に影響を与える可能性がある。

工学への影響. 適切な電源選択はバランス、アクチュエータサイジング、ミッション能力に直接影響する。ピーク電流、内部抵抗、セル経年変化、または熱暴走リスクを考慮しなければ、性能の劣化、予期しないシャットダウン、または安全危険が生じる可能性がある。計測に基づくサイジング、堅牢な BMS 設計、電気的要求を満たしながらヒューマノイドの安定性を保つ機械的統合を優先する。

10.2 バッテリ寿命の管理

前小節でのセル選定とパケットポロジのトレードオフを踏まえ、ここではヒューマノイドロボットのミッション継続時間、安全性、俊敏性の目標を満たすシステムレベルのバッテリー管理に焦点を当てる。効果的な管理は、電気化学挙動を制御ポリシー、熱制約、機械設計選択に結びつける。

問題定義と運用目標。ヒューマノイドは動的動作中に高いピーク電力を要求し、計算、センシング、通信により負荷が変動する。エンジニアリング目的は以下の通りである。

- 指定タスクの有用な稼働時間を最大化する。
- 性能劣化を引き起こす電圧サグと熱逸走を防ぐ。
- バッテリ寿命とミッション終了時の予測可能な挙動を保持する。

技術分析：エネルギー収支とバッテリー電気モデル。

- エネルギーバジェット：アクチュエータ、電子機器、損失に必要な正味エネルギーを算出する。タスクプロファイル $P(t)$ に対して、

$$[H]E_{\text{task}} = \int_{t_0}^{t_f} P(t) dt. \quad (86)$$

利用可能なパックエネルギー (Wh) は概ね $E_{\text{pack}} = Q_{\text{Ah}} \cdot V_{\text{nom}}$ であり、 Q_{Ah} はアンペア時定格容量である。

- 定常平均負荷下での稼働時間見積もり：

$$[H]t_{\text{run}} \approx \frac{Q_{\text{Ah}} V_{\text{nom}}}{P_{\text{avg}}}. \quad (87)$$

- 簡易電気モデル：開回路電圧と内部抵抗により負荷時の瞬時電圧を捉える。

$$[H]V_{\text{load}}(t) = V_{\text{oc}}(\text{SOC}(t)) - I(t)R_{\text{int}}. \quad (88)$$

熱損失は $P_{\text{loss}} = I^2(t)R_{\text{int}}$ であり、セル温度を上昇させ老化を加速させる。

- 状態定義：

1. 充電状態 (SOC)：残り利用可能充電量の割合。クーロン計数により

$$[H]SOC(t) = SOC(t_0) - \frac{1}{Q_{\text{nom}}} \int_{t_0}^t I(\tau) d\tau, \quad (89)$$

記号規則は放電電流を正とする。

2. 健康状態 (SOH)：現在容量と定格容量の比。容量試験と内部抵抗傾向から SOH を監視する。

- 推定手法：クーロン計数を周期的な開回路電圧 (OCV) チェックやカルマンフィルタと組み合わせ、動的電流下で SOC を改善する。熱モデルを用いて高温時の利用可能容量を補正する。

実装：ソフトウェアと制御戦略

1. エネルギー認識モーションプランニング

- 歩容プランナの目的関数にエネルギーコストを含める。関節電力 $P_j = \tau_j \omega_j$ は機械仕事に積分される。
- 低エネルギータスクには準静的または受動ダイナミクスを優先する。エネルギーの蓄積と返還により正味バッテリー消費を削減するため、直列弾性アクチュエータを用いる。

2. ピークシェーピングとバッファリング

- DC バスコンデンサまたは小型スーパーキャパシタバンクを用いて短い過渡ピークを供給し、深い電圧降下とコントローラリセットを防ぐ。
- $I(t)$ が I_{max} に近づく際の安全運転のため、電力電子電流制限を実装する。

3. 回生回収

- 下降または減速時に機械エネルギーを回収する。回生エネルギーを $E_{\text{regen}} = \eta_{\text{reg}} \int P_{\text{neg}}(t) dt$ とモデル化し、 η_{reg} は変換効率である。
- 回生はパックの充電受容制約に従って制限する。

4. タスクスケジューリングとグレースフルデグラデーション

- 必須サービス（バランス、安全性）を優先し、SOC が高いときに非クリティカルタスクをスケジュールする。
- センシングと計算の QoS レベルを実装する；低 SOC 条件下では CPU 周波数やセンサフレームレートを削減する。

5. 熱管理と安全性

- 高デューティサイクルには能動冷却；継続タスクには単純な気流または伝熱経路を用いる。
- BMS はセルバランシング、過/不足電圧、過電流、熱遮断を実装しなければならない。

具体的なアルゴリズムビルディングブロック：軽量 SOC 予測器と稼働時間推定器。以下のコードはクーロン計数、簡易回生処理、短時間移動平均による電流平滑化を用いて、ロボットの電源マネージャへの組み込みに適している。

コードサンプル 36 Simple SOC predictor and runtime estimator for onboard power management.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from __future__ import annotations
```

```

import math
from typing import Final

# 充電効率のデフォルト値（リジェネ時）
DEFAULT_ETA_REG: Final[float] = 0.90
# SOC の許容範囲
SOC_MIN: Final[float] = 0.0
SOC_MAX: Final[float] = 1.0

def update_soc(
    soc: float,
    q_ah: float,
    i_a: float,
    dt_s: float,
    eta_reg: float = DEFAULT_ETA_REG,
) -> float:
    """
    クーロンカウンティングによるSOC推定（リジェネ効率考慮）
    i_a > 0: 放電, i_a < 0: 回生
    """
    if q_ah <= 0:
        raise ValueError("q_ah must be positive")
    if dt_s < 0:
        raise ValueError("dt_s must be non-negative")

    # 変化量を Ah 単位で計算
    dah = i_a * dt_s / 3600.0
    if i_a < 0:
        dah *= eta_reg # 回生時は充電効率を適用

    soc_new = soc - dah / q_ah
    # クランプ処理
    return max(SOC_MIN, min(SOC_MAX, soc_new))

def predict_runtime(
    soc: float,
    q_ah: float,
    v_nom: float,

```

```

        i_avg_a: float,
    ) -> float:
        """
        平均電流 i_avg_a で定常放電した場合の残り運転時間を秒単位で返す
        """
        if q_ah <= 0:
            raise ValueError("q_ah must be positive")
        if i_avg_a <= 0:
            return math.inf # 放電電流がゼロ以下なら無限
        ah_rem = soc * q_ah
        return ah_rem / i_avg_a * 3600.0

# 動作確認
if __name__ == "__main__":
    soc = 0.9
    q_ah = 10.0 # 10 Ah パック
    v_nom = 25.9 # 今回は未使用だが将来的な拡張用に保持

    soc = update_soc(soc, q_ah, i_a=15.0, dt_s=0.5) # 15 A バースト
    runtime_s = predict_runtime(soc, q_ah, v_nom, i_avg_a=8.0)
    print(f"SOC: {soc:.3f}, Runtime: {runtime_s:.1f}s")

```

設計トレードオフと運用上のリスク

- 追加のバッテリー質量は移動エネルギーを増加させる；セルエネルギー密度と必要ピーク電力性能を最適化する。重いパックは重力仕事を増やし E_{task} を上昇させる。
- 積極的回生戦略はセル充電受容率を超え、発熱または不均衡を引き起こす可能性がある。
- 不十分な電流バッファリングは衝突または回復中にコントローラブラウナウトを引き起こし、転倒を招く。
- 保守的な SOC カットオフは安全性を高めるが、ミッションの柔軟性を低下させる。

具体的なエンジニアリング影響：BMS テレメトリをロボットのタスクプランナに統合し、モーション制約に熱限界を含め、バッテリーをオーバーサイズすることなくピーク電力要求を満たすようにバッファコンデンサをサイジングする。パック質量とアクチュエータ選定を確定する前に、上記のエネルギー収支手法でトレードオフを定量化する。

10.3 電気設計における安全性

電源を選定し、バッテリー寿命のトレードオフを定量化した後、残る具体的な危険——予想される故障電流、蓄積エネルギー、保守中の人体曝露——に対処する。本小節では、これらのリスクを人型ロボット向けの定量化された設計要件と実践的な実装へと変換する。

問題の定義と工学目的。

- ・短絡または部品故障時の熱損傷と火災を防止する。
- ・運転・保守中の人体への安全な近接を保証する。
- ・故障後も制御性を維持し（安全停止）、潜在的な危険状態を回避する。
- ・組込み制御および電力電子機器に適用される安全規格を満たす。

技術解析：故障モデルと保護装置のサイジング。現実的な故障モデルは、バッテリーの無負荷電圧とその内部インピーダンスから始まる。予想短絡電流は

$$[H]I_{sc} \approx \frac{V_{oc}}{R_{int} + R_{path}}, \quad (90)$$

で与えられる。ここで V_{oc} はバッテリー無負荷電圧、 R_{int} はバッテリー内部抵抗、 R_{path} は母線、コネクタ、配線を経るループ抵抗である。 R_{int} は動作温度と充電状態で実測し、データシート値のみではピークエネルギーを過小評価する。

保護選択を支配する 2 つのエネルギー指標：

$$[H]I^2t = \int_0^{t_{clear}} I(t)^2 dt, \quad (91)$$

および故障に供給される電気エネルギー：

$$[H]E \approx V_{oc} \int_0^{t_{clear}} I(t) dt \approx V_{oc} I_{sc} t_{clear}. \quad (92)$$

ヒューズ、ブレーカ、または MOSFET ベースの電子ブレーカを、それらの遮断 I^2t が導体または PCB の I^2t 耐量より低くなるように選定。逆に、保護装置の通過エネルギーが隣接構造物またはバッテリーを熱暴走に至らしめないことを確認する。

実践的保護要素と配置。

- ・一次保護：連続電流と突入（モータ・コンデンサ）に合わせてサイズされた高速バッテリー側ヒューズ。安全率として、連続電流の少なくとも $1.25 \times$ 定格ヒューズを選定し熱的デレーティングを行う。
- ・二次保護：アクチュエータ毎の電流検出と電子電流リミッタでストール電流を動的に抑制。
- ・プリチャージ抵抗器とコンタクタ：大容量 DC 母線コンデンサ接続時、プリチャージ抵抗で突入電流を制限。プリチャージ時定数 $\tau = R_{pre}C_{bus}$ を算出。
- ・絶縁：高圧母線と制御電子回路間の絶縁；人体アクセス可能箇所には強化絶縁と PEEK/セラミックバリア。
- ・安全機能の冗長化：安全完整性が要求される場合、デュアルチャネル安全リレーまたは安全定格 MOSFET を使用。ISO 13849 に従い性能レベル解析、IEC 61508 に従い SIL 評価を実施。

PCB・配線の熱的考慮。

- ・導体断面積と銅厚は過渡熱容量を考慮して選定。短時間大電流パルスでは、トレースのヒューズ特性は I^2t 対断面積二乗に基づく実験式に従う。設計繰返しでは、測定した母線抵抗から始め、集中熱容量モデルで温度上昇をシミュレート。
- ・電流シャントとセンシングは負荷近くに配置し、共通インピーダンスによる測定誤差を回避。

EMC・センシング・制御への影響。

- 高 dV/dt スイッチングはスナバとスルーレート制御ドライバが必要で、IMU や通信線へのノイズ注入を回避。コモンモードチョークを用い、電力線と信号線を分離配線。
- 電流センサ（ホール素子またはシャント増幅器）は、故障検出に用いる短時間過渡を捉えられる十分な帯域を持つように配置。

実装：組込み過電流遮断ルーチン。マイクロコントローラは、ハードリアルタイムの保護動作を実装し、診断用にイベントをログ記録する。以下の組込み C ルーチンは安全な遮断パターンを示す。アナログ電流測定、意図的なデバウンス、パワー MOSFET ゲートへのハードウェア駆動を用いる。ルーチンは最小構成で、ハードウェアがフェイルセーフでデフォルトオフのゲートを持つことを前提とする。

コードサンプル 37 Simple overcurrent trip and latch for actuator power

```
cpp
#include <cstdint>
#include <atomic>

extern int  sampleADC();           // mA 単位で返す
extern void setGate(bool on);
extern void logEvent(const char* tag, int value);

namespace {
constexpr int TRIP_MA      = 35000;
constexpr int DEBOUNCE_MS = 5;
std::atomic<bool> latched{false};
} // namespace

void powerLoop()
{
    static int tcount = 0;

    if (latched.load(std::memory_order_acquire)) {
        return;           // 手動リセットまで保護状態を維持
    }

    const int i_ma = sampleADC();

    if (i_ma > TRIP_MA) {
        if (++tcount >= DEBOUNCE_MS) {
            setGate(false);           // ハードウェア切断
            latched.store(true, std::memory_order_release);
            logEvent("OVERCURRENT", i_ma);
        }
    }
}
```

```

    } else {
        tcount = 0;           // ノイズ期間が短ければリセット
    }
}

```

設計チェックリストと検証手順。

1. SOC および温度全域で実際の R_{int} を測定。
2. 式 (1) を用いて予想 I_{sc} を算出し、部品レベル I^2t 比較を実施。
3. プリチャージ時間およびコンタクタ定格を実機コンデンサ・母線測定で検証。
4. 最悪ケース通過エネルギーについて熱暴走シミュレーションを実施。
5. アクチュエータ動作下で EMC 免疫試験を行い、センサ整合性を確認。
6. ファームウェアにウォッチドッグと安全状態検証を実装。

工学への影響、トレードオフ、運用上のリスク。

- ・ヒューズを大型化すると誤動作トリップは減るが、通過エネルギーと導体損傷リスクが増大。
- ・ソリッドステートブレーカは制御柔軟性を追加するが、機械式装置と同等の短絡エネルギー定格が必要。
- ・冗長化は重量・コストを増やすが、人体近傍で動作する人型プラットフォームにとって単一点故障リスクを低減。
- ・コンデンサおよびバッテリーの残留エネルギーは保守時の潜在危険；目視可能な絶縁とロックアウト手順を含める。
- ・実使用条件下でのバッテリー内部抵抗の特性評価を怠ると、故障エネルギーを過小評価し火災リスクが増大。

実験的特性評価、解析サイジング、ハードウェアインザループ故障注入を組合せた保守的かつ計測に基づく設計プロセスを採用することで、人型ロボットにおける破壊的電気故障の確率を低減し、通常運転および保守時の双方で予測可能な安全な挙動を実現する。

10.4 電力システムの試験とモニタリング

先ほど議論した安全対策とバッテリー寿命戦略を基に、試験と継続的モニタリングは設計ルールを運用上の保証に変換する。検証は潜在故障を捕捉し、モニタリングはランタイム制限を強制し、人型ロボットでのグレースフルデグラデーションを支援する。

試験とモニタリングの目的

- ・代表的な歩容と操縦タスク下で、電力サブシステムがピーク電力、エネルギー、熱予算を満たすことを確認する。
- ・セル不均衡、高内部抵抗、コネクタ発熱、寄生ドレインを検出する。
- ・状態推定、予知保全、故障トリガ動作のための入力を提供する。

問題定義. 人型はアクチュエータに高く急速に変化する電流を供給しながら、電子機器と人間を保護しなければならない。試験装置とオンボードモニタは、過渡挙動、エネルギー容量、および充電状態 (SoC) と健康状態 (SoH) を確実に推定しなければならない。試験は、突然のトルク要求、回生ブレー

キ、長時間アイドルなど、フライトに似た擾乱を再現すべきである。

測定・推定すべき主要電氣量

- 各レールおよびセル毎の端子電圧 V_t 。
- 回生のための極性対応測定を伴う負荷電流 I 。
- セル、電力電子機器、コネクタの温度。
- 安定性マージン用の内部抵抗 R_{int} と過渡インダクタンス L 。
- SoH 用の累積エネルギーとクーロンスループット。

モニタリング用の簡易物理モデルセンサデータを融合するために、コンパクトで物理的に意味のあるモデルを用いる。端子電圧は等価回路モデルに従う：

$$[H]V_t = V_{\text{OC}}(\text{SoC}) - I R_{\text{int}} - L \frac{dI}{dt}, \quad (93)$$

ここで V_{OC} は SoC に対する開回路電圧である。クーロン計測は SoC 推定のベースラインを提供する：

$$[H]\text{SoC}(t) = \text{SoC}(0) - \frac{1}{C_{\text{nom}}} \int_0^t I(\tau) d\tau. \quad (94)$$

電圧補正と温度補償がドリフトを低減する。アクチュエータ電流が高度に動的な場合は、拡張カルマンフィルタ (EKF) または相補フィルタを実装する。

実験室試験プロトコル

1. 容量・サイクル特性評価

- アクチュエータ C レートに一致するレートで放電するプログラマブル電子負荷を使用する。
- 電圧、電流、温度、時刻を記録し、 $V_{\text{OC}}(\text{SoC})$ 曲線をフィットする。

2. パルス・過渡試験

- 歩行と突然の押力をシミュレートする電流パルスを印加する。
- ステップ応答から R_{int} を測定し、抵抗性と誘導性効果を分離する。

3. 熱・コネクタストレス

- 赤外線撮影で最悪デューティサイクルを実行し、熱ホットスポットを特定する。

4. 回生と EMC

- プリチャージ・回生回路を検証し、電圧オーバーシュートと制御電子機器への過渡結合を試験する。

5. セルバランシング検証

- 内蔵バランシングをアクティブにしてセルをサイクルし、不均衡減衰とバランシング電流を測定する。

インシチュモニタリングアーキテクチャ

- センサ：電流用高精度シャントまたはホール効果、セル毎電圧用 ADC、温度用 NTC/サーミスタアレイ。
- サンプリング戦略：
 - 過渡保護と安全のための高レート電流サンプリング (1–10 kHz)。
 - SoC・熱管理のための低レート電圧・温度サンプリング (1–10 Hz)。
- 処理：

- エッジマイクロコントローラまたはリアルタイムプロセッサが安全チェックと高速保護を実行する。
- 上位レベル計算機がデータを集約し、EKF を実行し、予知保全のためログを取る。

実用的推定器設計

- クーロン計測をモデル (93) による電圧ベース補正と組み合わせる。
- 温度依存の C_{nom} と V_{OC} ルックアップテーブルを用いる。
- センサドロップアウトと推定器発散を処理するためのウォッチドッグを実装する。

最小 Python モニタリングノード例

コードサンプル 38 Onboard battery monitor: simple coulomb count and alarm

```
#!/usr/bin/env python3
import time
import logging
from dataclasses import dataclass
from typing import Callable

# ログ設定
logging.basicConfig(level=logging.INFO, format="%(asctime)s_[(levelname)s]_(message)")
logger = logging.getLogger("bms")

# 定数
NOMINAL_CAPACITY_AH = 50.0          # 公称容量
DT_S = 0.1                          # 積分周期
VOLTAGE_MIN_V = 42.0                # 下限電圧
TEMP_MAX_C = 65.0                   # 上限温度
SOC_MIN = 0.05                      # 下限SoC
CURRENT_THRESHOLD_A = 0.5           # 低電流判定閾値
OCV_SLOPE = 0.4                     # OCV線形近似傾き
OCV_INTERCEPT_V = 3.6             # OCV線形近似切片
SOC_FILTER_GAIN = 0.1               # 電圧補正ゲイン

# ハードウェアI/F (本番ではドライバに置換)
def read_current() -> float: return 12.3 # A
def read_voltage() -> float: return 48.0 # V
def read_temp() -> float: return 35.0    # °C

@dataclass
class BmsState:
    soc: float
```

```

class BmsCore:
    def __init__(self, capacity_ah: float, initial_soc: float):
        self.capacity_c = capacity_ah * 3600.0 # クーロン変換
        self.state = BmsState(soc=max(0.0, min(1.0, initial_soc)))

    def update(self, i_a: float, v_v: float, dt_s: float) -> None:
        # クーロン計数
        self.state.soc -= (i_a * dt_s) / self.capacity_c
        self.state.soc = max(0.0, min(1.0, self.state.soc))

        # 低電流時のOCV補正
        if abs(i_a) < CURRENT_THRESHOLD_A:
            voc_per_cell = OCV_INTERCEPT_V + OCV_SLOPE * self.state.soc
            measured_soc = (v_v / 16.0 - OCV_INTERCEPT_V) / OCV_SLOPE
            # 16Sパック想定
            measured_soc = max(0.0, min(1.0, measured_soc))
            self.state.soc = (1.0 - SOC_FILTER_GAIN) * self.state.soc + SOC_FILTER_GAIN * measured_soc

    def check_alarm(self, v_v: float, t_c: float) -> bool:
        return v_v < VOLTAGE_MIN_V or t_c > TEMP_MAX_C or self.state.soc < SOC_MIN

def main() -> None:
    bms = BmsCore(NOMINAL_CAPACITY_AH, 0.9)
    while True:
        i = read_current()
        v = read_voltage()
        t = read_temp()
        bms.update(i, v, DT_S)
        if bms.check_alarm(v, t):
            logger.warning("ALARM_V=%0.2f_I=%0.2f_T=%0.1f_SoC=%0.3f", v, i, t, bms.state.soc)
            time.sleep(DT_S)

if __name__ == "__main__":
    main()

```

異常検知と予知保全

- R_{int} 推定値と容量劣化のローリング統計量を計算する。
- 抵抗または容量が設計閾値を超えたときにメンテナンスをトリガする。
- ログサイクルと温度履歴を用いて残存有効寿命を予測する。

設計トレードオフと運用上リスク

- 低電力コントローラにおけるサンプリング周波数 vs 消費電力。
- 推定器複雑性 vs 頑健性；EKF は正確なモデルとチューニングを要する。
- センサ配置はレイテンシと熱測定忠実度に影響する。
- EMI とグラウンドループは、対策しないと誤警報を引き起こす。

技術的影響

- 信頼できる SoC・SoH のために高分解能電流感測とセルレベル電圧モニタリングに投資する。
- アクチュエータストール下での破局的故障を防ぐため、決定論的タイミングの保護ループを設計する。
- 故障後の根本原因解析を可能にするため、ログ帯域とセキュアテレメトリを計画する。
- 過度に保守的な閾値はミッション可用性を低下させ、保守的でない閾値は熱暴走リスクを増大させることを認識する。

11 センサ統合

11.1 センサの種類（ビジョン、IMU など）

信頼性の高い電源配給と監視を確保した後、センサ選択では利用可能な電力、配線トポロジ、および電磁干渉の制約を考慮しなければならない。センサのセットとその配置は、ヒューマノイドロボットにとって知覚の忠実度、制御帯域、および機械的なトレードオフを決定する。

センサの種類と運用上の役割：

- ビジョンシステム：単眼 RGB、ステレオリグ、深度カメラ（構造化光、ToF）、LiDAR、イベントカメラ。視覚センサはシーン理解と障害物検出を提供する。RGB カメラでは典型的な帯域は 30～120 fps、深度センサは一般的に 30 Hz で動作する。頭部には状況認識用の広視野光学系を、マニピュレータには精密操作用の焦点距離の短いカメラを用いる。
- 慣性計測装置（IMU）：3 軸加速度計とジャイロ、しばしば磁気計と組み合わせる。IMU は高レート姿勢と短期速度手がかりを提供し、バランスと歩容制御に不可欠である。高性能 IMU は $f_s \geq 200$ Hz でサンプリングし、バイアス不整定性が低い。
- 関節位置・速度センサ：アクチュエータに搭載された絶対エンコーダ、増分エンコーダ、磁気エンコーダ。これらは低レベルサーボ制御の主要なフィードバックであり、EtherCAT 経由で最小遅延と決定論的更新レートを提供しなければならない。
- 力・トルクセンサと触覚アレイ：手首と足首の 6 軸 F/T センサ、足底と指先の分散型タクセルアレイ。これらは安定歩行と操作の接触状態とインピーダンス制御手がかりを提供する。
- 近接・測距センサ：本体近傍の衝突回避に適した超音波、赤外、短距離 ToF モジュール。LiDAR ユニットは長距離測距と角度スキャンをマッピング用に供給する。
- 音響センサ：音声コマンドと音響シーン解析用のマイクアレイ。HRI と話者定位に有用。
- 環境センサ：温度、湿度、気圧、GPS（屋外ヒューマノイド用）。屋内では GPS は限定的だが、屋外タスクでは依然として関連する。

- ・電流・電圧センサ：間接トルク推定と過電流時の安全シャットダウンのためのモータ電流を測定する。

評価すべき性能パラメータ：

1. サンプリングレートと遅延。制御ループは決定論的タイミングを要求する；IMU とエンコーダは通常 > 200 Hz で供給し、ビジョンパイプラインは予測フィルタと組み合わせれば高遅延を許容できる。
2. ダイナミックレンジと感度。加速度計は衝撃時の飽和を避けるため適切なレンジが必要；カメラは変化する照明に対して露出制御が必要。
3. ノイズ特性：IMU のバイアス不整合性・ランダムウォーク、カメラの SNR。これらのパラメータはセンサ融合のフィルタチューニングを決定する。
4. インタフェースと同期。アクチュエータと高レートセンサには決定論的バス（例：EtherCAT、CAN FD）を用いる。ハードウェア同期（トリガ線または PTP）がマルチセンサタイムスタンプライメントに必要。

数理的概略：姿勢推定のための IMU と関節計測の融合は、クォータニオン運動学と加速度計または磁気計からの補正を用いる。角速度 ω からの連続クォータニオン伝播は

$$[H]\dot{\mathbf{q}}(t) = \frac{1}{2} \mathbf{q}(t) \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega}(t) \end{bmatrix}, \quad (95)$$

ここで \otimes はクォータニオン乗算を表す。単軸離散相補フィルタは

$$\theta_k = \alpha(\theta_{k-1} + \omega_k \Delta t) + (1 - \alpha) \theta_{\text{acc},k},$$

と書け、 $\alpha \in [0, 1]$ はジャイロ積分と加速度計傾斜推定の重み付けである。

実装上の統合と同期：

- ・タイムスタンプ：センササンプリング瞬間に単調ハードウェアタイムスタンプを割り当てる。ソフトウェアタイムスタンプはジッタを加え融合を複雑にする。
- ・トリガリング：ビジョン慣性オドメトリ（VIO）実行時はカメラと IMU に共通トリガを用いる。回転 LiDAR ではエンコーダ角度をタイムスタンプ付きスキャンに同期する。
- ・帯域予算：各センサにバス帯域と CPU/GPU 時間を割り当てる。例：30 Hz の 4K カメラは秒間数ギガビットを消費する；ローカル処理が無い場合は圧縮またはダウンサンプリングする。

実装ノート：以下の ROS2 Python スニペットは、基本的なバイアス除去とタイムスタンプを備えた簡単な IMU リーダ兼パブリッシャループを示す。インラインコメントは簡潔で実用的である。

コードサンプル 39 Simple ROS 2 IMU publisher with bias removal

```
import rclpy
from rclpy.node import Node
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
from sensor_msgs.msg import Imu
import spidev
```

```

import struct
import time
from typing import List, Tuple

class ImuPublisher(Node):
    def __init__(self) -> None:
        super().__init__('imu_publisher')

    # パラメータ宣言
    self.declare_parameter('frame_id', 'imu_link',
                           ParameterDescriptor(type=ParameterType.PARAMETER_STRING))
    self.declare_parameter('bus', 0,
                           ParameterDescriptor(type=ParameterType.PARAMETER_INTEGER))
    self.declare_parameter('device', 0,
                           ParameterDescriptor(type=ParameterType.PARAMETER_INTEGER))
    self.declare_parameter('max_speed_hz', 1_000_000,
                           ParameterDescriptor(type=ParameterType.PARAMETER_INTEGER))
    self.declare_parameter('timer_period_sec', 0.005,
                           ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE))
    self.declare_parameter('bias_gyro', [0.0, 0.0, 0.0],
                           ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE))

    # パラメータ取得
    frame_id = self.get_parameter('frame_id').value
    bus = self.get_parameter('bus').value
    device = self.get_parameter('device').value
    max_speed_hz = self.get_parameter('max_speed_hz').value
    timer_period = self.get_parameter('timer_period_sec').value
    self.bias_gyro: List[float] = self.get_parameter('bias_gyro').value

    # Publisher
    self.pub = self.create_publisher(Imu, 'imu/data_raw', 10)

    # SPI初期化
    try:
        self.spi = spidev.SpiDev()
        self.spi.open(bus, device)
        self.spi.max_speed_hz = max_speed_hz
        self.spi.mode = 0b11
        self.get_logger().info(f'SPI初期化完了 bus={bus} device={device}')

```

```

except Exception as e:
    self.get_logger().error(f'SPI初期化失敗:{e}')
    raise

# タイマー
self.timer = self.create_timer(timer_period, self.timer_callback)

# メッセージ雛形
self.imu_msg = Imu()
self.imu_msg.header.frame_id = frame_id

def read_imu(self) -> Tuple[List[float], List[float]]:
    # レジスタアドレスとダミー読出し（センサ依存）
    try:
        raw = self.spi.xfer2([0x3B, 0x00] + [0x00] * 12)
        data = bytes(raw[2:])
        ax, ay, az, gx, gy, gz = struct.unpack('>6h', data)
        scale_acc = 1e-3
        scale_gyro = 1e-3
        return ([a * scale_acc for a in (ax, ay, az)],
                [g * scale_gyro for g in (gx, gy, gz)])
    except Exception as e:
        self.get_logger().warn(f'SPI通信エラー:{e}')
        return ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0])

def timer_callback(self) -> None:
    acc, gyro = self.read_imu()
    gyro = [g - b for g, b in zip(gyro, self.bias_gyro)]

    self.imu_msg.header.stamp = self.get_clock().now().to_msg()
    self.imu_msg.linear_acceleration.x = acc[0]
    self.imu_msg.linear_acceleration.y = acc[1]
    self.imu_msg.linear_acceleration.z = acc[2]
    self.imu_msg.angular_velocity.x = gyro[0]
    self.imu_msg.angular_velocity.y = gyro[1]
    self.imu_msg.angular_velocity.z = gyro[2]
    self.pub.publish(self.imu_msg)

def destroy_node(self) -> None:
    self.spi.close()

```

```

        super().destroy_node()

def main(args=None):
    rclpy.init(args=args)
    node = ImuPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

設計上のトレードオフと運用上のリスク：

- ・冗長性対重量・電力。センサの重複は堅牢性を向上させるが、質量と消費電力を増加させる。
- ・遅延対精度。高解像度ビジョンは知覚精度を高めるが遅延を加える；制御ループで IMU 予測を補完する。
- ・EMI とケーブリング。モータ近くのセンサ配置は EMI を増加；ケーブルを配線し差動信号を用いる。
- ・故障モード。単一点センサ故障が安全でない動作を引き起してはならない；残存センサを用いた縮退モードコントローラを実装する。

エンジニアはセンシング忠実度、計算、電力、安全性をバランスさせなければならない。適切なインタフェース選択、時間同期、故障処理は、信頼性の高いヒューマノイド動作にとって重要である。

11.2 センサキャリブレーション技術

前小節では、カメラ、IMU、力・トルクセンサ、関節エンコーダなど、ヒューマノイドに通常搭載されるセンサの種類を概観した。センサキャリブレーションは、これらの記述を実行可能なモデルおよび変換に翻訳し、知覚および制御が一貫した世界座標およびロボット座標で動作できるようにする。

問題定義と工学的的重要性。ヒューマノイドロボットは、バランス、歩行、操縦、相互作用のために、空間的および時間的に一貫したセンサ出力を必要とする。キャリブレーション誤ったセンサは、状態推定のドリフト、誤った足の配置、安全でない相互作用力を引き起こす。キャリブレーションのタスクは、以下を推定することである：

- ・内在パラメータ（例：カメラ焦点距離、IMU ノイズ特性）、
- ・センサとロボット本体間の外変換（例：カメラ-to-IMU、IMU-to-ベース）、
- ・時間オフセット（センサ遅延）および各センサのバイアス（例：加速度計バイアス、エンコーダゼロ）。

可観測性と実験設計。成功したキャリブレーションには、各パラメータを可観測にする励起が必要である。重要な原則：

- 外変換の可観測性のために、少なくとも 3 自由度で姿勢を変化させる。
- カメラ内部パラメータのために、高コントラストのキャリブレーションターゲット（チェッカーボードまたは AprilTag）を使用する。
- IMU バイアスおよびスケール推定のために、静止保持と動的挙動を組み合わせる。
- カトルクセンサを励起し、線形性を観測するために、既知の力または重りを印加する。
- 同期されたデータストリームを記録し、可能な場合はハードウェアクロックでタイムスタンプを記録する。

数学的定式化。キャリブレーションを、内在パラメータ、外変換、バイアス、時間オフセットを集めたパラメータベクトル x に対する非線形最小二乗問題として定式化する。各モダリティから残差を定義し、共分散の逆数で重み付けする。統一目的関数は

$$[H]x^* = \arg \min_x \sum_{k \in \mathcal{K}} w_k \|r_k(x)\|^2, \quad (96)$$

であり、ここで r_k はモダリティ固有の残差、 w_k はチューニング重みである。典型的な残差：

1. カメラ再投影：画像点 u_{ij} 、カメラ内部パラメータ K 、本体からカメラへの変換 T_{cb} 、3D ランドマーク X_j に対して $r_{\text{cam}} = u_{ij} - \pi(K, T_{cb}, X_j)$ 。
2. IMU 運動学：事前積分測定を用いて、順運動学から予測された姿勢と IMU 姿勢を比較し、 \mathbb{R}^3 で $r_{\text{imu}} \approx \text{Log}(R_{\text{pred}}^T R_{\text{imu}})$ を生成。
3. エンコーダオフセット：測定関節角 θ_m と運動学的に予測された角からオフセット $\delta\theta$ を引いた値との線形残差。
4. カトルク：測定レンチと、接触力を与えた動的モデルによって予測されたレンチとの残差。

線形化と数値解法。(96) を解くためにガウス-ニュートンまたはレーベンバーグ・マルカートを使用する。SE(3) 上の変換には、リー代数 $\mathfrak{se}(3)$ を介して摂動を表現し、指数写像で更新する。反復 n の線形化更新：

$$J_n \Delta x = -r(x_n), \quad x_{n+1} = \exp(\Delta x) \circ x_n,$$

ここで J_n は積み上げヤコビアンである。適切な共分散重み付けは、マルチセンサ結合を安定化し、支配的モダリティが他をバイアスすることを防ぐ。

実用的パイプラインと手順。堅牢なヒューマノイドキャリブレーションパイプラインは以下に従う：

1. 同期データセットを収集：マルチビューチェッカーボード画像、IMU 静止区間、動的振動、ワークスペースを網羅する関節軌道。
2. 前処理：特徴検出、外れ値除去、IMU ノイズの Allan 解析のための静止区間セグメンテーション。
3. パラメータ初期化：標準的なカメラキャリブレーションで K を、ハンド・アイ手法（例：Tsai-Lenz）でおおよその外変換を、静止平均で IMU バイアスを求める。
4. 合同最適化：スケールおよびバイアスに適切な事前分布を用いて、結合コスト (96) を最小化。
5. 検証：未見姿勢で交差検証；残差、再投影 RMS、IMU ドリフト、歩行・把持などの閉ループ挙動を確認。

実装例。以下の Python スニペットは、カメラ再投影および IMU 姿勢残差の最小残差スタックを

`scipy.optimize.least_squares` で解く例を示す。この例は、実用的な構造を重視し、製品レベルの堅牢性は追求しない。

コードサンプル 40 最小IMU-カメラ合同キャリブレーション残差（例示的）。

```
import numpy as np
from scipy.optimize import least_squares
from typing import Dict, List, Any

def quat_rotate(q: np.ndarray, v: np.ndarray) -> np.ndarray:
    """クォータニオン $q$ でベクトル $v$ を回転"""
    q = q / np.linalg.norm(q)
    w, x, y, z = q
    vx, vy, vz = v
    return np.array([
        (1 - 2*y*y - 2*z*z)*vx + 2*(x*y - w*z)*vy + 2*(x*z + w*y)*vz,
        2*(x*y + w*z)*vx + (1 - 2*x*x - 2*z*z)*vy + 2*(y*z - w*x)*vz,
        2*(x*z - w*y)*vx + 2*(y*z + w*x)*vy + (1 - 2*x*x - 2*y*y)*vz
    ])

def logSO3(R: np.ndarray) -> np.ndarray:
    """SO(3)行列の対数マップ（3次元軸角）"""
    tr = np.clip((np.trace(R) - 1) / 2, -1, 1)
    theta = np.arccos(tr)
    if np.abs(theta) < 1e-6:
        return np.zeros(3)
    return theta / (2*np.sin(theta)) * np.array([
        R[2,1] - R[1,2],
        R[0,2] - R[2,0],
        R[1,0] - R[0,1]
    ])

def reprojection_residuals(x: np.ndarray, observations: Dict[str, List[Any]]) -> np.ndarray:
    """
    再投影・IMU回転残差を計算
    x: [fx, fy, cx, cy, tx, ty, tz, qx, qy, qz, qw]
    """
    K = np.array([[x[0], 0, x[2]], [0, x[1], x[3]], [0, 0, 1]])
    t_cam = x[4:7]
    q_cam = x[7:11]
```

```

res: List[float] = []

# 画像再投影残差
for obs in observations.get('images', []):
    X_body = np.asarray(obs['X_body'])
    u_meas = np.asarray(obs['u'])
    X_cam = quat_rotate(q_cam, X_body - t_cam)
    if X_cam[2] <= 0:
        res.extend([1e3, 1e3])
        continue
    u_pred = (K @ (X_cam / X_cam[2]))[:2]
    res.extend(u_meas - u_pred)

# IMU 回転残差
for imu_obs in observations.get('imu', []):
    R_pred = np.asarray(imu_obs['R_pred'])
    R_imu = np.asarray(imu_obs['R_imu'])
    res.extend(logSO3(R_pred.T @ R_imu))

return np.array(res)

# 初期推定値（例：ID カメラ、零並進、単位クォータニオン）
x0 = np.array([500., 500., 320., 240., 0., 0., 0., 0., 0., 1.])

# 観測データ（外部で用意）
# observations = {'images': [...], 'imu': [...]}

sol = least_squares(
    reprojection_residuals,
    x0,
    args=(observations,),
    method='trf',
    loss='huber',
    verbose=2
)
# sol.x が推定パラメータ

```

運用上の考慮事項とチェック。以下の実用的チェックを適用する：

- タイムスタンプアライメントを確保する；ミリ秒単位のオフセットでも高速ヒューマノイド動作では誤差を生じる。

- IMU Allan 分散を定期的に再計算し、センサ劣化を検出する。
- 外れ値またはモデル不整合を示す非ガウスタイルの残差分布を監視する。
- 機械的メンテナンスまたはエンコーダ交換後は外変換を再キャリブレーションする。

設計上のトレードオフとリスク。複数モダリティを合同でキャリブレーションすると系統誤差は減少するが、複雑さが増し、局所最小に陥るリスクが高まる。内部パラメータを別々にキャリブレーションした後、外変換を行うと最適化は簡単になるが、結合バイアスが残る可能性がある。単一データセットに過適合すると、新しい地形やタスクへの転移性が低下する。安全上重要なヒューマノイド挙動には、安全上重要なセンサ（IMU、力・トルク）に対して保守的な重み付けが賢明である。定期的なキャリブレーションスケジュール、自動検証ルーチン、保守的フォールバック挙動が運用上のリスクを軽減する。

11.3 データ取得と処理

キャリブレーションとセンサモダリティの棚卸しの後、取得パイプラインは異種のタイムスタンプ付き信号をバランス、歩行、操作の整合された状態推定値に変換しなければならない。以下の設計は、同期、バッファリング、および融合を、リアルタイムデッドラインとハードウェア制約によって制限された結合した工学問題として扱う。

問題定義. ヒューマノイドは高レート IMU、中レート関節エンコーダおよび力・トルクセンサ、低レートカメラからデータを取り込まなければならない。各センサは異なるサンプルレート、レイテンシ、ノイズを持つ。目標は制御ループ周波数で融合され時間整合された状態を、境界付きレイテンシと予測可能なジッタで生成することである。

技術分析.

- 時刻同期: ハードウェアタイムスタンプが利用可能な場合はそれを使用する。Precision Time Protocol (PTP) または Pulse-Per-Second (PPS) 信号はセンサ間スキューをマイクロ秒単位に削減する。ソフトウェアのみのタイムスタンプは可変レイテンシを導入する。
- バッファリングとリサンプリング: ハードウェアタイムスタンプでインデックスされたセンサごとのリングバッファを維持する。要求出力時刻 t^* に対し、慣性およびエンコーダサンプルを線形補間でリサンプルする:

$$[H]x(t^*) = x_k + \frac{t^* - t_k}{t_{k+1} - t_k} (x_{k+1} - x_k). \quad (97)$$

これは因果性を保ちながら統一エポックでの融合を可能にする。

- レートミスマッチとスループット: 最悪ケーススループットを計算しバスとプロセッサを供給する。例えば、30 fps、640×480、3 チャンネルのステレオ RGB カメラは

$$[H]B = 30 \cdot 640 \cdot 480 \cdot 3 \approx 27.6 \text{ MB/s} \quad (98)$$

非圧縮で、ステレオペアあたりとなる。圧縮と選択的 ROI は負荷を削減する。

- 融合アルゴリズムの選択: ヒューマノイド状態推定には、拡張カルマンフィルタ (EKF) または反復

カルマン変種が動力学モデリングとセンサ非線形性のバランスを取る。離散時間カルマン更新:

$$\begin{aligned}
 \hat{x}_{k|k-1} &= f(\hat{x}_{k-1|k-1}, u_{k-1}) \\
 P_{k|k-1} &= F P_{k-1|k-1} F^\top + Q \\
 [H] \quad K_k &= P_{k|k-1} H^\top (H P_{k|k-1} H^\top + R)^{-1} \\
 \hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k (y_k - h(\hat{x}_{k|k-1})) \\
 P_{k|k} &= (I - K_k H) P_{k|k-1}
 \end{aligned} \tag{99}$$

ここで f, h は離散プロセスおよび観測写像、 F, H はそれらの線形化、 Q, R はプロセスおよび観測共分散である。

実装パターン. 取得スタックを3層に設計する:

1. ハードウェアタイムスタンプと DMA を持つデバイスドライバ。
2. タイムスタンプ、バッファリング、決定的リサンプリングを行うリアルタイム取得ノード。
3. 固定制御周波数で EKF を実行する融合ノード。

実践的アルゴリズム手順:

1. ハードウェアタイムスタンプ付き生サンプルを取得しリングバッファにプッシュする。
2. 制御エポック t^* で最近傍サンプルをプルし、(97) を用いてリサンプルする。
3. マハラノビスゲーティングで外れ値を拒否: $(y - h(\hat{x}))^\top S^{-1} (y - h(\hat{x})) > \gamma$ なら y を拒否。
4. リサンプル観測を用いて EKF 予測と更新を実行する。
5. 融合状態をレイテンシ予算メタデータと共に公開する。

コード例: Python での非同期取得とリサンプリング。以下に IMU およびエンコーダデータ向けの最小限の本番適用パターンを示す。コメントは実世界の懸念（タイムスタンプ、ロック、境界付きバッファサイズ）を説明する。

コードサンプル 41 非同期取得、リサンプリング、融合スケルトン.

```

import asyncio
import numpy as np
from collections import deque
from typing import Tuple, Optional, Callable
import logging
import time

# ロギング設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# センサデータ型定義
IMUData = np.ndarray # shape=(6,) [acc, gyro]
```

```

JointData = np.ndarray # shape=(n_joints,)

class SensorBuffer:
    """タイムスタンプ付きセンサデータのリングバッファ"""
    def __init__(self, maxlen: int = 1024):
        self._buf: deque[Tuple[float, np.ndarray]] = deque(maxlen=maxlen)
        self._lock = asyncio.Lock()

    async def append(self, ts: float, data: np.ndarray):
        async with self._lock:
            self._buf.append((ts, data.copy()))

    async def resample(self, t_star: float) -> Optional[np.ndarray]:
        """線形補間によるリサンプリング"""
        async with self._lock:
            buf_list = list(self._buf)

            if len(buf_list) < 2:
                return None

            for i in range(len(buf_list) - 1):
                t0, x0 = buf_list[i]
                t1, x1 = buf_list[i + 1]
                if t0 <= t_star <= t1:
                    alpha = (t_star - t0) / (t1 - t0)
                    return x0 + alpha * (x1 - x0)
            return None

class StateEstimator:
    """センサ融合状態推定器"""
    def __init__(self, n_joints: int, control_rate: float):
        self.imu_buf = SensorBuffer()
        self.enc_buf = SensorBuffer()
        self.n_joints = n_joints
        self.dt = 1.0 / control_rate
        self._state_pub: Optional[Callable] = None

    def register_publisher(self, pub_func: Callable):
        self._state_pub = pub_func

```

```

async def imu_callback(self, ts: float, acc_gyro: IMUData):
    await self.imu_buf.append(ts, acc_gyro)

async def enc_callback(self, ts: float, joint_angles: JointData):
    await self.enc_buf.append(ts, joint_angles)

async def fusion_step(self, t_star: float) -> Optional[np.ndarray]:
    """1ステップのセンサ融合"""
    imu_sample = await self.imu_buf.resample(t_star)
    enc_sample = await self.enc_buf.resample(t_star)

    if imu_sample is None or enc_sample is None:
        logger.warning(f"データ欠損 at t={t_star:.3f}")
        return None

    # 相補フィルタによる姿勢推定
    fused = self._complementary_filter(imu_sample, enc_sample)
    return fused

def _complementary_filter(self, imu: IMUData, enc: JointData) -> np.ndarray:
    """簡易相補フィルタ: ジャイロ積分 + エンコーダドリフト補正"""
    # 実装例: エンコーダ角度とIMU角速度を融合
    alpha = 0.98 # 補完係数
    gyro_angle = enc + imu[3:6] * self.dt # ジャイロ積分
    fused_angle = alpha * gyro_angle + (1 - alpha) * enc
    return np.hstack([fused_angle, imu[:3]]) # [角度, 加速度]

async def fusion_loop(estimator: StateEstimator, control_rate: float):
    """非同期センサ融合ループ"""
    dt = 1.0 / control_rate
    t_next = time.time()

    while True:
        t_next += dt
        t_star = t_next

        try:
            state = await estimator.fusion_step(t_star)
            if state is not None and estimator._state_pub:
                latency = time.time() - t_star

```

```

        estimator._state_pub(state, latency)
    except Exception as e:
        logger.error(f"融合エラー:{e}")

    sleep_time = max(0, t_next - time.time())
    await asyncio.sleep(sleep_time)

# 使用例
async def main():
    estimator = StateEstimator(n_joints=7, control_rate=100)

    # パブリッシャ登録
    def publish_state(state: np.ndarray, latency: float):
        logger.info(f"公開状態:{state[:3]} (遅延:{latency*1000:.1f}ms)")

    estimator.register_publisher(publish_state)

    # シミュレーションタスク
    async def simulate_sensors():
        t0 = time.time()
        while True:
            t = time.time() - t0
            await estimator.imu_callback(t, np.random.randn(6) * 0.1)
            await estimator.enc_callback(t, np.random.randn(7) * 0.05)
            await asyncio.sleep(0.001)

    # 並行実行
    await asyncio.gather(
        fusion_loop(estimator, 100),
        simulate_sensors()
    )

if __name__ == "__main__":
    asyncio.run(main())

```

運用上の懸念とトレードオフ.

- レイテンシ対忠実度: 高レート融合は推定器ラグを削減するが CPU 負荷とジッタを増加させる。
DMA または FPGA に前置フィルタをオフロードして最悪ケース CPU を制限する。
- 決定性: 制御安定性が安全上重要な場合はリアルタイム OS を使用してスケジューリングジッタを境界付ける。

- 通信バス: センサ配置はケーブル長と EMI に影響する。シールドと差動リンクを使用する (例: CAN-FD, LVDS, GigE Vision)。
- 故障モード: ハードウェアタイムスタンプ喪失、バッファオーバーラン、センサ飽和は安全な劣化をトリガしなければならない、例: 歩行凍結またはコンプライアント制御へ切替。

工学上の影響、トレードオフ、リスク:

- 正確なハードウェアタイムスタンプは整列を劇的に簡素化するが、互換ドライバとネットワーク時刻インフラを必要とする。
- 融合複雑度の選択は推定器精度と計算決定性の間のトレードオフである。
- 帯域幅または CPU の供給不足はフレームドロップまたは遅延状態を引き起こし、転倒または安全でない相互作用をリスクとする。
- グレースフルデグラデーションを設計する: 各センサモダリティに対してフォールバックコントローラ、タイムアウトポリシー、故障検出閾値を指定する。

11.4 実世界のケーススタディ: 障害物検出

前述のキャリブレーションとタイムスタンプ戦略は、センサ間およびフレーム間の信頼性の高い融合を直接可能にする。整合した外部パラメータ、同期されたタイムベース、既知の遅延境界は、以下に記述する障害物検出パイプラインの前提条件である。

問題定義: 胴体搭載型ヒューマノイドが歩行および操作タスク中に静的・動的障害物を検出できるようにする。制約としては、限られたペイロード、四肢による断続的な遮蔽、エゴモーションバイアスを引き起こすプラットフォーム運動、オンボード組み込みプロセッサによって制限される計算予算がある。システムは、運動スタックに対して遅延が制限された障害物位置および速度推定値を提供しなければならない。

技術分析

• センサと配置:

1. 胴体への 3D ライダ (ソリッドステートまたはマルチビーム) は、スパースな障害物に対する長距離・高精度なレンジリターンを提供する。
2. 頭部への ToF 深度カメラまたはステレオペアは、小さな障害物および落下物に対するより高密度な近距離・中距離深度マップを提供する。
3. 胴体への IMU および脚部エンコーダは、歩行誘起運動を補償するためのエゴモーション推定値を提供する。

• 同期と変換:

- すべての測定値は、キャリブレーション済み外部パラメータと同期されたタイムベースを用いてロボットベースフレームに変換される。ハードウェア同期または遅延ジッタが制限されたソフトウェアレベルメッセージスタンプを使用する。

• 前処理:

1. タイムスタンプアライメントとモーション補償: IMU / 脚オドメトリをセンサタイムスタンプ間で積分することによりエゴモーションを除去する。時刻 t_i の点群 p_i に対して、推定ロボットポーズ $T(t_i \rightarrow t_{\text{ref}})$ により変換する。

2. ボクセルグリッドダウンサンプリングは、密度を減らしながら障害物ジオメトリを保持する。最小障害物サイズに基づいてボクセルサイズを選択（例：0.02–0.05 m）。
3. RANSAC を用いた地面プレーン除去は、床面リターンによる誤検出を減らし、低い障害物を保持する。

• 占有表現：

- 歩行決定には 2D 確率的占有グリッドを、操作用および足跡計画には 3D ボクセル占有を使用する。
- 占有セルは、数値的に安定なベイズ更新のため対数オッズを用いて更新する。時刻 t の増分更新は：

$$[H]l_t(c) = l_{t-1}(c) + \log \frac{P(z_t(c) | \text{occ})}{1 - P(z_t(c) | \text{occ})} - l_0, \quad (100)$$

ここで、 l は対数オッズ、 c はセルをインデックス、 $z_t(c)$ はセンサ観測、 l_0 は事前対数オッズである。

• 動的障害物追跡：

- ダウンサンプル点群でユークリッドクラスタリングを用いて点を物体仮説にクラスタリングする。
- 各クラスタに対して平面座標 (x, y, \dot{x}, \dot{y}) でカルマンフィルタを用いて仮説を追跡する。検出が頻繁な場合は定速度線形モデルを使用する。
- 観測モデルはクラスタ重心観測を状態にマッピングし、共分散はセンサ固有のレンジノイズを考慮しなければならない。

• 軽減すべき故障モード：

1. 深度カメラは太陽光および反射面で性能が低下する。
2. ライダは薄い障害物に対する解像度が限られ、四肢によって遮蔽されることがある。
3. 変換更新の遅延は位置ラグを生じ、高速移動障害物の誤分類を引き起こす。

実装（実用的なコードパターン）

- 以下のリストは、ライダと深度を購読し、ボクセルフィルタを適用し、Open3D で RANSAC 地面除去を行い、2D 占有グリッドを更新するコアパイプラインを実装する ROS スタイル Python ノードのスケッチである。コメントは、キャリブレーション済み変換と時刻補償を挿入する場所を示す。

コードサンプル 42 ライダと深度カメラを組み合わせた最小限の障害物検出パイプライン

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import PointCloud2, PointField
from nav_msgs.msg import OccupancyGrid, MapMetaData
from geometry_msgs.msg import Pose
import numpy as np
```

```

import open3d as o3d
from sensor_msgs_py import point_cloud2 as pc2
import tf2_ros
from tf2_ros import TransformException
from tf_transformations import quaternion_matrix

# QoS設定：LiDAR/DepthはBestEffortで低遅延
QOS_BEST = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=5
)

class ObstacleNode(Node):
    def __init__(self):
        super().__init__('obstacle_node')
        # パラメータ宣言と取得
        self.declare_parameter('voxel_size', 0.03)
        self.declare_parameter('grid_res', 0.05)
        self.declare_parameter('grid_size', 200)
        self.voxel_size = self.get_parameter('voxel_size').value
        self.grid_res = self.get_parameter('grid_res').value
        self.grid_size = self.get_parameter('grid_size').value

        # 購読・配信
        self.sub_lidar = self.create_subscription(
            PointCloud2, '/lidar', self.cb_lidar, QOS_BEST)
        self.sub_depth = self.create_subscription(
            PointCloud2, '/depth', self.cb_depth, QOS_BEST)
        self.pub_grid = self.create_publisher(
            OccupancyGrid, '/obstacle_grid', 10)

        # TFバッファ・リスナ
        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        # 内部地図：log-odds
        self.log_odds = np.zeros((self.grid_size, self.grid_size), dtype=np.float32)
        self.l_occ = 0.9
        self.l_free = -0.7

```

```

# タイマーで定期的に地図配信
self.create_timer(0.1, self.publish_grid)

def pc2_to_xyz(self, msg: PointCloud2) -> np.ndarray:
    # generator→xyz配列へ
    gen = pc2.read_points(msg, field_names=('x', 'y', 'z'), skip_nans=True)
    return np.array(list(gen))

def transform_points(self, pts: np.ndarray, frame_id: str, stamp) -> np.ndarray:
    # base_linkへ変換
    try:
        trans = self.tf_buffer.lookup_transform(
            'base_link', frame_id, stamp, timeout=rclpy.duration.Duration(seconds=
except TransformException:
        return pts # 失敗時は変換なし
    q = [trans.transform.rotation.x, trans.transform.rotation.y,
          trans.transform.rotation.z, trans.transform.rotation.w]
    T = quaternion_matrix(q)
    T[0, 3] = trans.transform.translation.x
    T[1, 3] = trans.transform.translation.y
    T[2, 3] = trans.transform.translation.z
    pts_h = np.hstack([pts, np.ones((pts.shape[0], 1))])
    return (T @ pts_h.T)[:3, :].T

def cb_lidar(self, msg: PointCloud2):
    pts = self.pc2_to_xyz(msg)
    pts = self.transform_points(pts, msg.header.frame_id, msg.header.stamp)
    pcd = o3d.geometry.PointCloud(o3d.utility.Vector3dVector(pts))
    pcd = pcd.voxel_down_sample(self.voxel_size)
    plane_model, inliers = pcd.segment_plane(
        distance_threshold=0.02, ransac_n=3, num_iterations=50)
    non_ground = pcd.select_by_index(inliers, invert=True)
    self.update_occupancy(non_ground)

def cb_depth(self, msg: PointCloud2):
    pts = self.pc2_to_xyz(msg)
    pts = self.transform_points(pts, msg.header.frame_id, msg.header.stamp)
    pcd = o3d.geometry.PointCloud(o3d.utility.Vector3dVector(pts))
    pcd = pcd.voxel_down_sample(self.voxel_size / 2)

```

```

pcd, _ = pcd.remove_statistical_outlier(nb_neighbors=20, std_ratio=2.0)
self.update_occupancy(pcd)

def update_occupancy(self, pcd: o3d.geometry.PointCloud):
    pts = np.asarray(pcd.points)
    # 2Dグリッド座標へ
    gx = ((pts[:, 0] + self.grid_size * self.grid_res / 2) / self.grid_res).astype(int)
    gy = ((pts[:, 1] + self.grid_size * self.grid_res / 2) / self.grid_res).astype(int)
    mask = (0 <= gx) & (gx < self.grid_size) & (0 <= gy) & (gy < self.grid_size)
    gx, gy = gx[mask], gy[mask]
    # 占有更新
    self.log_odds[gy, gx] += self.l_occ
    # レイキャスティングでfree空間（簡易版）
    for (x, y) in zip(gx, gy):
        self.ray_cast_free(0, 0, x, y)

def ray_cast_free(self, x0, y0, x1, y1):
    # Bresenhamでfree空間を減算
    dx, dy = abs(x1 - x0), abs(y1 - y0)
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    err = dx - dy
    while True:
        if 0 <= x0 < self.grid_size and 0 <= y0 < self.grid_size:
            self.log_odds[y0, x0] += self.l_free
        if x0 == x1 and y0 == y1:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy

def publish_grid(self):
    grid = OccupancyGrid()
    grid.header.stamp = self.get_clock().now().to_msg()
    grid.header.frame_id = 'base_link'
    grid.info = MapMetaData()

```

```

grid.info.resolution = self.grid_res
grid.info.width = self.grid_size
grid.info.height = self.grid_size
grid.info.origin = Pose()
grid.info.origin.position.x = -self.grid_size * self.grid_res / 2
grid.info.origin.position.y = -self.grid_size * self.grid_res / 2
# 確率へ変換
prob = 1 - 1 / (1 + np.exp(self.log_odds))
grid.data = (prob * 100).clip(0, 100).astype(np.int8).ravel().tolist()
self.pub_grid.publish(grid)

```

```

def main(args=None):
    rclpy.init(args=args)
    node = ObstacleNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

```

性能指標とチューニング

- 遅延予算：通常のヒューマノイド歩行で知覚から制御までの遅延を 200 ms 未満に保ち、センサ融合サイクルは 100 ms 未満を目指す。
- 偽陰性リスクは、ボクセルサイズが最小の重要障害物寸法の半分を超えると増加する。
- 追跡ロスは、更新周波数が障害物通過時間を下回ると発生するため、観測された変動性にフィルタプロセスノイズを合わせる。

エンジニアリングへの影響、トレードオフ、運用上のリスク

- センサ冗長性と重量・電力のトレード：ライダと深度の両方を追加すると堅牢性が向上するが、ペイロードとエネルギーにコストがかかる。
- 計算上のトレードオフ：高密度点群は検出を改善するが、加速処理（GPU または専用ビジョン ASIC）が必要。
- 安全リスクには、低プロファイル障害物の見落としと同期エラーによるファントム障害物が含まれる。クロスチェックと保守的計画マージンで軽減する。
- 展開時は、屋内混雑、動的群衆、多様な照明でのシナリオベーステストで検証し、検出範囲、偽陽性率、検出までの時間を定量化してグリッド解像度およびフィルタパラメータをチューニングする。

12 制御システム

12.1 モータ制御の基礎

前節で説明した電力配分のトレードオフとセンサ校正手法を基に、モータ制御は電気駆動能力とセンサフィードバックを統合し、安定した関節運動を実現する。本小節では制御問題を定式化し、電気機械ダイナミクスを解析し、ヒューマノイド関節制御の実用的な実装を示す。

問題定義。ヒューマノイド関節は高忠誠度位置制御、コンプライアントな相互作用、外乱抑制を要求する。設計はトルク制限、熱制約、リアルタイムデッドラインを満たす必要がある。制御問題は、電気・機械制約を尊重しながら所望関節軌道を生成するモータ指令を生成することに帰着する。

電気機械モデルと仮定。コントローラ設計に適した簡略化集中モデルを定義する：

$$[H]V(t) = K_e \omega(t) + R_a i(t) + L_a \frac{di(t)}{dt}, \quad (101)$$

$$[H]J \frac{d\omega(t)}{dt} + B \omega(t) = \tau(t) - \tau_{\text{load}}(t), \quad (102)$$

電圧 V 、逆起電圧定数 K_e 、アーマチュア抵抗 R_a 、インダクタンス L_a 、角速度 ω 、ロータ慣性 J 、粘性減衰 B 、モータトルク $\tau = K_t i$ 。ブラシレスモータではSI単位で $K_e \approx K_t$ 。これらの式はカスケード制御アーキテクチャを正当化する。

カスケード制御アーキテクチャ。ヒューマノイド関節のために3重ループを用いる：

1. 内側電流（トルク）ループ：高速、通常モータドライブファームウェアで実装。 $\tau \approx K_t I_{\text{ref}}$ を帯域 ω_i で実現。
2. 中間速度ループ：任意、減衰と軌道追従を改善。
3. 外側位置ループ：位置誤差から所望トルクまたは電流を計算。

実用的な外側ループ制御則はフィードフォワードトルクを含むPD構造：

$$[H]\tau_{\text{cmd}} = K_p (\theta_{\text{des}} - \theta) + K_d (\dot{\theta}_{\text{des}} - \dot{\theta}) + \tau_{\text{ff}}, \quad (103)$$

ここで θ は関節角、 τ_{ff} は重力と既知の慣性効果を補償する。内側電流ループを帯域内でユニティゲインと近似すると、式 (102) と (103) は閉ループ特性を与える：

$$[H]Js^2 + (B + K_d)s + K_p = 0, \quad (104)$$

よって固有振動数 $\omega_n = \sqrt{K_p/J}$ 、減衰比 $\zeta = (B + K_d)/(2\sqrt{JK_p})$ 。これらの関係は所望帯域・減衰のためのゲイン選択を指針とする。

ヒューマノイドの実装上の考慮事項。

- ・フィードフォワードと重力補償は静止姿勢近傍で積分器windアップと過熱を回避するために不可欠。
- ・摩擦力は非線形；摩擦力モデルを含めるかディザ/粘性項でスティクションを管理。
- ・ギアボックス（ハーモニックドライブ）はコンプライアンスとバックラッシュを導入。ゲイン調整時は等価剛性 k_{gear} でモデル化。

- シリーズエラスティックアクチュエータ（SEA）は意図的にコンプライアンスを追加；制御はモータ位置ではなくバネたわみ測定を基に速度/トルクリープを閉じる．
- センサ遅延と量子化（エンコーダ分解能）は達成可能な閉ループ帯域を制限．サンプリング時間は目標ループ帯域の少なくとも 10 倍にすべき．

実装スケッチ．以下の Python スニペットは速度制限付き外側 PD とフィードフォワード重力トルク，アンチwindアップ，電流制限を実装する．モータドライブが API でトルク設定値を公開すると仮定．`read_sensors()` と `send_torque()` をハードウェア固有の呼び出しに置き換える．

コードサンプル 43 ヒューマノイド関節外側ループ PD：フィードフォワードと安全クランプ付き．

```
#!/usr/bin/env python3
import time
import numpy as np
import threading
from typing import Tuple, Optional

# 制御パラメータ
Kp: float = 150.0
Kd: float = 5.0
MAX_TAU: float = 25.0 # Nm
SAMPLE_DT: float = 0.002 # 2 ms

# グローバル停止フラグ
running: threading.Event = threading.Event()
running.set()

def gravity_comp(theta: float) -> float:
    """ リンクの近似重力補償トルク """
    return 9.81 * 0.5 * 1.0 * np.sin(theta)

def read_sensors() -> Tuple[float, float]:
    """ エンコーダとIMUから角度・角速度を取得（ダミー実装） """
    # TODO: 実際のハードウェアインターフェースに置き換え
    return 0.0, 0.0

def compute_trajectory(t: float) -> Tuple[float, float, float]:
    """ 時刻tにおける目標角度・角速度・フィードフォワードトルク（ダミー） """
    # TODO: 軌道生成器に置き換え
```

```

    return 0.0, 0.0, 0.0

def send_torque(tau: float) -> None:
    """ドライバへトルク指令送信（ダミー）"""
    # TODO: 実際のCAN/EtherCAT等に置き換え
    pass

def control_loop() -> None:
    """2ms周期で実行されるリアルタイム制御ループ"""
    prev_error: float = 0.0
    t_start: float = time.perf_counter()

    while running.is_set():
        t0: float = time.perf_counter()

        # 時刻取得
        t: float = t0 - t_start

        # センサ読み出し
        theta, theta_dot = read_sensors()

        # 目標値生成
        theta_des, theta_dot_des, tau_ff = compute_trajectory(t)

        # PD+FF+重力補償
        error: float = theta_des - theta
        derror: float = (error - prev_error) / SAMPLE_DT
        tau_cmd: float = (
            Kp * error
            + Kd * (theta_dot_des - theta_dot)
            + tau_ff
            + gravity_comp(theta)
        )

        # 安全クランプ
        tau_cmd = np.clip(tau_cmd, -MAX_TAU, MAX_TAU)

        # 指令送信

```

```

        send_torque(tau_cmd)

    prev_error = error

    # 周期同期
    elapsed: float = time.perf_counter() - t0
    time.sleep(max(0.0, SAMPLE_DT - elapsed))

def main() -> None:
    """メインエントリ：スレッド起動&安全停止"""
    try:
        control_loop()
    except KeyboardInterrupt:
        running.clear()
        # トルクをゼロにして終了
        send_torque(0.0)

if __name__ == "__main__":
    main()

```

チューニング戦略. 低い K_p から始めて安定性を確認. 意図した帯域に到達するまで K_p を増加させ, 次に K_d を加えて減衰を与える. 周波数領域同定を用いて J と実効 B を推定. 可能であればハードウェアインザループを用いて, ハードウェアの前にシミュレーションでゲインを検証.

エンジニアリング上の意味, トレードオフ, リスク.

- 帯域対トルク: 高いゲインは応答性を上昇させるがモータ電流と発熱リスクを増大. 熱マージン計算でバランスを取る.
- コンプライアンス対精度: 弾性要素を追加するとピーク負荷を低減するが, トルクセンシングとより複雑な制御が必要.
- 安全上致命的な故障: エンコーダ脱落, ドライブ飽和, 通信遅延が不安定化を引き起こす. ウォッチドッグとフェイルセーフトルク低減を実装.
- リアルタイム制約: 決定論的ループタイミングが必須. 内側ループには RT 対応 OS または MCU を使用.
- 検証: 広範なシミュレーションと段階的ハードウェア試験により, フルヒューマノイドアセンブリへのコントローラ移行時のリスクを低減.

これらの運用上の要点が, ヒューマノイドモータ制御設計におけるコントローラアーキテクチャ, センサ割り当て, 緊急時対処戦略を決定する.

12.2 閉ループ対開ループシステム

モータ制御の基礎に続き、ヒューマノイド関節のための開ループと閉ループアーキテクチャを対比し、実用的な設計選択と電気工学への影響に焦点を当てる。以下の説明では制御問題を枠組みし、簡潔な関係式を導出し、組み込みヒューマノイドハードウェアに適した実行可能な制御ループ例を提示する。

ヒューマノイド関節制御問題の記述：アクチュエータは衝撃、重力変化、接触ダイナミクスからの外乱を抑制しながら所望の軌道を追従しなければならない。主要な制約には限られたアクチュエータ帯域、センサ遅延、熱限界がある。開ループと閉ループの選択は追従精度、安全性、エネルギー使用量に影響する。

定義と実用的な例。

- ・開ループ制御：コントローラは誤差を補正するためのセンサフィードバックを用いずにアクチュエータ指令を発行する。例：エンコーダなしでステッピングモータに N ステップ移動を指令する。
- ・閉ループ制御（フィードバック）：コントローラはシステム出力を測定し、誤差を減少させるために指令を調整する。例：エンコーダと PID コントローラを用いた関節位置サーボ。

技術解析。

- ・簡略化された回転形式の関節ダイナミクスは本質的な電気・機械相互作用を捉える：

$$[H]J\ddot{\theta}(t) + b\dot{\theta}(t) + \tau_g(\theta) = \tau_{\text{act}}(t) + d(t), \quad (105)$$

ここで J は慣性、 b は粘性減衰、 τ_g は重力トルク、 τ_{act} はアクチュエータトルク、 $d(t)$ は外乱である。

- ・追従誤差を $e(t) = r(t) - y(t)$ と定義し、基準 $r(t)$ 、測定 $y(t)$ とする。トルク指令に適用される標準的な PID 則は

$$[H]\tau_{\text{cmd}}(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t) + \tau_{\text{ff}}(t), \quad (106)$$

オプションのフィードフォワードトルク τ_{ff} は逆ダイナミクスから計算される。

- ・伝達関数形式では、トルク指令から出力に至るプラントダイナミクスを $G(s)$ 、コントローラを $C(s)$ とする。閉ループ伝達関数は

$$[H]T(s) = \frac{G(s)C(s)}{1 + G(s)C(s)}. \quad (107)$$

この式はコントローラゲインが定常誤差と外乱抑制をどう形作るかを示す。

開ループ対閉ループトレード解析。

- ・精度：閉ループはモデルミスマッチと外乱からの定常・過渡誤差を減少させる。開ループ精度は完全にモデル忠実度に依存する。
- ・頑健性：閉ループは未モデルダイナミクスとペイロード変化に対する頑健性を向上させる。しかし高いフィードバックゲインは未モデル高周波モードを励起し得る。
- ・複雑さとコスト：閉ループはセンサ、ADC、演算を必要とする。開ループはセンシングシステムによるハードウェア複雑さとエネルギーを削減する。

- 安全性：閉ループは能動的トルク制限と故障検出を可能にする。開ループは予期せぬ接触時に制御不能な挙動のリスクがある。

ヒューマノイド関節のための実用的設計パターン。

1. ネステッド制御ループ：

- インナ電流またはトルクリープ（高速、数百 Hz～kHz）。モータ電流センシングと低遅延電力電子機器を用いる。
- ミドル速度ループ（数百 Hz）。トルク指令を滑らかにし共振を減衰させる。
- アウタ位置ループ（数十～数百 Hz）。軌道追従と高次計画を扱う。

2. サンプリングと遅延制約：

- サンプル周波数 f_s を意図する閉ループ帯域の少なくとも十倍に選ぶ： $f_s \gtrsim 10 \times BW$ 。
- 演算・通信遅延を考慮する。純遅延 τ_d は位相余裕を周波数 ω で概ね $\Delta\phi \approx \omega\tau_d$ 減少させる。

3. ノイズとフィルタリング：

- エンコーダノイズを増幅しないよう測定値を慎重に微分する。生の有限差分の代わりに状態オブザーバまたはローパス微分フィルタを用いる。

4. アンチwindアップと飽和：

- アクチュエータ飽和時のオーバーシュートを防ぐため積分器アンチwindアップを実装する。飽和は達成可能なコンプライアンスも制限する。

実装：組込み制御ループ例。

- 以下の Python 疑似コードはヒューマノイド関節のためのタイトな閉ループ位置コントローラを示す。高速インナ電流制御層を仮定し、上位に位置 PID を露出する。コメントはハードウェア依存を示す。

コードサンプル 44 Position PID loop for a humanoid joint (pseudo-code)

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from builtin_interfaces.msg import Time
from std_msgs.msg import Float64
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectoryPoint
import threading
import numpy as np
from typing import Optional

class JointPidController(Node):
    def __init__(self) -> None:
        super().__init__('joint_pid_controller')
```

```

# --- ROS 2 パラメータ ---
self.declare_parameter('joint_name', 'joint_1')
self.declare_parameter('kp', 50.0)
self.declare_parameter('ki', 10.0)
self.declare_parameter('kd', 0.5)
self.declare_parameter('dt', 0.005)          # 200 Hz
self.declare_parameter('max_torque', 10.0)
self.declare_parameter('anti_windup', 0.1)   # 積分飽和防止係数

self.joint_name: str = self.get_parameter('joint_name').value
self.kp: float = self.get_parameter('kp').value
self.ki: float = self.get_parameter('ki').value
self.kd: float = self.get_parameter('kd').value
self.dt: float = self.get_parameter('dt').value
self.max_torque: float = self.get_parameter('max_torque').value
self.anti_windup: float = self.get_parameter('anti_windup').value

# --- 状態変数 ---
self.q_ref: float = 0.0
self.q: float = 0.0
self.qdot: float = 0.0
self.error: float = 0.0
self.error_prev: float = 0.0
self.integrator: float = 0.0

# --- 同期用 ---
self.state_lock = threading.Lock()

# --- QoS ---
qos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    history=HistoryPolicy.KEEP_LAST,
    depth=1
)

# --- 購読 ---
self.sub_js = self.create_subscription(
    JointState, 'joint_states', self.cb_js, qos)
self.sub_ref = self.create_subscription(

```

```

        JointTrajectoryPoint, 'joint_reference', self.cb_ref, qos)

# --- 配信 ---
self.pub_eff = self.create_publisher(Float64, 'effort_command', qos)

# --- タイマー ---
self.timer = self.create_timer(self.dt, self.control_cycle)

self.get_logger().info(f'{self.joint_name}_PID_controller_ready')

# -- コールバック --
def cb_js(self, msg: JointState) -> None:
    idx = msg.name.index(self.joint_name) if self.joint_name in msg.name else -1
    if idx < 0:
        return
    with self.state_lock:
        self.q = msg.position[idx]
        self.qdot = msg.velocity[idx] if len(msg.velocity) > idx else 0.0

def cb_ref(self, msg: JointTrajectoryPoint) -> None:
    with self.state_lock:
        self.q_ref = msg.positions[0] if msg.positions else self.q_ref

# -- 制御周期 --
def control_cycle(self) -> None:
    with self.state_lock:
        q_ref = self.q_ref
        q = self.q
        qdot = self.qdot

    error = q_ref - q
    self.integrator += error * self.dt

# 積分飽和防止
self.integrator = np.clip(
    self.integrator,
    -self.anti_windup / self.ki,
    self.anti_windup / self.ki
)

```

```

        derivative = (error - self.error_prev) / self.dt
        tau_ff = 0.0 # 必要に応じてfeedforwardを追加
        tau = self.kp * error + self.ki * self.integrator + self.kd * derivative + tau_ff
        tau = float(np.clip(tau, -self.max_torque, self.max_torque))

        self.error_prev = error

        # 指令送信
        cmd = Float64()
        cmd.data = tau
        self.pub_eff.publish(cmd)

def main(args=None):
    rclpy.init(args=args)
    node = JointPidController()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

工学への影響、トレードオフ、リスク。

- 環境と相互作用するアンクルや膝といった主要ヒューマノイド関節には閉ループ制御を用いる。頑健性と安全性の利点がセンサ・演算コストを正当化する。
- モデル忠実度が高く接触のない動作、例えば診断中の予測可能なアームスイープでは開ループが許容できる。
- 設計トレードオフ：
 - 高いゲインは外乱抑制を改善するがアクチュエータ発熱を増やし構造共振を励起し得る。
 - センサ分解能を上げると閉ループ性能が向上するがデータレートとノイズ感度が上昇する。
- 運用上のリスク：
 - センサ故障または飽和は不安定な閉ループ挙動を引き起こす。ウォッチドッグとフォールバック開ループプロファイルを実装する。
 - バス混雑からの通信遅延は位相余裕を減少させる。インナループのためのリアルタイムバス

トラフィックを優先する。

- 逆ダイナミクスフィードフォワードへの過信はアクチュエータ限界を隠蔽し、衝突時にトルク飽和を引き起こす。

これらのガイドラインはヒューマノイドプラットフォームでセンサ、モータドライブ、リアルタイムコントローラを統合する際の電気・制御エンジニアを導く。応答性、安全性、エネルギー効率をバランスさせるため、設計時に帯域、遅延、熱バジェットを定量化する。

12.3 リアルタイムデータ処理

前の小節では、閉ループ戦略と開ループ戦略の区別と、ヒューマノイドアクチュエータで使用されるモーター制御プリミティブを確立した。リアルタイムデータ処理は、センシング、推定、および駆動を厳密なタイミングと安定性要件に制約することで、これらのアイデアを結びつける。

問題定義。ヒューマノイドロボットは、異種のレイテンシ予算を持つ階層化された制御ループを必要とする。典型的なインナーループのトルクまたは電流制御は 500–2000 Hz で動作する。中レベルのバランスおよび歩容コントローラは 100–500 Hz で動作する。高レベルのプランナーとビジョンプロセスは 10–60 Hz で動作する。工学上の課題は、高レートのセンササンプルを取得し、それらを融合し、制御動作を計算し、不安定または安全でない動作を回避するために決定的なデッドライン内でアクチュエータコマンドを発行するデータパイプラインを設計することである。

技術分析。実用的な制御パイプラインは、次の段階に分解される：

1. センサ取得：IMU、ジョイントエンコーダ、力・トルクセンサ、およびカメラ。
2. 前処理：タイムスタンプ、折り返し防止、および外れ値除去。
3. 状態推定：センサ融合とフィルタによる姿勢、速度、および接触状態の提供。
4. 制御計算：フィードバック則、逆動力学、および軌道追従。
5. アクチュエータコマンド発行：モータードライバ、安全チェック、およびウォッチドッグ。

各段階はレイテンシに寄与する。 t_{acq} 、 t_{proc} 、 t_{est} 、 t_{ctrl} 、および t_{act} を段階レイテンシとする。総制御レイテンシは

$$[H]t_{total} = t_{acq} + t_{proc} + t_{est} + t_{ctrl} + t_{act}. \quad (108)$$

遅延は位相遅れを導入し、安定性マージンを減少させる。開ループカットオフ周波数 ω_c を持つコントローラについて、目標位相マージン損失 ϕ_{max} に対する許容遅延 τ_{max} は

$$[H]\tau_{max} = \frac{\phi_{max}}{\omega_c}. \quad (109)$$

例えば、 $\omega_c = 20 \text{ rad/s}$ および $\phi_{max} = \pi/6$ (30 度) の場合、 $\tau_{max} \approx 0.026 \text{ s}$ である。この境界はスケジューリングとハードウェア選択を導く。

実装パターンと緩和策。デッドラインを満たし、安定性を保持するために次の工学技術を使用する：

- ハードリアルタイムとソフトリアルタイムの分割：インナーループトルク制御をリアルタイム対応マイクロコントローラまたは RTOS で実装する。ビジョンと高レベルプランニングにはソフトリアルタイムホストを使用する。
- ロックフリーバッファリング：センササンプル用リングバッファを使用して、リアルタイムパス

でのブロッキングを回避する。

- オフロードと並列化：センサストリーミング用 DMA、決定的な前置フィルタリング用 FPGA、バッチビジョン推論用 GPU。
- 時刻同期：センサ間でタイムスタンプを PTP またはハードウェアトリガで同期し、一貫した融合を実現する。
- レート適応：厳しいレイテンシ予算の下で高価なセンサをダウンサンプルまたはイベント駆動する。
- 優先度と CPU 分離： $SCHED_FIFO$ でリアルタイムスレッドをスケジュールし、CPU を分離してジッタを削減する。
- ウォッチドッグと安全モニタ：アクチュエータコマンドタイムアウトとフォールバック動作を強制する。

状態推定の例。慣性安定化のために、相補フィルタはジャイロスコープ積分と加速度計角度推定を融合する。サンプリング周期 T_s で、離散相補更新は

$$\theta[k] = \alpha(\theta[k-1] + \omega_{\text{gyro}}[k] T_s) + (1 - \alpha)\theta_{\text{acc}}[k],$$

ここで、 $\alpha \in (0, 1)$ は融合帯域幅を設定する。 α を選択することは、応答性と雑音除去のトレードオフである。推定器は、最小位相遅れのために IMU で利用可能な最高レートで動作しなければならない。

実用的なコード例。リストは、IMU とエンコーダを読み取り、相補フィルタを適用し、トルクセットポイントを出力するリアルタイム制御ループのプロトタイピングパターンを示す。この Python の例はシミュレーションまたはプロトタイピング専用であり、本番システムではハードリアルタイム保証のために C/C++ または RTOS コンポーネントを使用すべきである。

コードサンプル 45 Prototyping real-time pipeline with lock-free ring buffer

```
#!/usr/bin/env python3
import math
import threading
import time
from collections import deque
from dataclasses import dataclass
from typing import Deque, Dict, Final, Optional

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Float32MultiArray

@dataclass
class ImuSample:
    stamp: float
```

```

acc_x: float
acc_y: float
acc_z: float
gyro_x: float
gyro_y: float
gyro_z: float

```

```

class ImuDriver:
    """ハードウェア依存を隠蔽したIMUドライバ"""
    def read(self) -> ImuSample:
        # TODO: 実機ではI2C/SPI経由で読み出す
        raise NotImplementedError

class TorqueDriver:
    """モータードライバへの非ブロッキング送信インターフェース"""
    def send(self, tau: float) -> None:
        # TODO: 実機ではCAN/USB 送信
        raise NotImplementedError

class ImuBuffer:
    """最新サンプルをロックフリーで取得するリングバッファ"""
    def __init__(self, maxlen: int = 256) -> None:
        self._buf: Deque[ImuSample] = deque(maxlen=maxlen)
        self._lock = threading.Lock()

    def append(self, sample: ImuSample) -> None:
        with self._lock:
            self._buf.append(sample)

    def latest(self) -> Optional[ImuSample]:
        with self._lock:
            return self._buf[-1] if self._buf else None

class BalanceController(Node):
    """倒立振子PD制御ノード"""
    # 制御パラメータ

```

```

Ts: Final[float] = 0.002                # 500 Hz
ALPHA: Final[float] = 0.98               # 相補フィルタ係数
KP: Final[float] = 12.0
KD: Final[float] = 0.8

def __init__(self) -> None:
    super().__init__('balance_controller')
    self._imu_buf = ImuBuffer()
    self._theta = 0.0
    self._torque = TorqueDriver()

    qos = QoSProfile(
        reliability=ReliabilityPolicy.BEST_EFFORT,
        history=HistoryPolicy.KEEP_LAST,
        depth=1
    )
    self._pub = self.create_publisher(Float32MultiArray, 'joint_command', qos)

    # スレッド起動
    self._imu_thread = threading.Thread(target=self._imu_reader, daemon=True)
    self._ctl_thread = threading.Thread(target=self._control_loop, daemon=True)
    self._imu_thread.start()
    self._ctl_thread.start()

def _imu_reader(self) -> None:
    """IMU取得スレッド (1kHz) """
    imu = ImuDriver()
    while rclpy.ok():
        raw = imu.read()
        stamp = time.time()
        self._imu_buf.append(
            ImuSample(
                stamp=stamp,
                acc_x=raw.acc_x,
                acc_y=raw.acc_y,
                acc_z=raw.acc_z,
                gyro_x=raw.gyro_x,
                gyro_y=raw.gyro_y,
                gyro_z=raw.gyro_z,
            )
        )

```

```

    )
    time.sleep(0.001)

def _control_loop(self) -> None:
    """500Hz PD制御ループ"""
    next_time = time.time()
    while rclpy.ok():
        sample = self._imu_buf.latest()
        if sample is None:
            continue

        # 相補フィルタで姿勢推定
        acc_angle = math.atan2(sample.acc_y, sample.acc_z)
        self._theta = (
            self.ALPHA * (self._theta + sample.gyro_z * self.Ts)
            + (1.0 - self.ALPHA) * acc_angle
        )

        # PD制御
        error = -self._theta
        u = self.KP * error - self.KD * sample.gyro_z
        self._publish_torque(u)

        # 周期維持
        next_time += self.Ts
        sleep_time = next_time - time.time()
        if sleep_time > 0:
            time.sleep(sleep_time)
        else:
            next_time = time.time() # デッドラインミス再同期

def _publish_torque(self, tau: float) -> None:
    msg = Float32MultiArray()
    msg.data = [tau]
    self._pub.publish(msg)
    self._torque.send(tau)

def main(args=None):
    rclpy.init(args=args)

```

```

node = BalanceController()
try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

工学上の影響、トレードオフ、およびリスク。決定的なリアルタイムコンポーネントと柔軟なソフトウェアの複雑さと消費電力を増加させる。積極的なサンプリングは応答性を改善するが、通信および処理負荷を増加させる。 t_{total} を境界付けできないと、位相マージンが減少し、振動が発生し、アクチュエータが飽和し、安全でない転倒が引き起こされる可能性がある。リスクを、階層化された安全モニタ、境界付きレートアクチュエータ、およびデッドライン逸脱時にヒューマノイド動作を制限する段階的な縮退モードで緩和する。

12.4 ヒューマノイドモーション制御における課題

リアルタイムデータ処理における決定論的タイミングとレイテンシ制約、および閉ループ対開ループシステムの安定性トレードオフを踏まえ、本小節ではヒューマノイドの運動を指令する際に生じる具体的な制御上の課題を分析する。焦点は運用上の関連性にあり、センシング、計算、駆動がどのように相互作用して安定で安全なヒューマノイド挙動を生成するかである。

問題定義. ヒューマノイドモーション制御は、複数のしばしば矛盾する要求を同時に満たす必要がある：

- マルチフェーズ接触シーケンス中にロボットのバランスを維持する。
- 限られたアクチュエータ帯域で歩容・操縦軌道を追従する。
- 厳格なリアルタイム予算内で外部摂動に反応する。
- センサ故障および未モデル動作下で安全性を保持する。

技術分析. 困難の主な源は高次元性、片脚支持期の駆動不足、高速接触切替、センサ・アクチュエータ非理想性、計算遅延である。全身動力学は多自由度 (DoF) を結合する。歩行解析に用いられるコンパクトな表現は線形倒立振子モデル (LIPM) である：

$$[H]\ddot{x} = \omega^2(x - p), \quad (110)$$

ここで x は重心 (CoM) 水平位置、 p は支持点 (ZMP) 位置、 $\omega = \sqrt{g/z_c}$ は定常 CoM 高さ z_c に対する値である。バランス回復に有用な瞬時捕捉点 (ICP) は

$$[H]x_{\text{cp}} = x + \frac{\dot{x}}{\omega}. \quad (111)$$

これらの簡略化関係はプランナ設計を導くが、完全制御には逆動力学が必要である。剛体逆動力学方程式は

$$[H]M(q)\ddot{q} + C(q, \dot{q}) + g(q) = S^T \tau + J_c^T \lambda, \quad (112)$$

一般化座標 q 、質量行列 M 、コリオリ・遠心力項 C 、重力 g 、アクチュエータ選択行列 S 、関節トルク τ 、接触ヤコビアン J_c 、接触レンチ λ を伴う。実用的なコントローラは (4) を二次計画 (QP) で用いて接触制約とトルク制限を満たすトルク指令を計算する。

遅延とサンプリング効果は決定的である。制御ループ遅延 τ_d とサンプリング周期 T_s は達成可能な閉ループ帯域幅を低下させる。単純なコントローラで交差角周波数 ω_c を持つ場合、純遅延による位相遅れは $\omega_c \tau_d$ に等しい。最小位相余裕 ϕ_{\min} を保持するには

$$[H]\omega_c \tau_d < \phi_{\min}. \quad (113)$$

より大きな遅延は ω_c を低く強制し、摂動除去を損なう。ヒューマノイドでは許容 τ_d はインナーループトルク制御でしばしば数ミリ秒未満である。

センサ融合と状態推定制約も同様に重要である。IMU バイアスは姿勢ドリフトを引き起こす。接触検出誤差は CoM・ZMP 推定を破損する。実用的な状態推定器は IMU、関節エンコーダ、力・トルクセンサ、ビジョンを拡張カルマンフィルタ (EKF) で統合する。推定器はインナーループ向けに高レート低レイテンシ状態出力を提供しつつ、低速知覚モジュールがプランナ向けに環境モデルを洗練する。

実装パターン. 頑強なヒューマノイドスタックは制御を層に分解する：

1. 高レベルプランナ：歩行パターン、足場位置、10–100 Hz で動作。
2. ミドルレベル全身コントローラ：逆動力学 QP または動作空間コントローラ、100–200 Hz。
3. ローレベルトルク・位置ループ：アクチュエータドライバ、500–2000 Hz。

設計原則：

- ・重力と所望加速度を補償するモデルベースフィードフォワードトルクを用いる。
- ・未モデル高周波コンプライアンスを励起しないよう低ゲインフィードバックを加える。
- ・関節角度・トルク・接触力のハードセーフティ制限を実施する。
- ・リアルタイム OS と優先スケジューリングでクリティカルループを決定論的に保つ。

実装例. 以下の C++制御ループスニペットは最小リアルタイムトルクループ構造を示す。低レイテンシ読取、逆動力学呼出、トルク飽和、ウォッチドッグセーフティチェックを実演する。

コードサンプル 46 Realtime torque-loop skeleton (conceptual).

```
cpp
#include <chrono>
#include <cmath>
#include <array>
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/joint_state.hpp>
#include <std_msgs/msg/float64_multi_array.hpp>
#include <hardware_interface/system_interface.hpp>
```

```

using namespace std::chrono_literals;

constexpr size_t kNumJoints = 12;
using JointArray = std::array<double, kNumJoints>;

class HighRateController : public rclcpp::Node
{
public:
    HighRateController()
    : Node("high_rate_controller"),
      clock_(std::make_shared<rclcpp::Clock>(RCL_STEADY_TIME)),
      torque_pub_(create_publisher<std_msgs::msg::Float64MultiArray>("joint_torque_cmd",
                                                                    *this)),
      state_sub_(create_subscription<sensor_msgs::msg::JointState>(
          "joint_states", 1,
          [this](const sensor_msgs::msg::JointState::SharedPtr msg) { latest_state_ = *msg; })),
      loop_timer_(create_wall_timer(1ms, std::bind(&HighRateController::controlLoop, this)))
    {
        declare_parameter("torque_limit", std::vector<double>(kNumJoints, 40.0));
        get_parameter("torque_limit", torque_limit_);
    }

private:
    void controlLoop()
    {
        const auto t0 = clock_->now();

        // センサー未受信ならスキップ
        if (latest_state_.position.empty() || latest_state_.velocity.empty()) return;

        // 状態ベクトル構築
        JointArray q{}, qdot{};
        std::copy_n(latest_state_.position.begin(), kNumJoints, q.begin());
        std::copy_n(latest_state_.velocity.begin(), kNumJoints, qdot.begin());

        // 100 Hzで目標加速度更新
        static auto last_plan = clock_->now();
        if ((t0 - last_plan) >= 10ms) {
            qdd_des_ = planner_.get_qdd_des(t0.seconds());
            last_plan = t0;
        }
    }
};

```

```

// 逆動力学フィードフォワード
const auto tau_ff = inverseDynamics(M_, C_, g_, q, qdot, qdd_des_);

// PDフィードバック (低ゲイン)
JointArray tau_fb{};
for (size_t i = 0; i < kNumJoints; ++i) {
    tau_fb[i] = Kp_ * (q_des_[i] - q[i]) + Kd_ * (qdot_des_[i] - qdot[i]);
}

JointArray tau_cmd{};
for (size_t i = 0; i < kNumJoints; ++i) {
    tau_cmd[i] = tau_ff[i] + tau_fb[i];
    // トルクリミット適用
    if (std::abs(tau_cmd[i]) > torque_limit_[i]) {
        tau_cmd[i] = std::copysign(torque_limit_[i], tau_cmd[i]);
    }
}

// 力センサオーバーロード時は即時ゼロトルク
if (ft_sensor_.overload()) {
    tau_cmd.fill(0.0);
    estop_.raise();
}

// 指令送信
auto msg = std_msgs::msg::Float64MultiArray();
msg.data.assign(tau_cmd.begin(), tau_cmd.end());
torque_pub_ -> publish(msg);

// 遅延監視
const auto latency = (clock_ -> now() - t0).seconds();
if (latency > 0.001) RCLCPP_WARN_THROTTLE(get_logger(), *get_clock(), 1000, "制御局
}

rclcpp::Clock::SharedPtr clock_;
rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::SharedPtr torque_pub_;
rclcpp::Subscription<sensor_msgs::msg::JointState>::SharedPtr state_sub_;
rclcpp::TimerBase::SharedPtr loop_timer_;

```

```

sensor_msgs::msg::JointState latest_state_;

struct {
    JointArray get_qdd_des(double) { return {}; }
} planner_;
struct {
    bool overload() { return false; }
} ft_sensor_;
struct {
    void raise() {}
} estop_;

JointArray q_des_{}, qd_des_{}, qdd_des_{};
std::vector<double> torque_limit_;
const double Kp_ = 20.0, Kd_ = 1.0;
std::array<double, kNumJoints * kNumJoints> M_{}, C_{};
JointArray g_{};
JointArray inverseDynamics(const auto&, const auto&, const auto&,
                           const JointArray& q, const JointArray& qdot,
                           const JointArray& qdd)
{
    JointArray tau{};
    // 簡易計算例（実機ではライブラリ使用）
    for (size_t i = 0; i < kNumJoints; ++i) {
        tau[i] = 1.0 * qdd[i] + 0.1 * qdot[i] + 0.05 * q[i];
    }
    return tau;
}

};

int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<HighRateController>());
    rclcpp::shutdown();
    return 0;
}

```

技術的影響とトレードオフ.

- 計算レイテンシ：リッチモデル（例：完全剛体 QP）は性能を向上させるが計算時間を増やす。

ハードウェア加速または簡略モデルを用いて期限を満たす。

- ・剛性対コンプライアンス：高インピーダンスは追従性を向上させるが衝撃頑健性を悪化させる。能動コンプライアンスまたは直列弾性アクチュエータが衝撃を緩和する。
- ・帯域幅対雑音増幅：フィードバックゲインを上げるとセンサ雑音への感度が上昇する。遅延を意識したフィルタリングを適用する。
- ・エネルギー対安定性：積極的な安定化はしばしば消費電力を増やす。歩行軌道をエネルギー意識安定性に最適化する。

運用上のリスクと緩和策.

- ・未モデル動力学による振動：ゲインを下げ、ノッチフィルタを追加、またはモデル忠実度を上げる。
- ・推定器発散による転倒：フォールバック挙動とソフト接触挙動を実装する。
- ・持続高トルクによるアクチュエータ過熱：熱状態を監視し熱考慮トルク制限を含める。

設計トレードオフは定量化要求として文書化すべきである。最悪ケースループ遅延、要求位相余裕、許容 CoM 偏差を指定する。レイヤードセーフティを実装して故障を隔離する。これらの手順によりヒューマノイドモーション制御における故障の確率と影響を低減する。

ソフトウェアの基礎

13 ロボティクスオペレーティングシステムの導入

13.1 ROS および ROS2 の概要

先に議論したリアルタイム制御とセンサ統合の基盤を踏まえ、本小節ではミドルウェアの選択を人間型ロボットの性能に直接影響する工学的決定として位置づける。以下では、人間型ロボットシステムにおける ROS (ROS1) と ROS2 のアーキテクチャ機能、タイミングモデル、実装上のパターンを比較する。

問題定義と運用上の関連性。人間型ロボットは、複数の計算ノードにまたがって密結合したセンシング、状態推定、アクチュエータコマンドを必要とする。制約事項は以下の通り：

- ・バランスおよび反射ループのための低遅延かつ決定的な遅延。
- ・ビジョンおよびポイントクラウドデータのための高スループット。
- ・分散計算（エッジ、オンボード、クラウド）のための堅牢なプロセス間・マシン間通信。
- ・長期デプロイメントにおける安全性、セキュリティ、ライフサイクル管理。

技術分析：コア抽象化とタイミングモデル。ROS1 も ROS2 も、ノード、トピック、サービス、アクションを主要な抽象化として公開する。人間型ロボットにおいて、システムレベルでの差異が重要となる：

- ・ミドルウェアと決定性：ROS2 はトランスポート層として DDS (Data Distribution Service) を使用する。DDS は信頼性、耐久性、デッドライン、遅延予算のための明示的な Quality-of-Service (QoS) 制御を提供する。これらの QoS ノブは、IMU フィードバックおよびモータ駆動の制御要

件に直接対応する。

- プロセスモデルとコンポジション：ROS2 は単一プロセス内でコンポーザブルノードをサポートし、プロセス間通信（IPC）オーバーヘッドを回避する。コンポジションはメッセージごとの遅延とジッタを削減し、数百 Hz で動作する制御ループにとって重要である。
- ライフサイクルとデプロイメント：ROS2 ライフサイクルノードは（設定済み、アクティブ、非アクティブ）という制御された遷移を可能にし、人間型ロボットのサブシステムに対して安全な立ち上げと決定的なシャットダウンシーケンスを実現する。

単純な遅延分解は、閉ループ動作のためのタイミング予算を工学的に立案するのに役立つ。全ループ遅延を

$$[H]T_{\text{loop}} = T_{\text{sense}} + T_{\text{comm}} + T_{\text{proc}} + T_{\text{act}}, \quad (114)$$

と表現する。ここで T_{comm} はミドルウェア選択と QoS の影響を受ける。リンクでのキューイング安定性のため、到着率 λ がサービス率 μ より小さいことを確保し、すなわち $\rho = \lambda/\mu < 1$ とする。

リアルタイム CPU 上の周期的制御タスクのスケジューラビリティについては、レート単調スケジューリング（RMS）の利用率境界を用いる：

$$[H]U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n \left(2^{1/n} - 1 \right), \quad (115)$$

ここで C_i はタスク i の計算時間、 T_i は周期である。ミドルウェアオーバーヘッドは C_i に影響し、許容される制御周期に影響を与える。

人間型ロボットサブシステムに対する実践的なミドルウェア決定：

- 高周波バランスループ（200–1000 Hz）：インプロセスコンポジションまたは RT 共有メモリトランスポート、最小 QoS オーバーヘッド、RT カーネルスケジューリングを使用。
- ビジョンパイプライン（30–120 Hz、高帯域幅）：*BEST_EFFORT* 信頼性と大きな履歴深度を使用し、リアルタイムスレッドのブロックを回避。
- テレオペレーションおよびミッションコマンド：RELIABLE QoS と永続的耐久性を使用し、再起動後の安全な復旧を実現。

実装例。以下の ROS2 rclpy 例は、低遅延センシング用に最適化された IMU パブリッシャと、信頼性を要求するモータコマンドサブスクリバを示す。コメントは人間型ロボット統合に適した設計選択を示す。

コードサンプル 47 ROS2 rclpy ノード：IMU をパブリッシュし、QoS 選択でモータコマンドをサブスクリブ。

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
from rclpy.duration import Duration
from rclpy.executors import MultiThreadedExecutor
from rclpy.callback_groups import ReentrantCallbackGroup, MutuallyExclusiveCallbackGroup
from sensor_msgs.msg import Imu
from std_msgs.msg import String
```

```

from geometry_msgs.msg import Vector3
import math
import time
import threading
from typing import Optional

class ImuMotorBridge(Node):
    def __init__(self) -> None:
        super().__init__('imu_motor_bridge')

        self.declare_parameter('imu_frame_id', 'imu_link')
        self.declare_parameter('publish_rate_hz', 200.0)
        self.declare_parameter('use_sim_time', False)

        self._imu_frame: str = self.get_parameter('imu_frame_id').value
        self._rate: float = self.get_parameter('publish_rate_hz').value
        self._period: float = 1.0 / self._rate

        self._cb_group_imu = ReentrantCallbackGroup()
        self._cb_group_cmd = MutuallyExclusiveCallbackGroup()

        imu_qos = QoSProfile(
            depth=10,
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            deadline=Duration(nanoseconds=10_000_000)
        )
        cmd_qos = QoSProfile(
            depth=1,
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            durability=DurabilityPolicy.VOLATILE
        )

        self._imu_pub = self.create_publisher(
            Imu, '/imu/data_raw', imu_qos, callback_group=self._cb_group_imu
        )
        self._cmd_sub = self.create_subscription(
            String, '/motor/commands', self._cmd_callback, cmd_qos,
            callback_group=self._cb_group_cmd

```

```

    )

    self._timer = self.create_timer(
        self._period, self._publish_imu, callback_group=self._cb_group_imu
    )

    self._last_cmd: Optional[str] = None
    self._lock = threading.Lock()

    self.get_logger().info(f'IMU_bridge_ready. Frame={self._imu_frame}, Rate={self._rate}')

def _publish_imu(self) -> None:
    msg = Imu()
    msg.header.stamp = self.get_clock().now().to_msg()
    msg.header.frame_id = self._imu_frame

    # ダミー値：実機ではドライバから取得
    msg.angular_velocity = Vector3(x=0.0, y=0.0, z=0.0)
    msg.linear_acceleration = Vector3(x=0.0, y=0.0, z=9.81)
    msg.angular_velocity_covariance[0] = -1.0
    msg.linear_acceleration_covariance[0] = -1.0

    self._imu_pub.publish(msg)

def _cmd_callback(self, msg: String) -> None:
    with self._lock:
        self._last_cmd = msg.data
    # 高速パス：コマンドを即座にアクチュエータへ転送（実装省略）
    # 必要に応じてCAN/USB送信を別スレッドで実行

def main(args=None):
    rclpy.init(args=args)
    node = ImuMotorBridge()
    executor = MultiThreadedExecutor()
    executor.add_node(node)
    try:
        executor.spin()
    except KeyboardInterrupt:
        pass
    finally:

```

```
executor.shutdown()  
node.destroy_node()  
rclpy.shutdown()
```

工学的指針とデプロイメントチェックリスト：

- ros2 topic hz、ros2 bag、OS レベルトレーシングなどのツールを用いてエンドツーエンドループ時間をプロファイリングする。
- 制御クリティカルノードにはコンポジションを用いて IPC 遅延を除去する。
- トピックごとに QoS を設定：非クリティカル高レートセンサには *BEST_EFFORT*；コマンドおよび状態には *RELIABLE* を配信必須とする。
- バランスおよび反射ループはリアルタイムカーネル上で動作させ、ミドルウェアオーバーヘッドを含めた RMS 境界（式 115）を検証する。
- 高帯域ビジョンパイプラインを別 NIC または計算ノードに隔離し、制御トラフィックとの競合を回避する。

設計トレードオフと運用上のリスク：

1. セキュリティ vs 遅延：DDS セキュリティを有効にするとハンドシェイクと暗号化コストが加わる。本番導入前に遅延影響をテストする。
2. 決定性 vs 利便性：ROS1 はツーリングが簡単だが、DDS QoS やライフサイクル制御が欠ける。ROS2 は決定性を高めるが、より深い設定が必要。
3. QoS の誤設定は黙ってパケットドロップまたはブロックを引き起こす。代表負荷下で QoS 組合せを必ずテストする。
4. コンポジションは IPC ジッタを削減するが、バグの影響範囲を広げる。信頼できないコードにはプロセスレベル隔離を用いる。

これらの考察はミドルウェア選択を測定可能な性能指標にマッピングする。エンジニアは QoS 設定、コンポジション戦略、スケジューリング優先度を繰り返し調整し、ハードウェアインザループテストで閉ループ安定性と安全性を検証すべきである。

13.2 ヒューマノイドロボット向け ROS のセットアップ

ROS および ROS2 の概要を踏まえ、本小節ではヒューマノイドロボットへ ROS を展開するための具体的な手順とエンジニアリング判断を詳述する。ミドルウェアおよびツールの選択をリアルタイム制御、センサストリーム、シミュレーション相互運用性に結びつける。

ヒューマノイドロボットは多自由度、多数のセンサ、タイトな制御ループを組み合わせる。主要な課題は、安全性と開発生産性を保ちながら、コントローラとハードウェア間で決定論的な指令・状態フローを実現することである。重要なエンジニアリング目標は以下の通りである：

- ジョイントレベルコントローラのための決定論的制御ループタイミング；
- URDF/SDF および TF フレームによる一貫した変換とボディ；
- IMU、カメラ、力センサのための信頼できるセンサデータパイプライン；
- 再現可能なシミュレーションからハードウェアへのワークフロー。

技術分析および推奨アーキテクチャ

1. ROS ディストリおよびミドルウェアの選択.
 - 新規システムでは、改良されたミドルウェア、サービス品質 (QoS)、ライフサイクルノードのため ROS 2 を優先する.
 - 遅延を調整するために、プラットフォーム認定済み DDS 実装 (例: Fast DDS, Cyclone DDS) を使用する.
2. ロボット記述と静的設定.
 - 慣性、衝突、視覚要素を含む完全な URDF を公開する.
 - `robot_state_publisher` を使用してジョイント変換を配信する. ジョイント命名をハードウェアドライバと一致させる.
3. ハードウェアインタフェースとコントローラ.
 - ハードウェアインタフェースと `controller_manager` を分離するために `ros2_control` を使用する.
 - リアルタイムセーフなハードウェアインタフェースを実装する. ソフトリアルタイムで不十分な場合は、リアルタイム対応エグゼキュータおよびリアルタイム OS カーネルを使用する.
4. 制御ループ周波数設計.
 - アクチュエータ帯域幅 f_{bw} を定義し、エイリアシングを回避し位相余裕を維持するためにコントローラ更新レート f_s を設定する.
 - 実用的なルールは
$$[H]f_s > 10 f_{bw}, \quad (116)$$
これは代表的な PID およびモデルベースコントローラに対して適切な離散化余裕を与える.
5. センサストリームと QoS.
 - 損失ありセンサ (カメラ) および高レート IMU 向けに ROS 2 QoS プロファイルを設定する.
 - 高帯域カメラには `reliability = best_effort` を使用; IMU および状態トピックには `reliability = reliable` を使用する.
6. 時刻同期.
 - NTP, chrony, または Precision Time Protocol (PTP) を使用してネットワーククロックを同期し、コンポーネント間でサブミリ秒精度を実現する.
7. ネットワークパーティショニングと名前空間.
 - 四肢またはサブシステムごとの分離のため ROS 名前空間を使用する.
 - マルチマシン展開では、DDS のユニキャスト/マルチキャスト動作を調整してジッタを削減することを確認する.
8. シミュレーションとハードウェアの等価性.
 - URDF/SDF, TF フレーム, コントローラパラメータをシミュレーションとハードウェア間で同一に保つ.
 - 現実的なセンサフィードバックのため Gazebo または Isaac Sim と ROS ブリッジを使用する.

実装チェックリスト (実践的な手順)

1. ROS 2 ワークスペースを作成し、URDF, コントローラ, 起動ファイルをパッケージ化する.
2. `ros2_control` API に準拠したリアルタイムハードウェアインタフェースを実装する.

3. 安全な起動およびシャットダウンを可能にするため、クリティカルサブシステム用ライフサイクルノードを記述する。
4. ハードウェア展開前にシミュレーションでコントローラゲインを調整する。
5. メッセージスキーマの単体テストとシミュレーション統合テストを継続的インテグレーションで実施する。

最小限の *ros2_control* 設定例

```
joint_state_broadcaster:
  type: joint_state_broadcaster/JointStateBroadcaster

lower_body_trajectory_controller:
  type: joint_trajectory_controller/JointTrajectoryController

lower_body_trajectory_controller:
  ros__parameters:
    joints: # 下半身12関節
      - left_hip_pitch
      - left_hip_roll
      - left_hip_yaw
      - left_knee
      - left_ankle_pitch
      - left_ankle_roll
      - right_hip_pitch
      - right_hip_roll
      - right_hip_yaw
      - right_knee
      - right_ankle_pitch
      - right_ankle_roll

    command_interfaces:
      - position
    state_interfaces:
      - position
      - velocity

    state_publish_rate: 100.0 # 状態配信周波数 [Hz]
    action_monitor_rate: 20.0 # アクション監視周波数 [Hz]

    allow_partial_joints_goal: false # 全関節必須
    open_loop_control: false # フィードバック有効
```

```

constraints:
  stopped_velocity_tolerance: 0.01 # [rad/s]
  goal_time: 0.5 # ゴール到達許容時間 [s]

  left_hip_pitch: { trajectory: 0.5, goal: 0.02 }
  left_hip_roll: { trajectory: 0.5, goal: 0.02 }
  left_hip_yaw: { trajectory: 0.5, goal: 0.02 }
  left_knee: { trajectory: 0.5, goal: 0.02 }
  left_ankle_pitch: { trajectory: 0.5, goal: 0.02 }
  left_ankle_roll: { trajectory: 0.5, goal: 0.02 }
  right_hip_pitch: { trajectory: 0.5, goal: 0.02 }
  right_hip_roll: { trajectory: 0.5, goal: 0.02 }
  right_hip_yaw: { trajectory: 0.5, goal: 0.02 }
  right_knee: { trajectory: 0.5, goal: 0.02 }
  right_ankle_pitch: { trajectory: 0.5, goal: 0.02 }
  right_ankle_roll: { trajectory: 0.5, goal: 0.02 }

```

実践的なチューニングおよびテスト戦略

- シミュレーションで低周波位置制御から開始する。運動学、衝突、TF 一貫性を検証する。
- CPU およびネットワーク負荷を監視しながら *update_rate* を目標値へ段階的に上げる。
- ソフトウェアおよびファームウェアの両方で安全停止およびソフトリミットを適用し、ハードウェアインザループテストを実施する。

設計トレードオフおよび運用上のリスク

- リアルタイム決定論性対開発速度：ハードリアルタイム OS またはマイクロカーネルを使用すると決定論性が向上するが、開発複雑性が増し移植性が低下する。
- QoS 選択は信頼性と遅延をトレードオフする。ベストエフォートはカメラストリームの遅延を削減するが、負荷下でフレーム落ちする可能性がある。
- 高コントローラ更新レートは位相遅延を削減するが、CPU およびバス使用率を増加させる。ピーク計算およびセンサバーストに対するヘッドルームを検証する。
- ネットワーク設定エラーは断続的な制御障害として現れることがある。DDS 設定を検証し、ROS トラフィックを非ロボットフローから分離する。
- セキュリティおよびライフサイクル管理：ライフサイクルノードは安全な遷移を改善するが、復旧中の慎重なオーケストレーションを必要とする。

具体的なエンジニアリングへの影響

- コントローラスレッド用に CPU コアを割り当て、スレッドアフィニティを設定して遅延を上限に抑える。
- 誤設定時の暴走指令を防ぐため、ウォッチドッグおよびハードウェアリミットを追加する。

- ・シミュレーションとハードウェア記述子の等価性を維持し、移行リスクを削減する。
- ・予測可能な統合を可能にするため、各センサおよびコントローラの QoS およびタイミング要件を文書化する。

全体として、セットアップ判断はヒューマノイドロボットの安全マージン、制御品質、開発生産性を決定する。決定論的制御パス、堅牢なセンサ QoS、再現可能なシミュレーション等価性を優先し、展開リスクを削減する。

13.3 基本的な ROS ノードとトピック

ROS のインストール、設定、および ROS と ROS2 のアーキテクチャ選択に続き、ヒューマノイドロボットで使用される最も単純な通信パターンを検討する。本小節では、ノード、トピック、およびセンシング、状態推定、低遅延アクチュエータコマンドを実装する実用的なメッセージングパターンに焦点を当てる。

ノード、トピック、メッセージセマンティクスは、ヒューマノイド上の疎結合ソフトウェアの主要なプリミティブである。ノードは単一のプロセスで機能を実行し、ヒューマノイドでは通常以下のいずれかである：

- ・センサ取得 (IMU、力・トルク、カメラ)、
- ・状態公開 (関節位置・速度)、
- ・低レベルアクチュエータコマンド生成、
- ・高レベルプランナーおよび動作マネージャ。

トピックは型付きメッセージのストリームをパブリッシャとサブスクライバ間で伝える。ヒューマノイドにおける一般的なトピックの役割は以下の通り：

- ・状態ブロードキャスト：sensor_msgs/JointState メッセージを運ぶ/joint_states。
- ・アクチュエータコマンド：trajectory_msgs/JointTrajectory を使用する/arm_controller/command。
- ・慣性センシング：sensor_msgs/Imu を使用する/imu/data。
- ・接触または力センシング：geometry_msgs/WrenchStamped を使用する/ft_sensor。

設計問題の提示：決定的遅延でアクチュエータコマンドを配信しつつ、高レートセンサデータを消費する。競合する 2 要件が生じる：

1. 制御ループのための高公開レート (通常 100–1000 Hz)。
2. カメラのようなリッチセンサを公開する際の帯域幅および CPU 制約。

技術分析はサンプリングと制御離散化に中心を置く。 f_s をトピック公開周波数、 $T_s = 1/f_s$ をサンプリング間隔とする。/joint_states をサブスクライブする制御ノードで実装される一般的な離散 PD 則は

$$[H]u[k] = K_p e[k] + K_d \frac{e[k] - e[k-1]}{T_s}, \quad (117)$$

ここで $e[k]$ はサンプル k における位置誤差である。この式は T_s への明示的依存を示す。 T_s が変動すると、微分項がノイズを増幅し制御ループを不安定化させる可能性がある。

実装上の考慮事項とベストプラクティス：

- リアルタイム制御ノードを重い知覚ノードから分離する。リアルタイムノードはより高いスケジューリング優先度で実行し、決定的な f_s で公開する。
- マルチセンサ融合には時刻同期メッセージを使用する。メッセージにタイムスタンプを含め、ハードウェアタイムスタンプが利用可能な場合はそれを優先する。
- ROS1 では、ラッチングにより遅れて参加したサブスクライバが最後のメッセージを受信できる。ROS2 では、代わりに QoS ポリシー（信頼性、耐久性）を使用する。
- 高レートトピックではメッセージペイロードをコンパクトに保つ。冗長な診断テキストの代わりに配列またはコンパクトな構造体を送信する。
- ネームスペースングにより複数の肢体制御器を分離する。明確にするため `/left_arm/controller/command` のような階層的な名前を使用する。

実用的なコード例：関節状態をサブスクライブし、コントローラへ簡単な位置軌道を公開する最小の ROS2 Python ノード。QoS 選択と効率的なメッセージ構築を示す。

コードサンプル 48 ROS2 ノード：関節状態をサブスクライブし、簡単な関節軌道を公開

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
import threading

class JointCommander(Node):
    def __init__(self):
        super().__init__('joint_commander')

        # パラメータ宣言
        self.declare_parameter(
            'joint_names',
            ['hip', 'knee', 'ankle'],
            ParameterDescriptor(
                type=ParameterType.PARAMETER_STRING_ARRAY,
                description='制御対象の関節名リスト'
            )
        )
        self.declare_parameter(
            'control_rate',
            50.0,
            ParameterDescriptor(
                type=ParameterType.PARAMETER_DOUBLE,
```

```

        description='制御周波数[Hz]'
    )
)
self.declare_parameter(
    'step_size',
    0.01,
    ParameterDescriptor(
        type=ParameterType.PARAMETER_DOUBLE,
        description='1ステップあたりの目標位置増分[rad]'
    )
)

# QoS設定
qos = QoSProfile(
    depth=10,
    reliability=ReliabilityPolicy.RELIABLE,
    durability=DurabilityPolicy.VOLATILE
)

# 購読
self._sub = self.create_subscription(
    JointState,
    '/joint_states',
    self._joint_cb,
    qos
)

# 配信
self._pub = self.create_publisher(
    JointTrajectory,
    '/arm_controller/command',
    qos
)

# 状態変数
self._lock = threading.Lock()
self._current_positions = []
self._joint_names = self.get_parameter('joint_names').value

# タイマー

```

```

        period = 1.0 / self.get_parameter('control_rate').value
        self._timer = self.create_timer(period, self._publish_command)

    def _joint_cb(self, msg: JointState):
        with self._lock:
            # 名前順序を揃えて保存
            name_to_pos = dict(zip(msg.name, msg.position))
            self._current_positions = [name_to_pos.get(n, 0.0) for n in self._joint_names]

    def _publish_command(self):
        with self._lock:
            if not self._current_positions:
                return
            step = self.get_parameter('step_size').value
            target = [p + step for p in self._current_positions]

            traj = JointTrajectory()
            traj.joint_names = self._joint_names
            point = JointTrajectoryPoint()
            point.positions = target
            point.time_from_start.sec = 0
            point.time_from_start.nanosec = int(self._timer.timer_period_ns)
            traj.points = [point]
            self._pub.publish(traj)

def main(args=None):
    rclpy.init(args=args)
    node = JointCommander()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

運用上の影響：

- QoS と信頼性：アクチュエータコマンドではメッセージの消失を避けるため RELIABLE を選択する。高レートセンサでは損失トランスポートが許容される場合 BEST_EFFORT を使用する。
- メッセージタイミング：パブリッシャが正確なタイムスタンプを含めることを確認する。タイム

スタンプの誤ったセンサデータは状態推定を劣化させる。

- TF の使用：ベースリンクとエンドエフェクタフレームの変換を適切なレートでブロードキャストする。ネットワークが飽和している場合は TF 更新をダウンサンプリングする。
- 安全性：メッセージデッドラインを逃した際にアクチュエータを安全状態に設定するウォッチドッグを実装する。

ヒューマノイドシステムにおける設計のトレードオフとリスク：

- 遅延対スループット：公開レートを上げると制御遅延は減少するが、CPU およびネットワーク負荷が増加する。
- モノリシック対モジュラーノード：モノリシックノードはプロセス間遅延を減らす、保守性とフォルト隔離を制限する。
- メッセージサイズ：リッチメッセージはデバッグを簡素化するが、リアルタイムトピックを飢餓させる可能性がある。制御クリティカルチャネルではバイナリコンパクトメッセージを使用する。
- 同期失敗：同期されていないセンサタイムスタンプは誤ったセンサ融合を引き起こし、不安定なバランス制御を生じる可能性がある。

エンジニアは決定的な制御タイミング、メッセージ信頼性、計算負荷をバランスさせる必要がある。リソース競合が発生した場合は制御ループのタイムラインを優先し、ランタイム障害中の運用上のリスクを軽減するための階層化されたウォッチドッグを実装する。

13.4 ROS ツールによるデバッグ

ノード／トピックプリミティブと ROS2 インストール・設定手順を踏まえた実践的デバッグは、稼働中のシステムと安全で信頼できるヒューマノイドのギャップを埋める。デバッグはタイミング、メッセージ整合性、リソース競合、ヒューマノイドの安定性に最も影響するミドルウェア設定に焦点を当てる。

問題設定。ヒューマノイドロボットは数百 Hz の IMU、力・トルクセンサ、カメラ、関節状態フィードバックといった複数の高レートストリームを統合する。障害は制御デッドラインの逸脱、センサドロップアウト、または静かな型不一致として顕在化する。エンジニアリング目標はハードウェアへのリスクを最小限に抑えつつ迅速に根本原因を特定することである。

主要概念とツール。まず Quality of Service (QoS) を定義する：QoS はパブリッシャとサブスクライバ間の配信保証（信頼性、耐久性、履歴、深度、寿命）を記述する。パケットロスや過渡的過負荷下での動作を QoS で制御する。ライフサイクルノードはコントローラの決定論的起動・終了状態を提供する。コールバックグループとエグゼキュータはスレディング、ひいてはレイテンシと競合を決定する。

実践的診断ワークフロー。

1. 可視性とトポロジ。

- `ros2 node list` および `ros2 topic list` で実行中のノードとトピックを確認する。`rqt_graph` で接続を可視化し、期待されるパブリッシャ／サブスクライバエッジを確かめる。

2. メッセージ健全性と型。

- `ros2 topic info /topic_name` でメッセージ型とスキーマを確認する。よくあるエラーはパブリッシャとサブスクライバが異なるメッセージ定義または不一致したフィールド期待を使用していることである。

3. レートと帯域。

- `ros2 topic hz` および `ros2 topic bw` を使ってパブリッシュレートとサイズを測定する。ネットワークがデータ負荷を維持できるか計算する：

$$[H]B = f \times S, \quad (118)$$

ここで B は必要帯域、 f はパブリッシュ周波数、 S は平均メッセージサイズである。よって最大持続可能レートは

$$[H]f_{\max} = \frac{B_{\text{avail}}}{S}. \quad (119)$$

これらの式を使って分散モータコントローラ用に Ethernet または CAN リンクを次元決定する。

4. レイテンシとジッタ。

- 制御ループ安定性を評価するために到着時間ジッタを測定する。帯域 f_b の関節サーボループでは、サンプル周期 T_s は動特性を捉えるために $T_s \ll 1/(2f_b)$ を満たす必要がある。到着時間間隔の平均と標準偏差を計算してジッタを定量化する。

5. QoS 不一致とポリシーデバッグ。

- サブスクライバが静かにデータをドロップする場合は QoS プロファイルを調査する。たとえば信頼できる配信を要求するサブスクライバはベストエフォートパブリッシャには決して接続しないことがある。

6. システム時刻とシミュレーション時刻。

- シミュレーションとハードウェアを切り替える際に `use_sim_time` パラメータを確認する。時刻ソースの誤設定によりタイムスタンプベースコントローラが状態推定を誤計算する。

7. リソース競合とスレッディング。

- ブロッキングコールバックについてエグゼキュータとコールバックグループ設定をチェックする。長時間実行コールバックは別コールバックグループを使用するか非ブロッキングスレッドにオフロードすべきである。

8. 低レベルプロファイリングとトレーシング。

- クイックチェックには `ros2 topic echo`、トレースキャプチャには `ros2 bag record`、ROS2 C++ノードのカーネル・ミドルウェアレベルレイテンシキャプチャには `ros2_tracing` と `LTTng` を使用する。

具体的測定例。軽量サブスクライバを使って `/joint_states` のタイムスタンプギャップとジッタを測定する。これによりセンサまたはネットワークジッタが制御許容値を侵害するかを明らかにする。

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
```

```

from sensor_msgs.msg import JointState
from typing import List, Optional
import statistics
import time

# QoS設定：センサデータ用の信頼性・履歴ポリシー
QOS_PROFILE = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=10
)

class JitterMonitor(Node):
    def __init__(self) -> None:
        super().__init__('jitter_monitor')
        self._sub = self.create_subscription(
            JointState, '/joint_states', self._cb, QOS_PROFILE)
        self._prev_ns: Optional[int] = None
        self._deltas: List[float] = []
        self._window_size = 200 # 統計計算ウィンドウ
        self._logger = self.get_logger()

    def _cb(self, msg: JointState) -> None:
        now_ns = self.get_clock().now().nanoseconds
        if self._prev_ns is not None:
            dt = (now_ns - self._prev_ns) * 1e-9
            self._deltas.append(dt)
            if len(self._deltas) >= self._window_size:
                mean = statistics.fmean(self._deltas)
                std = statistics.stdev(self._deltas)
                self._logger.info(f'avg={mean:.6f}s, std={std:.6f}s')
                self._deltas.clear()
            self._prev_ns = now_ns

def main(args=None) -> None:
    rclpy.init(args=args)
    node = JitterMonitor()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:

```

```
pass
finally:
    node.destroy_node()
    rclpy.shutdown()
```

ターゲットを絞った修正と検証。

- ジッタが設計限界を超える場合、コントローラプロセスの優先度を上げ、非必須ロギングを削減するか、リアルタイムカーネルで専用コアにコントローラを分離する。
- メッセージがドロップされる場合、QoS ポリシーを調整し、深度を増やすか、ネットワーク品質に応じて信頼性設定を変更する。
- 型非互換が現れる場合、影響を受けるパッケージをリビルドし、ABI 互換メッセージ定義を使用する。

高度な技法。

- ネイティブ C++ ノードのクラッシュキャプチャには gdb を、メモリエラー検出には valgrind または AddressSanitizer を使用する。
- 繰り返し可能なテストのために ros2 bag を使ってセンサストリームをコントローラに再生する。
- ros2 ライフサイクル遷移を使って安全な起動と段階的障害注入をステージングする。

エンジニアリングへの影響、トレードオフ、運用上のリスク。

- QoS 信頼性またはメッセージ深度を上げるとメモリとネットワーク帯域を消費する；高レートセンサにはバランスが必要である。
- リアルタイムスケジューリングはレイテンシを減らす但デバッグを複雑化する；カーネルとドライバの相互作用が問題を隠蔽することがある。
- 広範なロギングはデバッグに役立つが、断続的障害を隠すタイミング摂動を引き起こすリスクがある。
- アクチュエータ損傷または人の安全を回避するため、フィールド配備前に必ずハードウェアインザループ環境で修正を検証する。

14 ビヘイビアツリーの基礎

14.1 ヒューマノイドロボティクスにおける意思決定の基礎

前述の ROS 統合と GROOUT によるビヘイビアツリーの議論を踏まえ、ここでは高レベルタスクをリアルタイムのヒューマノイド制御にマッピングするためのコアな意思決定プリミティブに焦点を当てる。これらのプリミティブは ROS トピック、サービス、およびシミュレーションと実機で用いられる実行モデルと互換性を持つ必要がある。

問題定義。ヒューマノイドロボットは、モジュラーで予測可能かつ中断可能な意思決定アーキテクチャを必要とする。このアーキテクチャは、安全性制約の下で移動、操縦、バランス、知覚を調整しなければならない。ビヘイビアツリー (BT) は、リアルタイムティックセマンティクスを保ちながら

そのような振る舞いを符号化するための構造化され再利用可能なアプローチを提供する。

技術分析。ビヘイビアツリーは、各内部ノードが制御フローを統治し、各リーフがセンシングまたはアクチュエーションを実行する、根付き有向非巡回グラフである。リーフノードは以下である：

- コンディション：SUCCESS または FAILURE を返す瞬時のチェック。
- アクション：RUNNING、SUCCESS、または FAILURE を返す可能性があるランタイム効果を持つコマンド。

コンポジットノードは制御ロジックを実装する：

- シーケンス：ある子が FAILURE または RUNNING を返すまで左から右へ子を実行する。
- フォールバック（セレクト）：ある子が SUCCESS または RUNNING を返すまで左から右へ子を実行する。
- パラレル：子を並行して実行し、閾値ポリシーで結果を集約する。
- デコレータ：単一の子をラップして振る舞いを変更する（タイムアウト、繰り返し、反転）。

形式セマンティクスはティック関数 τ によって簡潔に表現される。ツリー T と環境状態 x に対して、各ティックは $\tau(T, x)$ を呼び出し、ステータス $s \in \{\text{SUCCESS}, \text{FAILURE}, \text{RUNNING}\}$ を返す。子 C_1, \dots, C_n を持つシーケンスノードに対して、独立性仮定の下でのコンポジット成功確率は

$$[H]P_{\text{seq}} = \prod_{i=1}^n p_i, \quad (120)$$

である。ここで p_i は子 i が到達された際に最終的に SUCCESS を返す確率である。フォールバックノードに対しては

$$[H]P_{\text{fb}} = 1 - \prod_{i=1}^n (1 - p_i). \quad (121)$$

リアルタイムシステムではコストとレイテンシを考慮しなければならない。子 i が期待 CPU 時間 c_i を消費し、以前の子がすべて成功した場合にのみ到達される場合、シーケンスの期待累積コストは

$$[H]\mathbb{E}[C_{\text{seq}}] = \sum_{k=1}^n c_k \prod_{j=1}^{k-1} p_j. \quad (122)$$

これらの式は順序付けを導く：低コストで高失敗率のチェックを最初に配置して平均実行コストを削減する。

実装上の考慮事項。ヒューマノイドでは、各アクションフリーフを具体的なコントローラまたは ROS アクションサーバに結びつける。センサ状態（IMU、ジョイントエンコーダ、ビジョン）を融合してコンディションを実装する。安全性のためにデコレータを用いる：最大トルクコマンドを制限し、非常停止伝播を強制し、脚の動作の前にバランスチェックを挿入する。

ティックスケジューリングと並行性は重要である。典型的な設計：

- バランス関連サブツリーのための高周波ルートティック（50–200 Hz）。
- プランニングまたはビジョンサブツリーのための低周波ティック。
- パラレルノードを用いて高周波安定化を低周波プランニングから隔離する。

例：モジュラーリーフを持つヒューマノイドのピックアップタスク。簡略化されたビヘイビアツ

リー実行は Python でティックセマンティクスを示し、アクションノードが ROS またはシミュレーションとどのようにインターフェースするかを示す。

コードサンプル 50 ヒューマノイドピックタスクのための最小ビヘイビアツリーノードクラス

```
from __future__ import annotations
import enum
from typing import Callable, List, Optional, Protocol

class Status(enum.IntEnum):
    """動作の返り値"""
    SUCCESS = 0
    FAILURE = 1
    RUNNING = 2

class Node(Protocol):
    def tick(self) -> Status: ...

ConditionFn = Callable[[], bool]
StartFn     = Callable[[], None]
UpdateFn    = Callable[[], Status]

class Condition(Node):
    """条件判定ノード"""
    def __init__(self, check: ConditionFn) -> None:
        self._check = check

    def tick(self) -> Status:
        return Status.SUCCESS if self._check() else Status.FAILURE

class Action(Node):
    """ROSアクションラッパー"""
    def __init__(self, start: StartFn, update: UpdateFn) -> None:
        self._start    = start
        self._update    = update
        self._started  = False
```

```

def tick(self) -> Status:
    if not self._started:
        self._start()
        self._started = True
    return self._update()

def reset(self) -> None:
    self._started = False

class Sequence(Node):
    """子を順番に実行"""
    def __init__(self, children: List[Node]) -> None:
        self._children = children
        self._index = 0

    def tick(self) -> Status:
        while self._index < len(self._children):
            status = self._children[self._index].tick()
            if status == Status.RUNNING:
                return Status.RUNNING
            if status == Status.FAILURE:
                self._index = 0
                return Status.FAILURE
            self._index += 1
        self._index = 0
        return Status.SUCCESS

# -----
# 以下は利用例（実際のROSノードでは別ファイルに配置）
# -----
def check_visibility() -> bool: ...
def imu_balance_check() -> bool: ...
def plan_trajectory_start() -> None: ...
def plan_trajectory_update() -> Status: ...
def execute_motion_start() -> None: ...
def execute_motion_update() -> Status: ...
def close_gripper_start() -> None: ...

```

```
def close_gripper_update() -> Status: ...
```

```
def build_pick_tree() -> Node:  
    return Sequence([  
        Condition(check_visibility),  
        Condition(imu_balance_check),  
        Action(plan_trajectory_start, plan_trajectory_update),  
        Action(execute_motion_start, execute_motion_update),  
        Action(close_gripper_start, close_gripper_update),  
    ])
```

テストと検証。Isaac Sim と GROOT を用いてティック、タイミング、センサノイズをシミュレートする。ツリーをモデル化された知覚と駆動ノイズの下でモンテカルロシミュレーションを実行することで、成功確率を実験的に検証する。診断の優先順位付けとフォールバック順序の調整のために、式(120)–(122)を用いる。

エンジニアリングへの影響、トレードオフ、リスク：

1. 子の順序付けは平均レイテンシと失敗露出に影響する；高速失敗チェックを最初に配置する。
2. 確率式の独立性仮定はほとんど成立しない；相関失敗（センサドロップアウト、電源障害）は単純な見積もりを無効にする。
3. ティックレートのミスマッチは古いデータの使用を引き起こす；高帯域安定化ループをプランニングノードから分離したサブツリーまたは優先ティックで隔離する。
4. デコレータは安全性を実装するが複雑さを追加する；安全性クリティカルな振る舞いには厳密な形式検証または境界テストが必要である。
5. 並行実行は CPU および通信負荷を増加させる；アクチュエータコントローラのリアルタイム保証を確保する。
6. シミュレーションから実機へのミスマッチは安全でない振る舞いを引き起こすリスクがある；常にセンシングベースのハードストップとウォッチドッグタイマを含める。

設計トレードオフは、モジュラー性、予測可能性、計算制限のバランスを取らなければならない。実環境へのヒューマノイド展開時には、各サブツリーの最初に安全性デコレータと低コストチェックを優先する。

14.2 GROOT でのビヘイビアツリーの作成

前小節の決定理論的概念を踏まえ、人型ロボット向けに前提条件、アクション、リカバリ戦略を GROOT 内で実行可能なツリーに変換する方法を示す。焦点は実践的なものであり、原子スキルをモデル化し、センサとアクチュエータをブラックボード変数にマッピングし、ロボット上のミドルウェアに直接結びつくツリーをエクスポートすることである。

問題定義と運用コンテキスト。人型ロボットのタスクは、移動、知覚、操縦のきめ細かく連携したシーケンスを必要とする。単一のアクチュエータ故障や知覚ミスが連鎖して転倒や物体落下を引き起こす可能性がある。目標は以下を実現するビヘイビアツリー (BT) である：

- 安全な前提条件（バランス、到達可能な物体）をカプセル化する。
- 原子スキル（歩行、到達、把持）を実行する。
- 決定論的なフォールバック（バランスリカバリ、再取得）を提供する。

技術分析：ノード、成功合成、タイミング。GROOT は標準 BT ノードタイプのビジュアル構成を提供する：

- Sequence：子を順番に実行し、一つでも失敗すると停止する。
- Selector (Fallback)：子を順に試行し、一つでも成功すると停止する。
- Condition：センサ／ブラックボード状態に対する瞬時チェック。
- Action：コントローラまたはスキルを呼び出し、成功／実行中／失敗を返す。
- Decorator：タイムアウトやリトライを強制する。

信頼性の確率解析のため、各アクションの独立成功確率を p_i と定義する。すると：

$$[H]P_{\text{seq}} = \prod_{i=1}^n p_i \quad (123)$$

n 個のアクションを持つ Sequence に対して。成功確率 p_i の子を持つ Selector に対しては：

$$[H]P_{\text{sel}} = 1 - \prod_{i=1}^m (1 - p_i). \quad (124)$$

これらの式はエンドツーエンドの頑健性を工学的に見積もり、冗長化やリトライを追加すべき箇所を優先付けるのに役立つ。

人型ルーチンの設計パターン。GROOT で作成する前に以下の手順を用いる：

1. タスクを原子スキルに分解する。例：NavigateTo、Step、Reach、Grasp、Retreat、BalanceRecovery。
2. 前提条件と事後条件を Condition として定義する。例前提条件：バランスを要するアクションでは imu_stable が true。
3. 共有データ用のブラックボード変数を指定：ターゲットポーズ、知覚物体フレーム、歩行計画ハンドル、スキルステータス。
4. 実行セマンティクスを特定：ティックレート、タイムアウト、リトライ回数。

具象ツリーアーキテクチャ（テキスト概要）。頑健なピック動作は通常以下を用いる：

- トップレベル Selector：
 - Sequence: Preconditions -> NavigateTo -> ReachAndGrasp -> Retreat
 - BalanceRecovery デコレータ -> Retry Navigate and Grasp

この構造は不安定を検知するとリカバリノードを実行し、Selector が代替戦略を許容する。

実装：GROOT からロボットミドルウェアへの接続。GROOT はツリーを JSON または BT XML でエクスポートする。エクスポートされたツリーはランタイムで実装されたアクションおよび条件ノードを参照しなければならない。ランタイムは：

- ブラックボードサービスまたはトピックを提供する。
- ティックコマンドを受け付けステータスを返すアクションサーバを実装する。
- センサ状態を制御レートでブラックボード変数に公開する。

例：エクスポートされた GROOT ツリーが呼び出す Condition と Action を実装する最小 ROS2 ブリッジ。このノードは：

- 知覚を購読し object_visible を設定する。
- 把持コマンドを発行し成功を報告する簡単な把持アクションサーバを実装する。

コードサンプル 51 ROS2 ノード：GROOT ブラックボードとスキルをブリッジ（最小例）

```
#!/usr/bin/env python3
import json
from typing import Dict

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from std_msgs.msg import Bool, String
from geometry_msgs.msg import PoseStamped

class GrootBridge(Node):
    """Groot_ブラックボード同期・スキル指令ブリッジ"""

    def __init__(self) -> None:
        super().__init__("groot_bridge")

        # QoS: 信頼性重視で transient_local（後からSubscribeしても最新を得る）
        bb_qos = QoSProfile(
            depth=1,
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,
        )

        # ブラックボード更新購読
        self.create_subscription(
            Bool, "perception/object_visible", self._percept_cb, 10
        )

        # ブラックボード公開（Groot モニタ等が参照）
        self._bb_pub = self.create_publisher(String, "blackboard", bb_qos)

        # 把持指令
```

```

self._grasp_pub = self.create_publisher(PoseStamped, "cmd/grasp_pose", 10)

# スキル状態購読
self.create_subscription(
    String, "skill/grasp/status", self._grasp_status_cb, 10
)

# 内部ブラックボード
self._blackboard: Dict[str, bool] = {"object_visible": False}
self._publish_blackboard() # 初回送信

# --- callbacks -----
def _percept_cb(self, msg: Bool) -> None:
    self._blackboard["object_visible"] = bool(msg.data)
    self._publish_blackboard()

def _grasp_status_cb(self, msg: String) -> None:
    # 必要に応じて BT へ転送（本実装ではログ出力のみ）
    self.get_logger().debug(f"Grasp_status: {msg.data}")

# --- public API -----
def request_grasp(self, pose: PoseStamped) -> None:
    """外部スキルノードから呼ばれる把持指令"""
    self._grasp_pub.publish(pose)

# --- internal -----
def _publish_blackboard(self) -> None:
    # JSON 形式で送信（Groot 側でパースしやすく）
    self._bb_pub.publish(String(data=json.dumps(self._blackboard)))

def main(args=None) -> None:
    rclpy.init(args=args)
    node = GrootBridge()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()

```

`rcldpy.shutdown()`

タイミングと安定性に関する工学的注意。ティック周波数 f を選び、コントローラが期待タイムアウト内で完了できるようにする。アクションの平均実行時間を τ とすると、タイムアウト $T \approx k\tau$ ($k \in [1.5, 3]$) を設定する。短すぎるタイムアウトは誤った失敗を引き起こし、長すぎるタイムアウトはリカバリを遅延させる。

実践的ガイドラインとトレードオフ：

- 原子スキルの粒度：小さなスキルは再利用性と診断性を高めるが、通信オーバーヘッドを増やす。
- センサチェックは Condition で実施し、コストの高いモータシーケンスは Action に留める。
- Selector の冗長性と複雑性のバランス：選択肢を増やすと (124) に従い成功確率は上昇するが、ブラックボードで状態爆発を引き起こす。
- 安全第一：常に最上位 Selector に高優先度 BalanceRecovery ノードを追加する。ハードウェアキルスイッチが BT 駆動モータに優先することを確認する。

運用上のリスク：

- エクスポートされたノード名とランタイム実装のセマンティクスミスマッチは静かに失敗する；ノード ID と実装エンドポイントのマッピングレジストリを維持する。
- プロセス間でブラックボードが不整合になると競合状態が発生する；共有キーにタイムスタンプまたはバージョンカウンタを用いる。
- 検証なしの知覚への過度の依存は危険な動作を引き起こす；検証ループと保守的タイムアウトを含める。

設計者は Isaac Sim でシミュレーション内を繰り返し、タイミングと故障モードを検証し、ハードウェアへ段階的に展開すべきである。GROOT のビジュアルオーサリングとミドルウェアブリッジの組み合わせは、複雑な人型ルーチンに対して保守可能でテスト可能なビヘイビアをもたらす。

14.3 ツリーのデバッグと最適化

これまでに議論した GROOT ノードセマンティクスと意思決定プリミティブを基に、ヒューマノイド動作ツリーのランタイム障害診断と実行レイテンシ削減に焦点を当てる。実用的なデバッグには、バランス・歩行・操縦タスクに典型的なリアルタイム制約を満たすための定量的計測と構造的最適化の両方が必要である。

問題定義. ヒューマノイドコントローラは複数の動作ツリーを同時に実行することが多い。単一の低速ノードがループタイミングを侵害し、センサ処理の遅延や安全でないアクチュエータ指令を引き起こす。よくある障害モードは以下の通り：

- ブロッキング I/O や重い計算による無制限ノード実行時間；
- 古い状態を保持するメモリフルデコレータの誤用；
- フォールバック／シーケンス順序での優先度逆転；
- 知覚ノードと制御ノード間のリソース競合。

実行コストをモデル化することで、どこに努力を投資すべきかが明らかになる。ツリーティック周

期予算を T_{budget} とする。子実行時間 t_i を持つシーケンスノードの最悪実行時間は

$$T_{\text{seq}}^{\max} = \sum_{i=1}^n t_i. \quad (125)$$

子成功確率 p_i と実行時間 t_i を持つフォールバックノードの期待ティック時間は

$$\mathbb{E}[T_{\text{fb}}] = \sum_{i=1}^n \left(\prod_{j=1}^{i-1} (1 - p_j) \right) t_i, \quad (126)$$

その成功確率は $1 - \prod_{i=1}^n (1 - p_i)$ である。これらの式は子の並び替え、結果のキャッシュ、高価なチェックの前後移動に関する判断を導く。

計測と測定. ツリーティック内部での軽量プロファイリングから始める。測定項目：

- ノードごとの平均・最大実行時間；
- 成功・失敗回数で p_i を推定；
- 非同期ノードのメモリ割当とスレッド使用状況。

GROOT または ROS パラメータからランタイムで切替可能なロギングレベルを用いる。ノード入室時にタイムスタンプを記録。外乱値に反応しないようスライディングウィンドウでデータを集計する。

最適化戦略. 以下の対象介入を適用し、 T_{budget} に対する影響を測定する。

1. 期待コストに基づき子を並び替える。フォールバックノードでは、高成功・低コストチェックを前に配置して $\mathbb{E}[T_{\text{fb}}]$ を削減。早期失敗が多いシーケンスでは、失敗しそうなガードを先に配置して長いアクションを短絡。
2. ブロッキング呼び出しを非同期サービスに置換。重い知覚前処理を別スレッドまたは専用ノードにオフロードし、結果をブラックボードに公開。
3. 決定的計算をキャッシュ。ティックレートより速く変化しないセンサ融合出力には、生存時間付き短命キャッシュを用いる。
4. ウォッチドッグとタイムアウトを追加。ノードごとに上限 t_{\max} を実装。ノードが t_{\max} を超えたら安全なステータス（Failure または Running）を返しアラートを上げる。回路遮断パターンで不調サブツリーを無効化。
5. デコレータのステートフル性を最小化。決定論と再現性が要求される場合はステートレスガードを優先。メモリフルデコレータが必要な場合は、プリエンプションやツリーリコンフィギュレーション下でのリセットセマンティクスを文書化・テスト。
6. 並列ノードを慎重にプロファイル。並列実行は制御を高速化できるが同期障害を導入。並列時間を最大値+調整オーバーヘッドで置換して最悪ブロッキングをモデル化。
7. 滅多に使われない分岐を刈り込み・リファクタ。運用上の利益が小さいのに CPU を消費したり検証複雑性を増やす分岐を削除。

自動プロファイリングスニペット. 以下のリストは、GROOT スタイルツリーや `py_trees` に適した実用的計測フックを示す。ノードごとのタイミングをログし、シンプルなタイムアウトベース保護を適用する。

```

import time
import logging
from typing import Dict, Any, Optional

# ROS 2 用のインポート（必要に応じてコメントアウトを外す）
# import rclpy
# from rclpy.node import Node

# タイムアウト閾値（秒）
TIMEOUT: float = 0.05

# 統計情報を保持する辞書
stats: Dict[str, Dict[str, Any]] = {}

def profile_tick(node) -> str:
    """
    ノードの実行時間を計測し、統計情報を更新する。
    タイムアウトを超えた場合は警告を出してノードを中断する。
    """
    start: float = time.perf_counter()
    status: str = node.tick() # ノードの実行
    elapsed: float = time.perf_counter() - start

    # 統計情報の更新
    s: Dict[str, Any] = stats.setdefault(node.name, {
        'count': 0,
        'total': 0.0,
        'max': 0.0,
        'succ': 0
    })
    s['count'] += 1
    s['total'] += elapsed
    s['max'] = max(s['max'], elapsed)
    if status == 'SUCCESS':
        s['succ'] += 1

# タイムアウト検出

```

```

    if elapsed > TIMEOUT:
        logging.warning(
            f"Node_{node.name}_exceeded_timeout:{elapsed:.3f}s>_{TIMEOUT}s"
        )
        # ノードに中断メソッドがあれば呼び出す
        if hasattr(node, 'abort') and callable(getattr(node, 'abort')):
            node.abort()

    return status

def reset_stats() -> None:
    """統計情報をリセットする"""
    global stats
    stats.clear()

def get_stats(node_name: Optional[str] = None) -> Dict[str, Any]:
    """
    指定されたノードの統計情報を返す。
    node_nameがNoneの場合は全ノードの統計を返す。
    """
    if node_name is None:
        return stats
    return stats.get(node_name, {})

# 使用例 (ROS 2 ノード内での利用を想定)
# class ProfiledBehaviorTree(Node):
#     def __init__(self):
#         super().__init__('profiled_behavior_tree')
#         self.tree = ... # ビヘイビアツリーの構築
#
#     def tick(self):
#         for node in self.tree.nodes:
#             profile_tick(node)

```

運用テストとメトリクス. 変更適用後：

- 95 パーセンタイルティック時間が T_{budget} 未満であることを検証；
- エンドツーエンドタスク成功率がミッション要求を満たすことを確認；

- ・センサ遅延／故障を注入したフォルトインJECTIONテストを実行し、ツリーの頑健性を観測。

設計トレードオフとリスク。レイテンシ最適化は意思決定ノードの熟慮能力を低下させる可能性がある。積極的キャッシングは知覚駆動アクションの陳腐化リスクを増大。並列化は競合状態デバッグの複雑性を高める。タイムアウトは断続的だが正当な長時間計算を隠蔽しないよう調整されなければならない。

ヒューマノイドシステムへの具体的影響：

- ・バランス・歩行ループでは、厳格な T_{budget} 予算を厳守；ハードタイムアウトとティックごとの最小計算を適用。
- ・操縦タスクでは、重い計画を別スレッドに移し、ツリー内ステータスマニタで結果を安全に統合。
- ・ミッションクリティカル展開では、(126) の楽観的平均より (125) の予測可能な最悪境界を優先。

エンジニアは、応答性・信頼性・計算コストを天秤にかけながらツリー構造を変更する必要がある。誤った最適化は歩行・把握・人間相互作用で巧妙な障害を生む可能性がある。継続的プロファイリングと保守的タイムアウトは、ヒューマノイドの安全性とミッション成功を保ちながら運用リスクを削減する。

14.4 ツリー設計における高度なパターン

GROOT における実践的なツリー構築と前述のデバッグ戦略を踏まえ、本小節ではヒューマノイド制御の堅牢性、反応性、モジュール性を高める高度な構造パターンを示す。焦点は運用上の重要性にある：タイミング制約を満たすシーケンス、バランス喪失後のフォールバック戦略、タスク切り替えのための動的サブツリーである。

問題定義：ヒューマノイドコントローラは、高速な反射的な反応と長期的な熟慮的計画を組み合わせなければならない。単純なツリーは反応をブロックするかロジックを重複させる。高度なパターンは、制御ロジックを並行性、メモリ、仲裁、実行時の安全な挙動の挿入／削除をサポートするよう構造化することでこれを解決する。

技術分析とパターン

- ・k-of-n 成功による並列合成：複数の推定器や冗長モジュールがアクションを検証する場合に有用。満場一致を要求するのではなく、 n 個のモニタのうち k 個の成功閾値を設定する。センサが不一致の場合、 k 個が一致したときにのみコントローラは継続する。 n を子モニタの数、 k を必要な成功数とする。実用的な式は $k = \lceil pn \rceil$ とし、 p は要求信頼度分数である。

$$\lceil H \rceil k = \lceil pn \rceil \quad (127)$$

p を選択することは偽陽性と遅延アクションの間でトレードオフである。

- ・メモリ付きフォールバック（優先リカバリ）：最後に成功した子を記憶するフォールバックを実装する。ヒューマノイドがバランスを失った場合、高優先度のリカバリサブツリーが歩行をプリエンプトし、以前に実行していたタスクに制御を戻す。メモリはフォールバック選択間の振動を回避する。
- ・リアクティブシーケンス：単一ノード内で低遅延リアクティブブランチと熟慮プランナを組み合わせる。リアクティブサブノードは毎ティック実行され、SUCCESS を返せばプランナをプリエ

ンプトする。このパターンは歩行中の足配置修正に不可欠である。

- 時間的・安全制約のためのデコレータスタック：タイムアウト、レート制限、インバータ、リトライデコレータを適用する。タイムアウトデコレータは安全実行時間を超えたプランを停止する。バックオフ付きリトライデコレータは繰り返しアクチュエータストレスからハードウェアを保護する。
- ブラックボード駆動モジュール性：感覚状態、意図、サブツリー間交渉のための共有ブラックボードを使用する。競合状態を避けるため読み書きアクセスをカプセル化する。センサフュージョンで陳腐データを検出するためバージョンングまたはスタンプ付きエントリを使用する。
- 動的サブツリーの挿入・削除：ターゲットオブジェクトが識別されたときの操作など、ロール固有の挙動を実行時に追加することをサポートする。動的サブツリーはリソース（アクチュエータ、センサストリーム）を登録し、競合を回避するためクリーンに登録解除しなければならない。
- ソフト優先度のための効用ベース仲裁：複数タスクが競合する場合、各候補サブツリーに対して効用スコア u_i を計算する。ボルツマン分布によるソフト選択は飢餓を回避する：

$$[H]P(i) = \frac{\exp(\beta u_i)}{\sum_j \exp(\beta u_j)} \quad (128)$$

β を調整することでマルチタスクヒューマノイドコントローラにおける探索と活用のバランスを調節する。

実装スケッチ：動的挿入と k-of-n 並列ポリシーのための最小ランタイム

コードサンプル 53 k-of-n 並列と動的サブツリー挿入を示す最小ビヘイビアツリーランタイム

```
from __future__ import annotations
import logging
from dataclasses import dataclass, field
from enum import Enum, auto
from typing import Dict, List, Optional, Protocol, runtime_checkable

# ロギング設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class Status(Enum):
    SUCCESS = auto()
    RUNNING = auto()
    FAILURE = auto()

@runtime_checkable
class Node(Protocol):
```

```

def tick(self, blackboard: Dict) -> Status: ...

@dataclass
class Parallel(Node):
    children: List[Node]
    k: int = field(init=False)

    def __post_init__(self):
        n = len(self.children)
        if n == 0:
            raise ValueError("子ノードが空です")
        self.k = n # デフォルトは全成功

    @classmethod
    def with_ratio(cls, children: List[Node], p: float) -> "Parallel":
        inst = cls(children)
        inst.k = max(1, int(p * len(children)))
        return inst

    def tick(self, blackboard: Dict) -> Status:
        successes = 0
        for child in self.children:
            status = child.tick(blackboard)
            if status == Status.SUCCESS:
                successes += 1
            elif status == Status.RUNNING:
                return Status.RUNNING
        return Status.SUCCESS if successes >= self.k else Status.FAILURE

@dataclass
class Sequence(Node):
    children: List[Node] = field(default_factory=list)

    def tick(self, blackboard: Dict) -> Status:
        for child in self.children:
            status = child.tick(blackboard)
            if status != Status.SUCCESS:
                return status

```

```
return Status.SUCCESS
```

```
@dataclass
```

```
class DynamicManager(Node):
```

```
    root: Sequence = field(default_factory=Sequence)
```

```
    def inject(self, subtree: Node) -> None:
```

```
        self.root.children.append(subtree)
```

```
        logger.info("Subtree injected")
```

```
    def remove(self, subtree: Node) -> None:
```

```
        try:
```

```
            self.root.children.remove(subtree)
```

```
            logger.info("Subtree removed")
```

```
        except ValueError:
```

```
            logger.warning("指定されたSubtreeは存在しません")
```

```
    def tick(self, blackboard: Dict) -> Status:
```

```
        return self.root.tick(blackboard)
```

```
@dataclass
```

```
class CheckBalance(Node):
```

```
    threshold: float = 0.5
```

```
    def tick(self, blackboard: Dict) -> Status:
```

```
        stability = blackboard.get("stability", 0.0)
```

```
        return Status.SUCCESS if stability > self.threshold else Status.FAILURE
```

```
@dataclass
```

```
class PlanFootstep(Node):
```

```
    def tick(self, blackboard: Dict) -> Status:
```

```
        # 実際の計画処理は非同期で複数tickを要する
```

```
        return Status.RUNNING
```

```
def main() -> None:
```

```
    blackboard: Dict[str, float] = {"stability": 0.6}
```

```

p = Parallel.with_ratio(
    [CheckBalance(), CheckBalance(), CheckBalance()], p=0.66
)
mgr = DynamicManager()
mgr.inject(p)
mgr.inject(PlanFootstep())
print(mgr.tick(blackboard))

if __name__ == "__main__":
    main()

```

設計への影響、トレードオフ、運用上のリスク

- **トレードオフ：**

1. 並行性は反応性を向上させるが同期複雑性を増加させる。
2. p （式 1）を下げると判断が高速化されるが誤作動リスクが上昇する。
3. 効用仲裁はハード優先度を回避するが効用が変動すると非決定的挙動を生じる可能性がある。

- **エンジニアリング上の考慮事項：**

1. ブラックボードエントリにタイムスタンプを付け陳腐センサデータを拒否する。
2. デコレータスタックの深さを制限し診断の不透明性を回避する。
3. 動的サブツリーライフサイクルを単体・統合テストで検証する。

- **運用上のリスク：**

1. 複数の挿入挙動が同一アクチュエータを主張する際のリソース競合。
2. フォールバックメモリが制御を早急に戻した場合の振動。
3. 安全臨界タイムアウトはハードウェアレベルウォッチドッグを持たなければならない。

これらのパターンを用いて、ヒューマノイドロボットにおいて応答性、モジュール性、安全性をバランスさせたビヘイビアツリーを設計せよ。

15 ソフトウェアとハードウェアの統合

15.1 ROS を用いたリアルタイム制御

前述の実用的な ROS セットアップとビヘイビアツリーによる協調は、高レベルの意思決定のためのフレームワークを構築する。リアルタイム制御は、バランスと移動のための決定論的なモータおよびセンサタイミングを強制することでループを閉じる。本小節では、ヒューマノイド制御のデッドラインを満たすために必要なハードタイミング要件、ミドルウェアの選択、実装パターンについて扱う。

ヒューマノイドの問題定義。ヒューマノイドは複数のネストされた制御ループを必要とする。低レベルのサーボループはトルク制御のために 1-2 kHz で動作する。ミドルレベルの全身コントローラはバランスと接触スケジューリングのために 100-500 Hz で動作する。高レベルのプランナーとビヘイ

ピアツリーは 10–50 Hz で動作する。エンジニアリングの目標は、低およびミドルレベルループの応答時間を保証しジッタを抑えつつ、高レベルでは柔軟性を保つことである。

技術的分析。決定性は、計算、通信、OS スケジューリングという 3 つの遅延要因を制御することを要求する。制御周期 T とロボットの制御帯域 f_c を定義する。保守的なサンプリング制約は

$$[H]T \leq \frac{1}{10f_c}, \quad (129)$$

であり、これはサンプリングが閉ループ帯域の少なくとも 10 倍速いことを保証する。ジッタ σ は T と比べて小さく保たなければならない。実用的な境界は $\sigma \ll T/10$ である。最悪遅延 L_{\max} は、エイリアシングと位相クリティカルなコントローラのデッドラインミスを回避するため、 $L_{\max} + \sigma < T/2$ を満たさなければならない。

ミドルウェアおよび OS の選択。リアルタイム対応 DDS 実装を備えた ROS2 が推奨される。*deadline*、*latency_budget*、*reliability* のような厳格な Quality of Service (QoS) 制御をサポートする DDS を用いる。OS レイヤでは、制御プロセスをリアルタイムポリシー (*SCHED_FIFO*) で実行し、予約コアに CPU アフィニティを設定する。スケジューリング遅延を削減するため *PREEMPT_RT* またはリアルタイムカーネルを用いる。アクチュエータネットワークでは、リアルタイム向けに設計されたフィールドバスプロトコル (*EtherCAT*、適切なスケジューラを備えた *CAN_FD*、または専用サーボバスハードウェア) を選択する。最も高速なインナーループは可能であればマイクロコントローラまたはモータコントローラにオフロードする。

設計パターンと実装チェックリスト：

- ソフトウェアをリアルタイムプロセスと非リアルタイムプロセスに分割する。センシング、低レベル制御、状態推定をリアルタイムパーティションに留める。
- リアルタイムコードで使用するすべてのメモリを事前に割り当てる。実行中に割り当てを行う *malloc/new* および STL コンテナを避ける。
- リアルタイムと非リアルタイムコンテキスト間でロックフリーデータ受け渡し（リングバッファ、ダブルバッファリング）を用いる。
- DDS/ROS2 QoS を設定する：制御周期に *deadline* を設定し、状態トピックには *reliability = RELIABLE* を使用し、メモリを制限するため適切な深度で *history=KEEP_LAST* を設定する。
- センサとアクチュエータ間でクロックを PTP (IEEE 1588) で同期し、タイムスタンプの整合を保证する。
- 複数レベルでウォッチドッグを適用し、デッドラインミスを検出して安全状態に遷移する。

制御安定性への影響。通信遅延 L とジッタ σ が制御ループに入ると、実効位相余裕が減少する。プラントを線形化し、遅延を e^{-sL} としてモデル化する。遅延余裕解析またはループシェイピングを用いて頑健性を確保する。必要な遅延が安定性余裕で許容されるより大きい場合、ループ帯域幅を下げるか、ループをアクチュエータに近いハードウェアに移す。

実装例：C++による最小限の ROS2 リアルタイムノードスケルトン。このノードは固定周期タイミング、決定論的起床のための POSIX *sleep*、事前割り当て済みリアルタイムパブリッシャを示す。コメントはクリティカルなリアルタイム選択を説明する。

```
class RealtimeController : public rclcpp::Node public: explicit Real-
```

```

timeController(const rclcpp::NodeOptions options = rclcpp::NodeOptions()) :
Node("realtime_controller", options), period_ns(1'000'000)/1msrt_pub_std::make_unique < realtime_tools::RealtimePublisher<
// メモリロック：ページフォルト回避 if (mlockall(MCL_CURRENT|MCL_FUTURE) ==
-1)RCLCPP_FATAL(this->get_logger(), "mlockall failed"); rclcpp::shutdown();
void run() configureThread();
struct timespec next; clock_gettime(CLOCK_MONOTONIC, next);
while (rclcpp::ok()) next.tv_nsec += period_ns; tsNorm(next); clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next, nullptr);
const float tau = computeTorque(); publish(tau);
private: static constexpr size_t CPU_CORE = 1; static constexpr int RTPRIO = 80;
const long period_ns_std::unique_ptr < realtime_tools::RealtimePublisher < std_msgs::msg::
Float64 >> rt_pub,
void configureThread() cpu_set_t cpuset; CPU_ZERO(&cpuset); CPU_SET(CPU_CORE, &cpuset); pthread_setschedparam(pthread_self(), SCHED_FIFO, &param);
param.sched_priority = RTPRIO; if (pthread_setschedparam(pthread_self(), SCHED_FIFO, &param) != 0)RCLCPP_FATAL(this->get_logger(), "pthread_setschedparam failed"); rclcpp::shutdown();
static void tsNorm(struct timespec ts) while (ts.tv_nsec >= 1'000'000'000L) ts.tv_nsec -= 1'000'000'000L; ++ ts.tv_sec;
float computeTorque() // ここに決定論的制御則を実装 return 0.0f;
void publish(float torque) if (rt_pub->trylock())rt_pub->msg.data = torque;rt_pub->unlockAndPublish();;
int main(int argc, char ** argv) rclcpp::init(argc, argv); auto node = std::make_shared <
RealtimeController > (); node->run(); rclcpp::shutdown(); return 0;

```

テストと検証。エンドツーエンドで遅延とジッタを測定する。ハードウェアインザループを用いて実アクチュエータタイミングを検証する。負荷下での DDS 転送遅延を測定し QoS をチューニングする。非リアルタイムプロセスをアクティブにしたストレステストを実行し、優先度逆転を露呈させる。

エンジニアリングへの影響、トレードオフ、リスク：

- ・決定性はシステムの柔軟性を低下させる。開発者の生産性を保つため、明確な関心の分離を維持する。
- ・SCHED_FIFO および昇格された優先度を用いると、他プロセスの飢餓リスクがある。優先度を慎重に割り当て、優先度継承を実装する。
- ・インナーループをモータコントローラにオフロードすると決定性は向上するが、デバッグが複雑になり観測可能性が低下する。
- ・リアルタイムコードで割り当てやロックを制限できないと、再現が困難な断続的な不安定性が生じる。
- ・ネットワークおよびクロック同期の障害は制御性能を黙って劣化させ、継続的に監視する必要がある。

15.2 センサおよびアクチュエータとの通信

リアルタイム制御における ROS のタイミングと決定性の原則を踏まえ、本小節ではヒューマノイドのセンサとアクチュエータ間の信頼性の高い通信のための実践的なパターンとプロトコルに焦点を当てる。物理層の選択、メッセージ設計、タイミング予算、データ検証、および実際のヒューマノイ

ドプラットフォームで採用可能な具体的な ROS ベースの実装パターンを扱う。

問題設定. ヒューマノイドはバランス維持および操作用にセンサからアクチュエータへ至るタイトなフィードバックループを必要とする. センサには IMU, 力・トルクセンサ, 関節エンコーダ, カメラが含まれる. アクチュエータにはトルク制御モータおよび直列弾性アクチュエータが含まれる. 通信基盤はレイテンシ, ジッタ, スループット, および安全性の要求を満たしながら ROS/ROS2 制御スタックと統合されなければならない.

技術分析.

- ・レイテンシとジッタ：閉ループ安定化はしばしば $f_c = 500 \text{ Hz}$ 以上の制御ループを要求する. 単純な要求として, 片方向通信レイテンシ T_p と制御計算時間 T_c は

$$[H]T_p + T_c < \frac{1}{f_c}. \quad (130)$$

を満たす必要がある. レイテンシジッタ σ_{T_p} は実効ループマージンを低下させる. 最悪ケースの $T_p + \sigma_{T_p}$ を設計に用いる.

- ・メッセージレートと帯域：代表的なセンサレート：

- IMU: 200–2000 Hz, 低ペイロード.
- 関節エンコーダ：500–2000 Hz 集計.
- 力・トルク：500 Hz.

高レートセンサは非決定性 Ethernet よりも決定性バス (EtherCAT, CAN-FD) を好み, TSN が利用可能な場合を除く.

- ・決定性トランスポート：EtherCAT は同期関節制御に適したサブミリ秒サイクルタイムを提供する. CAN および CAN-FD は四肢を制御する分散マイクロコントローラにとってコスト効果が高い. カメラのような集中型高帯域センサには Ethernet+TSN またはリアルタイム Ethernet を用いる.
- ・データ整合性：CRC, シーケンス番号, 単調タイムスタンプを用いる. タイムスタンプは PTP またはハードウェアタイムスタンプを伴う ROS 時間で同期する. 古い計測値は検出され拒否されなければならない.
- ・メッセージセマンティクス：次のための個別トピック/メッセージを用いる：
 1. 高レート制御フィードバック (エンコーダ位置, 速度, IMU).
 2. 中レート状態推定 (融合された姿勢, 速度).
 3. 低レート設定/診断.

高レートチャンネルにはコンパクトなバイナリフォーマットを用いる. ROS2 ではこれらをリアルタイムセーフパブリッシャとして実装するか, ハードウェアインターフェースを伴う `ros2_control` を用いる.

センサ融合と座標整合. センサフレームは明確に定義され較正されていないなければならない. 相補フィルタを用いた IMU ベースピッチ推定では,

$$[H]\theta_k = \alpha(\theta_{k-1} + \omega_k \Delta t) + (1 - \alpha)\theta_{\text{acc},k}, \quad (131)$$

ここで ω_k はジャイロレート, $\theta_{\text{acc},k}$ は加速度計からの傾斜, α はドリフトとノイズを釣り合う. 3D 姿勢融合にはクォータニオン数学を用いる.

実装パターン。次を行うハードウェア抽象化層（HAL）を用いる：

1. 上位層に対して標準的なセンサ/アクチュエータ API を公開する。
2. トランスポート詳細（EtherCAT マスタ，CAN スタック，シリアルパーサ）を処理する。
3. CRC およびタイムスタンプを実行する。
4. *ROS2* トピックにパブリッシュするか，*ros2_control* コントローラに登録する。

次の最小限の *ROS2* 例は IMU データを購読し，単純なフィルタを実行し，関節トルクをパブリッシュする。本コードは例示的であり，本番システムではリアルタイムセーフ構造と *ros2_control* を用いたコマンド仲裁を行わなければならない。

コードサンプル 54 *ROS2* ノード：単純な IMU 駆動トルクコマンド（例示）

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import Imu
from std_msgs.msg import Float64MultiArray
import numpy as np
from typing import List
import threading
from rclpy.executors import MultiThreadedExecutor

class ImuTorqueController(Node):
    def __init__(self) -> None:
        super().__init__('imu_torque_controller')

        # QoS: センサデータ用ベストエフォート
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        self._sub = self.create_subscription(
            Imu, 'imu/data', self.imu_cb, qos)

        self._pub = self.create_publisher(
            Float64MultiArray, 'joint_torques', 10)

        # パラメータ宣言+取得
```

```

self.declare_parameter('dt', 0.002)
self.declare_parameter('alpha', 0.98)
self.declare_parameter('kp', 30.0)
self.declare_parameter('max_torque', 50.0)

self.dt: float = self.get_parameter('dt').value
self.alpha: float = self.get_parameter('alpha').value
self.kp: float = self.get_parameter('kp').value
self.max_torque: float = self.get_parameter('max_torque').value

self.pitch: float = 0.0
self.lock = threading.Lock()

def imu_cb(self, msg: Imu) -> None:
    # 計算は別スレッドでブロックしない
    threading.Thread(target=self._compute_and_publish,
                     args=(msg,), daemon=True).start()

def _compute_and_publish(self, msg: Imu) -> None:
    gz = msg.angular_velocity.y
    acc = msg.linear_acceleration
    pitch_acc = np.arctan2(-acc.x, np.hypot(acc.y, acc.z))

    with self.lock:
        self.pitch = self.alpha * (self.pitch + gz * self.dt) + \
            (1.0 - self.alpha) * pitch_acc
        tau = -self.kp * self.pitch

    # 左右ヒップ対称
    torques: List[float] = [max(min(tau, self.max_torque), -self.max_torque),
                           max(min(tau, self.max_torque), -self.max_torque)]

    cmd = Float64MultiArray()
    cmd.data = torques
    self._pub.publish(cmd)

def main(args=None):
    rclpy.init(args=args)
    node = ImuTorqueController()

```

```

executor = MultiThreadedExecutor()
executor.add_node(node)
try:
    executor.spin()
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

```

エンジニアリングベストプラクティスとチェック.

- すべての計測値にソースでタイムスタンプを付与し、ハードウェア時刻を状態推定に伝播する.
- パケットロス検出のためシーケンス番号を用い; グレースフルデグラデーションを設計する.
- ゲートウェイまたは VLAN を用いて高レートバスを非決定性ネットワークから分離する.
- HAL 内でウォッチドッグおよびアクチュエータトルクリミットを実装し、暴走コマンドを防ぐ.
- マイクロコントローラおよびクロスプラットフォームノード間でエンディアンおよびシリアルライゼーションを検証する.

設計トレードオフと運用上のリスク.

1. EtherCAT を選択するとジッタは減少するが、複雑さおよび開発コストが増加する.
2. 集中型コンピューティングは融合を単純化するが、単一障害点を生む.
3. 低レベル閉ループ制御をマイクロコントローラにオフロードするとレイテンシは減少するが、高レベル協調が複雑になる.
4. 相補フィルタで高い α を用いるとノイズは減少するが、ドリフト感度が増加する.

ヒューマノイドシステムへの影響には、バランス安定性、エネルギー使用、および安全コンプライアンスへの測定可能な影響が含まれる. ヒューマノイドロボットのセンサ・アクチュエータ通信を設計する際には、決定性トランスポートおよびハードウェアレベルの安全インターロックを優先すること.

15.3 同期化の課題

15.4 ヒューマノイドロボットにおける同期とタイミング

前の小節では、センサおよびアクチュエータとの信頼性の高い低遅延通信と、ROS ベース制御によって課せられるリアルタイムスケジューリング制約について議論した。これらの詳細は、ヒューマノイドロボット上で複数の時間クリティカルなサブシステムを協調させる際に生じる同期の問題を動機付ける。

ヒューマノイドロボットにおける同期問題は、不整合なクロック、可変ネットワーク遅延、ソフトウェアスケジューリングジッター、アクチュエータコマンドタイミングから生じる。ヒューマノイドはこれらの問題を増幅させる。なぜなら、多くの関節、力・トルクセンサ、IMU、カメラ、LiDAR センサを融合して安定したバランスと操縦を実現しなければならないからである。エンジニアにとっての

問題定義は単純である。センサ測定値とアクチュエータコマンドが共通の時間基底を参照し、制御デッドラインを満たすことを保証すること。運用上の目標は、要求されるサンプリング周波数で決定論的な制御を実現することであり、通常、関節レベルのループでアクチュエータダイナミクスに応じて 250 Hz から 2 kHz の範囲である。

技術的な分析はクロックと遅延から始まる。センサデータがタイムスタンプ t_s で到着し、コントローラが時刻 t_c にコマンドを発行するとき、実効ループ遅延はその差にソフトウェアおよびネットワーク遅延を加えたものに等しい。連続時間解析では、制御ループ内の純粋な時間遅延 τ はプラント伝達関数に $e^{-s\tau}$ を乗じ、位相余裕を減少させ、潜在的に不安定性を引き起こす。遅延によって導入される周波数領域の位相シフトは

$$[H]\phi_{\text{delay}}(\omega) = -\omega\tau, \quad (132)$$

であるため、クロスオーバー周波数 ω_c では遅延は $\omega_c\tau$ ラジアン位の相を消費する。単純な安定性要件は $\omega_c\tau < \phi_{\text{margin}}$ であり、ここで ϕ_{margin} は利用可能な位相余裕（ラジアン）である。この不等式は、関節および全身コントローラの許容エンドツーエンド遅延予算を導く。

同期誤差の主要な原因とその定量的影響：

1. クロックオフセットとドリフト：補正されないオフセットは体系的な時間的ミスアライメントを生じる。異なる組み込みコントローラ上で数十 ppm のドリフトレートは、数分間でオフセットを増大させる。
2. ネットワークジッター：パケット到着間時間の可変性は見かけのサンプル時間分散を増加させる。コントローラサンプリング周期に匹敵するジッター標準偏差はフィルタ性能を劣化させる。
3. ソフトウェアスケジューリングジッター：非リアルタイムカーネルまたは不適切な優先度は非決定論的なタスク起床時刻を引き起こす。最悪実行時間（WCET）はスケジューラビリティ解析のために有界でなければならない。
4. ハードウェアタイムスタンプミスマッチ：センサまたは NIC がハードウェアタイムスタンプを提供する場合、ソフトウェアはそれらの瞬時を使用しなければならない。ソフトウェア受信時刻を使用すると、可変ネットワーク遅延が測定時刻に加わる。

緩和戦略と実装パターン：

- 単一の分散時刻基底を使用する。Precision Time Protocol (PTP) は EtherCAT または TSN 対応ネットワークなどのイーサネットベースコントローラ間でサブマイクロ秒同期を提供する。サブマイクロ秒アライメントがセンサ融合精度を改善する場合は PTP を使用する。
- ハードウェアタイムスタンプを優先する。センサおよび NIC がハードウェアタイムスタンプを発行するよう設定し、融合および制御タイミングロジックでそれらのタイムスタンプを使用する。
- リアルタイムスケジューリングを強制する。関節レベルコントローラを *PREEMPT_RT* または *SCHED_FIFO* 優先度を持つ専用コプロセッサ上で実行し、遅延とジッターを有界にする。
- 時間遅延推定器でアーキテクトする。避けられない遅延が存在する場合、ゼロ遅延サンプルを仮定するのではなく、遅延を明示的にモデル化するオブザーバおよびフィルタを設計する。
- 監視し適応する。ランタイム遅延モニタと、遅延スパイクが検出されたときに帯域幅を減少させる適応コントローラを実装する。

実用的な実装例として、ROS2（Robot Operating System 2）を使用する。ROS2 はミドルウェアトラ

ンスポートに DDS に依存する。以下のコードは、センサメッセージを読み取り、msg.header.stamp に存在するハードウェアタイムスタンプを優先し、処理遅延を計算する ROS2 Python サブスクリバを示す。ノードはオプションでシステム PTP デーモンに問い合わせてクロックオフセットを検証する。このパターンは、制御パイプラインが正しい時間参照を使用することを保証するのに役立つ。

コードサンプル 55 ROS2 subscriber that computes latency using hardware timestamp

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from builtin_interfaces.msg import Time
from sensor_msgs.msg import Imu
import time
import threading
from typing import Optional

class LatencyMonitor(Node):
    def __init__(self) -> None:
        super().__init__('latency_monitor')

        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self._sub = self.create_subscription(
            Imu, 'imu/data_raw', self._imu_cb, qos)

        self._declare_parameters()
        self._threshold: float = self.get_parameter('latency_threshold_ms').value * 1e-3
        self._window_size: int = self.get_parameter('window_size').value
        self._log_period: float = self.get_parameter('log_period_sec').value

        self._lock = threading.Lock()
        self._latencies: list[float] = []
        self._last_log_time = time.monotonic()

        self._logger.info('LatencyMonitor started.')

    def _declare_parameters(self) -> None:
        self.declare_parameter('latency_threshold_ms', 5.0)
```

```

        self.declare_parameter('window_size', 100)
        self.declare_parameter('log_period_sec', 1.0)

def _imu_cb(self, msg: Imu) -> None:
    now = time.monotonic()
    hw_sec = float(msg.header.sec) + float(msg.header.nanosec) * 1e-9
    latency = now - hw_sec

    with self._lock:
        self._latencies.append(latency)
        if len(self._latencies) > self._window_size:
            self._latencies.pop(0)

    if latency > self._threshold:
        self.get_logger().warn(
            f'High_latency_detected: {latency*1e3:.3f}ms')

    if now - self._last_log_time > self._log_period:
        self._last_log_time = now
        with self._lock:
            if self._latencies:
                avg = sum(self._latencies) / len(self._latencies)
                self.get_logger().info(
                    f'Average_latency(last_{len(self._latencies)}): {avg*1e3:.3f}')

def main(args=None) -> None:
    rclpy.init(args=args)
    node = LatencyMonitor()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

ネットワーク技術を選択する際のトレードオフ：

- EtherCAT および CAN-FD は決定論的サイクリック通信を提供する。ジッターを削減する代わりに、より複雑な配線およびデバイスサポートが必要。
- DDS を使用する標準イーサネットは柔軟であるが、決定論的性能のために TSN または PTP が必要。
- ローカルモータコントローラへの制御オフロードはバストラフィックを削減するが、厳格なファームウェアバージョンおよび機能パリティが要求される。

設計上の結果と運用上のリスク：

- 安全リスク：無界遅延はバランスの喪失またはアクチュエータ飽和を引き起こす可能性がある。保守的な閾値で安全モニタを設計する。
- 複雑性のトレードオフ：PTP およびハードウェアタイムスタンプの追加はシステム複雑性およびデバッグ作業を増加させるが、融合精度を改善する。
- リソース割当：リアルタイムスレッドは CPU パーティショニングを要求する。オーバーコミットメントはデッドライン逸脱のリスクを伴う。
- 保守性：ベンダー間の異種時間源は統合および長期保守を複雑にする。

エンジニアはしたがって遅延を予算し、必要に応じて決定論的バスを選択し、エンドツーエンドでタイミングを計測しなければならない。コントローラ内で遅延を明示的にモデル化し、WCET と優先度を強制し、ハードウェアタイムスタンプを優先して、安定した安全なヒューマノイド動作を保証する。

15.5 システム統合におけるベストプラクティス

これまでのタイミングと決定論的実行に関する議論は、一貫したタイミングを強制し、センサおよびアクチュエータとの堅牢な通信を実現し、管理可能なハードウェア抽象化を行う統合プラクティスへと自然に繋がる。これらのプラクティスは、理論的な制御ループと実装されたヒューマノイドシステムの間のギャップを埋める。

問題定義。ヒューマノイドロボットは高速な制御ループ、高帯域センサ、異種アクチュエータを組み合わせる。統合は遅延を有界に保ち、部分的な故障から回復し、シミュレーションとハードウェア間の動作の等価性を維持しなければならない。主要なエンジニアリング目標は以下である：

- 決定論的な制御ループタイミング；
- 一貫したタイムスタンプ付きセンサ融合；
- 安全なアクチュエータコマンド仲裁；
- 再現可能なデプロイとロールバック。

技術分析。まずタイミング要件を特徴づける。 BW を主要コントローラの閉ループ帯域幅 (Hz) とする。エイリアシングを回避し位相余裕を維持するため、サンプリング周期 T_s を

$$[H]T_s \leq \frac{1}{10 BW}, \quad (133)$$

を満たすように選び、サンプリング周波数 $f_s = 1/T_s \geq 10 BW$ とする。マルチレートサブシステムでは、より遅い推定器を T_s の整数倍に整列させる。

周期的リアルタイムタスクには、固定優先度プリエンプティブスケジューリング下で Liu and

Layland 利用率上限を用いてスケジューラビリティを評価する。計算時間 C_i 、周期 T_i を持つ n 個の独立した周期タスクに対して、

$$[H] \sum_{i=1}^n \frac{C_i}{T_i} \leq n \left(2^{1/n} - 1 \right). \quad (134)$$

n が大きいと右辺は $\ln 2 \approx 0.693$ に収束する。制御、センサ処理、ロギングタスクに優先順位を割り当てる際は保守的にこの上限を用いる。

デバイス間の時刻同期は必須である。サブミリ秒精度のネットワーク時刻プロトコルを用い、オンボードイーサネットにはできれば Precision Time Protocol (PTP) を用いる。センサ測定を同期させる際は、最大センサスキュー Δt が

$$[H] \Delta t \leq \frac{T_s}{2}, \quad (135)$$

を満たし、融合データが同じ制御瞬間を表し時間エイリアシングを回避するようにする。

アーキテクチャパターン。ハードウェアのばらつきを隔離し複雑さを管理するため、階層型アーキテクチャを実装する：

1. Hardware Abstraction Layer (HAL)：アクチュエータとセンサのための決定論的インタフェースを公開する薄くよくテストされたドライバ。
2. Real-Time Control Layer：リアルタイムスケジューリング (SCHED_FIFO) で隔離されたコア上で周期コントローラを実行する。この層は動的メモリ割り当てとブロッキング I/O を排する。
3. Middleware and Orchestration：低遅延にチューニングされた DDS プロファイルを持つ ROS2 を用い、可能ならインプロセス intra-process トランスポートを用いる。
4. Safety and Arbitration Layer：コマンド上限、ウォッチドッグ、モード遷移（例：トルク制御対位置制御）を実施する。
5. Monitoring and Diagnostics：ハートビート、パフォーマンスカウンタ、ヘルスログをスーパーバイザノードヘルレーティングする。

実装上の考慮事項と例。

- ・センサから制御へのデータパスにはロックフリーリングバッファを用い、無界ブロッキングを回避する。
- ・CPU 隔離と cpuset を用いて計算コアと I/O コアを分離しジッタを削減する。
- ・クリティカルパラメータにはコンパイル時設定を用い、ハードリアルタイムタスクではランタイム設定変更を避ける。
- ・正規のクロックソースを一つ維持し、PTP で時刻を伝播し単純なクロスチェックで検証する。

コード例。以下の ROS2 Python ノードは、ヒューマノイドバランスコントローラに供給される状態推定のため、IMU とジョイント状態ストリームを安全に同期する例である。

コードサンプル 56 ROS2 example: synchronizing IMU and joint states for estimator

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
from rclpy.parameter import Parameter
```

```

from rcl_interfaces.msg import SetParametersResult
from sensor_msgs.msg import Imu, JointState
from geometry_msgs.msg import Vector3Stamped
from message_filters import ApproximateTimeSynchronizer, Subscriber
import numpy as np
from typing import Optional, Tuple
import threading
from collections import deque

class StateEstimate:
    """スレッドセーフな状態推定値のコンテナ"""
    __slots__ = ['timestamp', 'q', 'qd', 'qdd', 'rpy', 'omega']
    def __init__(self):
        self.timestamp: rclpy.time.Time = rclpy.time.Time()
        self.q: np.ndarray = np.zeros(12)      # 関節角度
        self.qd: np.ndarray = np.zeros(12)     # 関節速度
        self.qdd: np.ndarray = np.zeros(12)    # 関節加速度（差分近似）
        self.rpy: np.ndarray = np.zeros(3)     # ロール・ピッチ・ヨー
        self.omega: np.ndarray = np.zeros(3)   # 角速度

class EstimatorNode(Node):
    def __init__(self):
        super().__init__('estimator_node')

        # パラメータ宣言
        self.declare_parameter('queue_size', 30)
        self.declare_parameter('slop', 0.005)
        self.declare_parameter('alpha', 0.1)  # 補完係数
        self.add_on_set_parameters_callback(self.param_cb)

        # QoS: リアルタイム制御に適した設定
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=1,
            durability=DurabilityPolicy.VOLATILE
        )

        # Subscriber
        imu_sub = Subscriber(self, Imu, '/imu/data', qos_profile=qos)

```

```

joint_sub = Subscriber(self, JointState, '/joint_states', qos_profile=qos)

# TimeSynchronizer
self.ats = ApproximateTimeSynchronizer(
    [imu_sub, joint_sub],
    queue_size=self.get_parameter('queue_size').value,
    slop=self.get_parameter('slop').value
)
self.ats.registerCallback(self.synced_callback)

# Publisher
self.state_pub = self.create_publisher(Vector3Stamped, '~/state/roll_pitch', 1)

# リングバッファ（最新推定値をロックフリーで共有）
self.state_buffer: deque = deque(maxlen=1)
self.lock = threading.Lock()

# 前回値保持（差分計算用）
self.prev_joint: Optional[Tuple[np.ndarray, rclpy.time.Time]] = None

self.get_logger().info("EstimatorNode initialized")

def param_cb(self, params: list) -> SetParametersResult:
    for p in params:
        if p.name == 'slop':
            self.ats.set_slop(p.value)
    return SetParametersResult(successful=True)

def synced_callback(self, imu_msg: Imu, joint_msg: JointState):
    state = StateEstimate()
    state.timestamp = imu_msg.header.stamp

    # IMUからロール・ピッチ・角速度を抽出
    orient = imu_msg.orientation
    # クォータンション → オイラー角 (ZYX)
    sinr_cosp = 2.0 * (orient.w * orient.x + orient.y * orient.z)
    cosr_cosp = 1.0 - 2.0 * (orient.x * orient.x + orient.y * orient.y)
    state.rpy[0] = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2.0 * (orient.w * orient.y - orient.z * orient.x)

```

```

state.rpy[1] = np.arcsin(np.clip(sinp, -1.0, 1.0))

state.omega = np.array([imu_msg.angular_velocity.x,
                        imu_msg.angular_velocity.y,
                        imu_msg.angular_velocity.z])

# 関節状態をNumPy配列に変換
q = np.array(joint_msg.position, dtype=np.float64)
qd = np.array(joint_msg.velocity, dtype=np.float64)

# 差分から加速度を近似（時刻差を考慮）
if self.prev_joint is not None:
    prev_q, prev_stamp = self.prev_joint
    dt = (state.timestamp - prev_stamp).nanoseconds * 1e-9
    if dt > 1e-4:
        state.qdd = (qd - prev_q) / dt
self.prev_joint = (qd, state.timestamp)

state.q = q
state.qd = qd

# バッファ更新（ロック保護）
with self.lock:
    self.state_buffer.append(state)

# デバッグ出力（軽量化のため最小限）
msg = Vector3Stamped()
msg.header.stamp = state.timestamp
msg.vector.x = state.rpy[0]
msg.vector.y = state.rpy[1]
msg.vector.z = state.omega[2]
self.state_pub.publish(msg)

def get_latest_state(self) -> Optional[StateEstimate]:
    """制御ループから呼ばれる：最新推定値を取得"""
    with self.lock:
        return self.state_buffer[-1] if self.state_buffer else None

def main(args=None):
    rclpy.init(args=args)

```

```

node = EstimatorNode()
try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

```

テストと検証。階層型テストを自動化する：

- ドライバをハードウェアインザループ（HIL）テストハーネスに対してユニットテストする。
- 継続的インテグレーションでは、同じ HAL スタブを用いて Isaac Sim でシミュレーションテストを実行する。
- デプロイ前に、センサドロップアウト、ネットワーク劣化、アクチュエータ故障などのエッジケースを行使する専用統合ラックで回帰スイートを実行する。

運用上の影響、トレードオフ、リスク。

- トレードオフ：コアを隔離し SCHED_FIFO を用いると決定論性が向上するが、同時タスクの柔軟性は低下する。リアルタイムパッチ（PREEMPT_RT）は保守性を複雑にする。
- パフォーマンス対安全性：積極的なコントローラゲインはパフォーマンスを高めるが、遅延とパケットロスに対する許容度は低下する。
- 緩和すべきリスク：
 1. デバイス間の時刻ずれが推定器バイアスを引き起こす—PTP を用いオフセットを監視する。
 2. リアルタイムタスク内の無界ロギングがデッドラインミスを引き起こす—ログを非同期でリダイレクトする。
 3. 検出されない部分的故障—冗長ヘルスチェックとグレースフルデグレードモードを実装する。

設計者は許容遅延を定量化し、利用率上限を用いて周期タスクをスケジュールし、時刻同期を実施しなければならない。これらのプラクティスは統合の予期せぬ事態を削減し、産業・フィールド展開におけるヒューマノイドロボットの安全性と再現性を向上させる。

シミュレーションでのヒューマノイドの訓練

16 Isaac Sim の基礎

16.1 シミュレーション環境の構築

本章で強調したヒューマノイド訓練におけるシミュレーションの価値を踏まえ、本小節では信頼性の高い環境構築のための実践的なエンジニアリング手順を示す。焦点は再現可能な物理、正確なアセット忠実度、および制御・センサタイミングを学習目的に合わせることである。

問題定義。シミュレーション環境は安定した繰り返し可能な動特性と現実的なセンサストリームを

生成しなければならない。不安定性やタイミングの不一致は方策学習を破壊し、実機への移行失敗を引き起こす。主なタスクは：

1. 安定な積分タイムステップを選ぶ,
2. ヒューマノイドスケールの相互作用向けに接触・ソルバパラメータを設定する,
3. 正確な質量と慣性プロパティを持つアセットを準備する,
4. センサおよび制御レートを対象ロボットハードウェアに合わせる.

技術的分析と重要な関係.

- 積分安定性：陽積分分子の場合，単自由度ばね一質量系の臨界タイムステップは

$$[H]dt_{\text{crit}} \approx 2\sqrt{\frac{m}{k}}, \quad (136)$$

ここで m は実効質量， k は実効剛性．ヒューマノイドでは最も高速な動特性はしばしば直列弾性アクチュエータや剛性の高い接触に由来する；最小の m/k 比を用いて dt を制限する．陽積分を用いる場合は物理タイムステップを $dt_{\text{phys}} \ll dt_{\text{crit}}$ に選び，大きなタイムステップが必要な場合は陰積分分子を用いる．

- 制御と物理タイミング： f_{ctrl} を制御周波数， f_{phys} を物理更新レートとする． $f_{\text{phys}} \geq f_{\text{ctrl}}$ を確保し，決定論的積分のため整数サブステップを用いる：

$$[H]N_{\text{sub}} = \left\lceil \frac{f_{\text{phys}}}{f_{\text{ctrl}}} \right\rceil. \quad (137)$$

サブステップングを適用することで制御コマンドと物理更新の間のエイリアシングを防ぐ．

- センサスループット：解像度 $W \times H$ ，チャンネル数 C ，チャンネルあたりバイト数 b ，フレームレート f のカメラに対し，データレートは $R = W \cdot H \cdot C \cdot b \cdot f$ ．これを用いてマルチエージェントまたはマルチセンサ学習向けに GPU およびネットワークリソースを見積もる．

実装チェックリスト（エンジニアリング手順）.

1. 物理コア設定

- 積分分子を選択（剛性接触には陰を推奨；小さな dt なら陽可）.
- 式 (136) を用いて 3 倍のマージンを取り dt_{phys} を設定.
- サブステップングを設定し，制御周波数が物理ステップを正確に割るようにする.
- ヒューマノイド安定性のためソルバ反復回数を増やす（PhysX での位置/速度反復）.

2. 接触と摩擦

- 衝突幾何を単純化：接触用に凸包またはプリミティブコライダ.
- 摩擦係数 μ を表面に合わせ調整；ゴム質ソールなら 0.5–0.8 程度から開始.
- ヒューマノイド足部の反発を 0–0.1 に減らし，バウンスアーティファクトを回避.
- アクチュエータコンプライアンスと整合した接触減衰および剛性値を用いる.

3. アセット忠実度と慣性

- アセットをメートル単位にスケール；ジョイント軸とローカルフレームを検証.
- CAD 質量プロパティからリンク慣性を計算．欠落時は一様箱として慣性を近似：

$$I_{xx} = \frac{1}{12}m(h^2 + d^2)$$

他軸は順次置換.

- 質量中心を正しく配置し, 持続的なドリフトを回避する.

4. センサと遅延

- カメラおよび IMU レートをハードウェア同等値に設定.
- センサノイズモデルを追加 (IMU にはガウスバイアス, カメラには画素ノイズ).
- ネットワーク化されたコントローラを学習する際は通信遅延とパケット損失をシミュレート.

5. 再現性

- 決定論が要求されるドメインランダム化実験ではランダムシードを固定.
- データセットプロベナンスのため環境パラメータをメタデータに記録.

実用的構成パターン. 以下の Python スニペットは物理サブステップを計算し, 安全な物理タイムステップを設定し, これらの値をシミュレータ設定に適用する場所を示す. 関数 `apply_to_simulator` は Isaac Sim API または設定ファイルへの実用的フックであり, その本体を利用可能な Isaac Sim バージョンの具体的 API 呼び出しにマッピングする.

コードサンプル 57 物理と制御のタイミングパラメータを計算し適用.

```
import math
from typing import Dict, Tuple, Final

# デフォルト値を定数化
DEFAULT_SAFETY: Final[float] = 3.0
DEFAULT_SOLVER_ITERS: Final[int] = 10
DEFAULT_FRICTION: Final[float] = 0.6
DEFAULT_RESTITUTION: Final[float] = 0.05

def compute_timing(
    m_eff: float,
    k_eff: float,
    f_ctrl: float,
    safety_factor: float = DEFAULT_SAFETY
) -> Tuple[float, int, float]:
    """
    臨界タイムステップを計算し、制御周波数に応じたサブステップ数を返す。
    戻り値: (物理ステップ, サブステップ数, 物理周波数)
    """
    if m_eff <= 0.0 or k_eff <= 0.0 or f_ctrl <= 0.0 or safety_factor <= 0.0:
        raise ValueError("全ての引数は正の値である必要があります")

    dt_crit = 2.0 * math.sqrt(m_eff / k_eff)          # 臨界ステップ
    dt_phys = dt_crit / safety_factor                 # 安全マージン適用
```

```

f_phys = 1.0 / dt_phys
n_sub = max(1, int(math.ceil(f_phys / f_ctrl))) # 整数サブステップ
return dt_phys, n_sub, f_phys

def apply_to_simulator(
    sim_config: Dict[str, object],
    dt_phys: float,
    n_sub: int,
    *,
    solver_iterations: int = DEFAULT_SOLVER_ITERS,
    friction: float = DEFAULT_FRICTION,
    restitution: float = DEFAULT_RESTITUTION
) -> Dict[str, object]:
    """
    与えられたシミュレータ設定辞書に物理パラメータを上書きして返す。
    元の辞書は変更されない。
    """
    cfg = sim_config.copy()
    cfg.update({
        "physics_step": dt_phys,
        "substeps": n_sub,
        "solver_iterations": solver_iterations,
        "friction_coefficient": friction,
        "restitution": restitution,
    })
    return cfg

# 使用例
if __name__ == "__main__":
    dt_phys, n_sub, f_phys = compute_timing(
        m_eff=2.0,
        k_eff=1e4,
        f_ctrl=200.0
    )
    sim_cfg = apply_to_simulator({}, dt_phys, n_sub)
    # sim_cfgをシミュレータ初期化へ渡す
    エンジニアリングへの影響, トレードオフ, リスク.

```

- `dt_phys` を小さくすると忠実度と安定性が向上するが、CPU/GPU 負荷が増し大規模学習バッチのリアルタイムスループットが低下する。剛性接触には陰ソルバを選び、ステップあたり計算コストと引き換えに大きなタイムステップを許容する。
- 過度に複雑な衝突幾何は接触計算を増やし、非決定的トンネリングを引き起こす。足部や手部には簡略化コライダを用い、視覚メッシュは知覚リアリズムのために保持する。
- 不正確な慣性プロパティは学習された制御にバイアスをかける。浮遊基底ダイナミクスは小型ハードウェア実験またはシステム同定で必ず検証する。
- センサレートとノイズモデルは方策の頑健性に直接影響する。ノイズの過小モデリングは脆い方策を生み、過大モデリングは収束を遅くする可能性がある。

運用上のリスク軽減：決定論シードを用いた小規模シナリオセットを検証し、`dt` およびソルバ反復回数に対する感度スイープを実行し、再現性のため設定バージョンを維持する。

16.2 仮想ツインの構築

仮想ツインの構築は、シーン、センサ、物理カーネルがシミュレーション環境に設定された後に開始される。ツインは環境設定を継承しながら、運動学、動力学、接触、センシングのための高忠実度なロボット固有モデルを追加しなければならない。

エンジニアリングの課題は、代表的な制御ポリシー下で実機の応答と一致するシミュレートされたヒューマノイドを作成することである。有用な運用上の定義は：ツインは同一の指令軌道および入力下で観測可能な差異を最小化する。差異は通常、トルク、関節軌道、圧力中心 (CoP)、センサ出力に現れる。したがって解決策は、幾何学的忠実度、パラメータ同定、接触モデリング、センサエミュレーションを組み合わせる。

技術分析

- 運動学と幾何学：
 - リンク形状と質量分布のための CAD 幾何学を取得する。リアルタイム接触には簡略化された凸型衝突メッシュを使用し、知覚のためには詳細なビジュアルメッシュを保持する。
 - 関節軸とリンクフレームがハードウェア組立図と一致することを確認する。わずかなフレームオフセットが慣性およびヤコビ誤差を引き起こす。
- 動力学とパラメータ同定：
 - 標準形式でロボット動力学を表現する。 p を同定すべき慣性および関節伝達パラメータのベクトルとする。測定関節トルク $\tau_{\text{real}}(t)$ と状態 $q(t)$, $\dot{q}(t)$, $\ddot{q}(t)$ に対して、

$$[H] \min_p \sum_t \left\| \tau_{\text{real}}(t) - (M(q; t; p) \ddot{q}(t) + C(q, \dot{q}; t; p) + g(q; t; p) + \tau_{\text{fric}}(\dot{q}; t; p)) \right\|_2^2. \quad (138)$$

を解く。

- パラメータベクトルに伝達効率、クーロンおよび粘性摩擦、アクチュエータトルクリミットを含める。
- 接触モデリング：
 - 歩行にとって足部接触は重要である。単一接触点ではなく、複数の接触パッチと適合接触スプリングを持つ足底をモデル化する。
 - すべりおよび落下実験のベンチテストを用いて表面摩擦係数と反発係数を校正する。

- センサと遅延：
 - IMU バイアス、スケール、ホワイトノイズのカメラ内部パラメータ、ローリングシャッター効果、およびノイズモデルをエミュレートする。
 - ネットワークおよび制御ループ遅延を含める。遅延は位相をずらし、閉ループ安定性を劣化させる。
- ドメインランダムマイゼーション：
 - 頑健性を高めるため、訓練中に検証済み範囲内でパラメータをランダム化する。ランダムマイゼーションは同定不確実性によって制約されるべきである。

実践的実装ワークフロー

1. 幾何学および関節レベルの忠実度を確立：
 - CAD 幾何学をインポートし、衝突メッシュを簡略化する。
 - モデルとハードウェアからの順運動学を比較することで関節変換を検証する。
2. ペアード実データを収集：
 - すべてのモードを励起する同定軌道をハードウェアで実行する。
 q , dq , ddq , τ , imu , $cameras$ を記録する。
 - 環境条件（温度、バッテリー電圧）を記録する。
3. 動力学パラメータを同定：
 - 式 (138) を用いて最小二乗または制約付き最適化により p をフィットさせる。
4. Isaac Sim へ統合：
 - USD アーティキュレーション慣性、質量、および関節プロパティを更新する。
 - 接触幾何学を追加し、PhysX 接触パラメータを調整する。
5. 閉ループテストで検証：
 - 同一のコントローラをシムとハードウェアで実行する。トルクプロファイル、CoP 位置、ステップリカバリを比較する。
 - 同定および接触チューニングを繰り返す。

実装例：慣性パラメータ同定

コードサンプル 58 最小二乗を用いた慣性パラメータ同定

```
import numpy as np
from scipy.optimize import least_squares
from typing import Tuple, Optional, Callable
import logging
import time
import os
import json

# ロギング設定
logging.basicConfig(level=logging.INFO, format='%(asctime)s-%(levelname)s-%(message)s')
logger = logging.getLogger(__name__)
```

```

class RBDParameterEstimator:
    """剛体ダイナミクスパラメータ推定クラス"""

    def __init__(self,
                  model_func: Callable[[np.ndarray, np.ndarray, np.ndarray, np.ndarray], np.ndarray],
                  param_bounds: Optional[Tuple[np.ndarray, np.ndarray]] = None,
                  cad_param_source: Optional[str] = None):
        """
        Args:
            model_func: RBDモデル関数 (params, q, dq, ddq) -> tau_model
            param_bounds: パラメータの上下限タプル (lower, upper)
            cad_param_source: CADパラメータソースファイルパス
        """
        self.model_func = model_func
        self.param_bounds = param_bounds
        self.cad_param_source = cad_param_source

    def residuals(self, params: np.ndarray, q: np.ndarray, dq: np.ndarray,
                  ddq: np.ndarray, tau_real: np.ndarray) -> np.ndarray:
        """残差計算"""
        try:
            tau_model = self.model_func(params, q, dq, ddq)
            return (tau_real - tau_model).ravel()
        except Exception as e:
            logger.error(f"モデル計算エラー: {e}")
            return np.full(tau_real.size, np.inf)

    def validate_data(self, q: np.ndarray, dq: np.ndarray,
                     ddq: np.ndarray, tau_real: np.ndarray) -> bool:
        """入力データ検証"""
        if not all(isinstance(arr, np.ndarray) for arr in [q, dq, ddq, tau_real]):
            logger.error("入力は全てnumpy.ndarrayである必要があります")
            return False

        if not (q.shape == dq.shape == ddq.shape == tau_real.shape):
            logger.error("全ての配列形状が一致している必要があります")
            return False

        if q.size == 0:

```

```

        logger.error("空の配列が入力されました")
        return False

    return True

def estimate(self, q: np.ndarray, dq: np.ndarray, ddq: np.ndarray,
             tau_real: np.ndarray, p0: Optional[np.ndarray] = None,
             **optimizer_kwargs) -> Tuple[np.ndarray, dict]:
    """
    パラメータ推定実行

    Returns:
    推定パラメータと最適化結果のメタデータ
    """
    if not self.validate_data(q, dq, ddq, tau_real):
        raise ValueError("無効な入力データです")

    # 初期推定値設定
    if p0 is None:
        p0 = self._get_initial_params()

    # デフォルト最適化設定
    default_kwargs = {
        'method': 'trf',
        'xtol': 1e-8,
        'ftol': 1e-8,
        'gtol': 1e-8,
        'max_nfev': 2000,
        'verbose': 2,
        'bounds': self.param_bounds
    }
    default_kwargs.update(optimizer_kwargs)

    logger.info(f"パラメータ推定開始: 初期パラメータ数={len(p0)}")
    start_time = time.time()

    try:
        result = least_squares(
            self.residuals,
            p0,

```

```

        args=(q, dq, ddq, tau_real),
        **default_kwargs
    )

    elapsed_time = time.time() - start_time
    logger.info(f"推定完了: 実行時間={elapsed_time:.2f}s, 反復回数={result.nfe}")

    metadata = {
        'success': result.success,
        'cost': result.cost,
        'optimality': result.optimality,
        'nfev': result.nfev,
        'elapsed_time': elapsed_time,
        'message': result.message
    }

    return result.x, metadata

except Exception as e:
    logger.error(f"最適化エラー: {e}")
    raise

def _get_initial_params(self) -> np.ndarray:
    """CADまたはデフォルトから初期パラメータ取得"""
    if self.cad_param_source and os.path.exists(self.cad_param_source):
        try:
            with open(self.cad_param_source, 'r') as f:
                params = json.load(f)
                return np.array(params['initial_parameters'])
        except Exception as e:
            logger.warning(f"CADパラメータ読み込み失敗: {e}")

    # デフォルト: 全パラメータを1.0で初期化
    logger.warning("デフォルトパラメータを使用します")
    return np.ones(10) # 適切な次元に調整必要

def save_results(self, params: np.ndarray, metadata: dict,
                 output_path: str) -> None:
    """推定結果をJSON形式で保存"""
    results = {

```

```

        'estimated_parameters': params.tolist(),
        'metadata': metadata,
        'timestamp': time.strftime('%Y-%m-%d_%H:%M:%S')
    }

    os.makedirs(os.path.dirname(output_path), exist_ok=True)
    with open(output_path, 'w') as f:
        json.dump(results, f, indent=2)
    logger.info(f"結果を保存:{output_path}")

# 使用例
def model_torque(params: np.ndarray, q: np.ndarray, dq: np.ndarray,
                 ddq: np.ndarray) -> np.ndarray:
    """
    RBDモデル関数 - 実際の実装に置き換えてください
    この例では簡単な線形モデルを使用
    """
    # ダミー実装 - 実際の剛体ダイナミクスモデルに置き換える
    n_samples, n_joints = q.shape
    tau = np.zeros_like(q)

    for i in range(n_joints):
        # 簡略化されたモデル:  $\tau = m \cdot ddq + c \cdot dq + g \cdot q$ 
        m, c, g = params[i*3:(i+1)*3]
        tau[:, i] = m * ddq[:, i] + c * dq[:, i] + g * q[:, i]

    return tau

# メイン実行部分
if __name__ == "__main__":
    # データ読み込み（実際のデータソースに置き換える）
    # q, dq, ddq, tau_real = load_measured_data()

    # ダミーデータ生成（実際の使用時は削除）
    np.random.seed(42)
    T, n = 1000, 3 # サンプル数、関節数
    q = np.random.randn(T, n)
    dq = np.random.randn(T, n) * 0.1
    ddq = np.random.randn(T, n) * 0.01
    tau_real = model_torque(np.array([1.0, 0.1, 9.8] * n), q, dq, ddq) + np.random.randn(T, n)

```

```

# 推定器初期化
estimator = RBDParameterEstimator(
    model_func=model_torque,
    param_bounds=(np.zeros(3*n), np.ones(3*n) * 10)
# パラメータの物理的制約
)

# パラメータ推定実行
p_est, metadata = estimator.estimate(q, dq, ddq, tau_real)

# 結果保存
estimator.save_results(p_est, metadata, "results/estimated_params.json")

# 推定精度評価
tau_est = model_torque(p_est, q, dq, ddq)
rmse = np.sqrt(np.mean((tau_real - tau_est)**2))
logger.info(f"推定RMSE: {rmse:.6f}")

```

検証メトリクスとテスト

- 関節ごとのトルク RMSE および正規化誤差。
- 立脚遷移中の圧力中心平均二乗誤差。
- ランダムプッシュに対するステップリカバリ成功率。
- 較正チェック用のビジュアルセンサピクセル再投影誤差。

設計トレードオフと運用上のリスク

- 忠実度対計算：高忠実度接触メッシュおよび完全慣性テンソルは CPU/GPU 負荷を増加させる。訓練スループット対転移性能をプロファイリングすることで忠実度をバランスさせる。
- 同定データへの過適合：狭い軌道セットに対して密にフィットさせると脆いツインが生成される。多様な励起を使用し、別の挙動で交差検証する。
- モデル化されない現象：ケーブルコンプライアンス、熱ドリフト、バンパー圧縮性はしばしばモデル化されず、残留 sim2real ギャップを引き起こす。残留を定量化してモデリング努力を優先付けする。
- 安全リスク：不完全なツインにコントローラ認証を依存すると不安定モードを隠蔽する可能性がある。常に保守的な安全マージンおよびハードウェアインザループチェックを含める。

エンジニアリングへの影響

- 正確なツイン構築は必要な実世界試行を削減し、ハードウェアの摩耗およびリスクを低減する。
- 同定および接触モデリングへの投資は歩行タスクに対して不釣り合いなほどの利得をもたらす。
- トレーサビリティを維持する：同定データセット、パラメータバージョン、および検証レポート

を保存して訓練結果を再現する。

16.3 物理ベースシミュレーションの機能

シーンジオメトリを設定し仮想ツインを配置した後は、シミュレータの物理忠実度に注目する。以下では Isaac Sim の主要な物理機能と、頑健なヒューマノイド学習のための設定方法を説明する。

正確なヒューマノイドシミュレーションは、ダイナミクスとコンタクトの問題である。 n 自由度の関節ヒューマノイドに対して、接触条件下の多体ダイナミクスは次のように簡潔に記述される。

$$[H]M(q)\ddot{q} + C(q, \dot{q}) + G(q) = \tau + J_c(q)^\top \lambda, \quad (139)$$

ここで M は質量行列、 C はコリオリ・遠心力項、 G は重力、 τ はアクチュエータトルク、 J_c は接触ヤコビアン、 λ は接触インパルスである。制御と接触解決は λ を介して相互作用し、 λ は接触点で一方向相補性制約を課す。

ヒューマノイドの運用要件は次を強調する：

- ・ バランスと歩容のための正確な足-地面接触および滑りモデリング、
- ・ 高周波アクチュエータ指令下での安定積分、
- ・ 再現可能な強化学習学習エピソードのための決定論的挙動、
- ・ 並列データ収集のための妥当なシミュレーションスループット。

主要なシミュレーション機能とエンジニアリングパラメータ

1. 積分器とタイムステッピング

- ・ 制御ループ下でのエネルギー挙動改善のため半陰的積分器を使用。離散更新は

$$[H]\dot{v}_{k+1} = \dot{v}_k + \Delta t M^{-1}(\tau_k + J_c^\top \lambda_k - C_k - G_k), \quad q_{k+1} = q_k + \Delta t \dot{v}_{k+1}. \quad (140)$$

- ・ アクチュエータ帯域が十分に解像されるように Δt を選ぶ。一般的なヒューマノイド制御ループは高精度トルク制御のため $\Delta t \leq 0.002$ s を要求；位置制御ポリシーなら 0.005–0.01 s で許容される。

2. ソルバイテレーションとサブステッピング

- ・ 接触ソルバイテレーションを増やし、相互貫通を減らしスティックスリップ遷移を安定化する。
- ・ サブステッピングはグローバルに Δt を減らさず高速接触ダイナミクスを扱う。二足歩行の足衝突には 1 物理ステップあたり 2–8 サブステップを用いる。

3. 接触モデルパラメータ

- ・ 摩擦係数 μ 、反発係数 e 、接触「オフセット」が立脚信頼性に影響する。靴底や足パッド材質ごとに μ を校正する。
- ・ 高速足振動中のトンネリング回避に連続衝突検出 (CCD) を有効化する。
- ・ 代表的初期値：ゴム質靴底なら $\mu \in [0.6, 1.2]$ 、 $e \approx 0.0$ –0.1 で低バウンス。

4. 関節とアクチュエータモデリング

- ・ 関節ドライブを PD コントローラと速度制限付きトルク源としてモデル化する。トルク飽和とアクチュエータ遅れを実装する。

- PD フィードポリシ：

$$[H]\tau = K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}) + \tau_{ff}. \quad (141)$$

- 粘性減衰とモータ慣性を含め、擾乱下での現実的な創発挙動を再現する。

5. 接触センサ、力-トルク、IMU モデリング

- センサノイズ、バイアス、遅延をシミュレートし、シムツールリアルギャップを埋める。
- 現実的なサンプリングレートを追加：IMU は 200–1000 Hz、力-トルクセンサは 100–500 Hz で制御要件に応じる。

6. 決定性と再現性

- CPU と GPU パスは異なる結果を生じることがある。ランダムシードを固定し、物理エンジンバージョンをロックして再現可能な学習を行う。
- GPU 加速ソルバはスループットを向上させるが、混合精度下で非決定性を引き起こすことがある。

実用的設定例以下のスニペットは Isaac Sim の物理タイムステップ、サブステップ、CCD、ソルバイテレーション、摩擦を Python API で設定する。アクチュエータ帯域と所望スループットに合わせてパラメータを調整する。

コードサンプル 59 Isaac Sim physics configuration for humanoid training.

```
import omni.isaac.core.utils.prims as prim_utils
from pxr import UsdPhysics, PhysxSchema

# シミュレーション取得
stage = omni.usd.get_context().get_stage()
scene = UsdPhysics.Scene.Get(stage, "/physicsScene")
physx_scene_api = PhysxSchema.PhysxSceneAPI.Apply(scene.GetPrim())

# タイムステップ設定 (2 ms, 4 substeps)
physx_scene_api.CreateTimeStepsPerSecondAttr().Set(500.0)
physx_scene_api.CreateMinPositionItersAttr().Set(4)
physx_scene_api.CreateMinVelocityItersAttr().Set(1)

# ソルバー反復回数
physx_scene_api.CreateSolverPositionIterationCountAttr().Set(50)
physx_scene_api.CreateSolverVelocityIterationCountAttr().Set(20)

# 接触オフセットとCCD
physx_scene_api.CreateContactOffsetAttr().Set(0.001)
physx_scene_api.CreateRestOffsetAttr().Set(0.0)
physx_scene_api.CreateEnableCCDAAttr().Set(True)
```

```
# デフォルトマテリアル
material_path = "/World/defaultMaterial"
prim_utils.create_prim(material_path, "PhysicsMaterial")
material_prim = stage.GetPrimAtPath(material_path)
material_api = UsdPhysics.MaterialAPI.Apply(material_prim)
material_api.CreateStaticFrictionAttr().Set(0.9)
material_api.CreateDynamicFrictionAttr().Set(0.9)
material_api.CreateRestitutionAttr().Set(0.02)
```

```
# GPU加速（利用可能な場合）
```

```
physx_scene_api.CreateEnableGPUDynamicsAttr().Set(True)
physx_scene_api.CreateBroadphaseTypeAttr().Set("GPU")
```

実用的校正ワークフロー

- CAD・ハードウェア仕様シートに一致する剛体質量・慣性から開始する。
- ハードウェア歩行試験から記録した力プロファイルを用いて接触摩擦・減衰を調整する。
- アクチュエータ遅延・トルク曲線をモータデータシートと照合して検証する。
- 決定的単体テスト（固定シード）と確率的テスト（ランダム地形）を実行し頑健性を評価する。

トレードオフと運用リスク

- 高忠実度は計算コストを上昇させる。学習フェーズに応じてスループットと物理的リアリズムをトレードする。
- 過剰に剛性の高い接触は微小 Δt を要求し、学習時間と GPU 負荷を増大させる。
- アクチュエータ非線形性を無視すると、非現実的ダイナミクスを悪用するポリシーが生じる。
- GPU 非決定性は学習回帰を不明瞭にすることがある；シードとソフトウェアバージョンを記録する。

エンジニアリングへの影響

- 初期ポリシー学習では、大量データを生成するため高速で若干近似の物理を優先する。最終ファインチューニングでは忠実度を上げアクチュエータダイナミクスを明示的にモデル化する。
- 接触モデルとアクチュエータ簡略化がもたらすシムツールリアルギャップを常に定量化する。
- シミュレーション接触力、関節トルク、慣性応答をハードウェアベースラインと比較するテストスイートを維持し、運用リスクを管理する。

16.4 シミュレーション問題のデバッグ

仮想ツインの妥当性を検証し、物理スタックを動作させた後は、シミュレーション挙動の再現可能なデバッグに焦点を移す必要がある。以下の手法は、ヒューマノイドが不安定化、貫通、予期しないセンサ出力を示す際に Isaac Sim で実行できる具体的な診断手順へと、物理機能とツインの不一致を結びつける。

問題の概要：Isaac Sim 内のヒューマノイドが非物理的な動き、爆発的な関節速度、接触中の持続的

な足の貫通を示す。これらの症状は、数値積分とタイムステップの不一致、接触ソルバの設定とメッシュ/コライダの誤り、仮想ツインと制御コード間のモデル幾何または駆動の不一致という 3 つの広い原因から生じる。目的は原因を隔離し、誤差を定量化し、的を絞った修正を適用することである。

技術的分析と診断

- ・タイムステップと積分器の安定性。陽的積分器はシステムの固有振動数に対して十分小さいシミュレーションタイムステップを必要とする。剛性 k と質量 m を持つばね・ダンパとしてモデル化された接触に対して、臨界ステップは概ね

$$[H]\Delta t < \frac{2}{\omega_n}, \quad \omega_n = \sqrt{\frac{k}{m}}. \quad (142)$$

Δt が (1) を違反するとエネルギーが増大し、ヒューマノイドが振動または爆発する。修正：dt を減らす、または接触剛性を下げてソルバサブステップを増やす。

- ・接触ソルバイテレーションとソルバ許容誤差。不十分なソルバイテレーションは貫通を持続させる。貫通深さ分布とソルバ残差を検査する。高接触数動作（例：両足蹴り出し）が発生する際はソルバイテレーションとサブステップを増やす。
- ・衝突形状と幾何精度。ビジュアルメッシュはしばしば衝突メッシュと異なる。大きな凹メッシュまたは重複コライダは不安定なインパルスを引き起こす。四肢コライダのために凸分解または簡略化カプセルを用いて接触を安定化させる。
- ・アクチュエータとコントローラの不一致。制御トルクがツイン内でアクチュエータ制限を超えると、積分器と関節制限が非現実的なトルクを生成する。指令対適用トルクをログし、ハードウェア同等の制限で指令を飽和させる。
- ・センサタイミングと同期。遅延センサを読み込む制御ループは不安定なフィードバックを生む。制御レートがシミュレーション固定ステップレートと等しいかその整数約数であることを確保し、コントローラで入力遅延を考慮する。

実践的な段階的ワークフロー（再現可能）

1. 最小シーンで再現：
 - ・アセットを単一のヒューマノイドと平面のみに削減する。
 - ・外部スクリプトとランダムイザを除去する。
2. 決定論的タイムステップを固定：
 - ・固定ステップ積分器を使用し、dt を保守的な値に設定する（高剛性接触には 1/240 s 以下）。
3. 物理と接触ビジュアライザを有効化：
 - ・接触法線と貫通表示をオンにし、インパルスが集中する箇所を検査する。
4. 各物理ステップで主要指標をログ：
 - ・最大貫通深さ、アクティブ接触数、関節速度、指令対適用トルク。
5. 1 パラメータずつ変化させる：
 - ・剛性を下げ、ソルバイテレーションを増やし、メッシュ衝突モデルを入れ替える。
6. 安定化後に複雑性を再導入する。

定量的ログと受け入れ基準

- ・閾値を定義：

- 歩行シナリオにおける足-地面接触の最大貫通深さ < 5 mm。
- 静止時の RMS 関節速度 < 0.05 rad/s。
- 指令トルク飽和率 < 制御ステップの 1
- ステップごとに単純指標と窓統計を計算：

$$[H]\max_penetration = \max_{c \in \mathcal{C}} penetration(c), \quad (143)$$

ここで \mathcal{C} はアクティブ接触の集合。

実装スニペット

- 以下の Python スクリプトは、固定物理タイムステップを設定し、ワールドをステップし、関節と接触統計を CSV ファイルにログするコンパクトな Isaac Sim デバッググループを示す。コールバックと属性名をプロジェクト API に合わせて適応させる。

コードサンプル 60 Minimal Isaac Sim debug loop for humanoid contact diagnostics

```
import os
import signal
import threading
import time
from pathlib import Path
from typing import Dict, List, Optional

import rclpy
from omni.isaac.core import World
from omni.isaac.kit import SimulationApp
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy

class ContactMonitor(Node):
    """Isaac Sim シミュレーション中の接触・関節情報を CSV に記録するノード"""

    def __init__(self, world: World, log_path: Path, robot_prim_path: str) -> None:
        super().__init__("contact_monitor")
        self.world = world
        self.robot_prim_path = robot_prim_path
        self.dt = 1.0 / 240.0 # 固定シミュレーション周期

        # CSV 初期化
        self.log_path = log_path
        self.csvfile = open(self.log_path, "w", newline="")
```

```

self.writer = csv.writer(self.csvfile)
self.writer.writerow(
    ["sim_time", "max_penetration_m", "active_contacts",
     "max_joint_vel", "max_command_torque"]
)
self.csv_lock = threading.Lock()

# 定期ログ用タイマー
self.create_timer(1.0, self.timer_callback)

# シャットダウンシグナル登録
signal.signal(signal.SIGINT, self._signal_handler)
signal.signal(signal.SIGTERM, self._signal_handler)

def step(self) -> None:
    """1ステップ分のデータ収集"""
    self.world.step(self.dt)
    self.world.update()
    sim_time = self.world.current_time

    # 接触情報取得
    contacts = self.world.get_active_contacts()
    max_pen = max((c["penetration_depth"] for c in contacts), default=0.0)
    active_contacts = len(contacts)

    # 関節情報取得
    joint_vels = self.world.get_joint_velocities(self.robot_prim_path)
    cmd_torques = self.world.get_commanded_torques(self.robot_prim_path)
    max_joint_vel = max(abs(v) for v in joint_vels) if joint_vels else 0.0
    max_cmd_torque = max(abs(t) for t in cmd_torques) if cmd_torques else 0.0

    # CSV 書き込み（排他制御）
    with self.csv_lock:
        self.writer.writerow(
            [sim_time, max_pen, active_contacts, max_joint_vel, max_cmd_torque]
        )

def timer_callback(self) -> None:
    """1Hzでログを画面出力"""
    contacts = self.world.get_active_contacts()

```

```

        max_pen = max((c["penetration_depth"] for c in contacts), default=0.0)
        self.get_logger().info(
            f"t={self.world.current_time:.2f}s"
            f"contacts={len(contacts)}┐max_pen={max_pen:.4f}m"
        )

def _signal_handler(self, signum, frame) -> None:
    """シグナル受信時にクリーンアップ"""
    self.get_logger().info("Shutdown┐signal┐received.")
    self.shutdown()

def shutdown(self) -> None:
    """リソース解放"""
    with self.csv_lock:
        self.csvfile.flush()
        self.csvfile.close()
    self.destroy_node()

def main(args=None):
    rclpy.init(args=args)

    # Isaac Sim 起動
    sim_app = SimulationApp({"headless": False})
    world = World(stage_units_in_meters=1.0)
    world.reset()

    # ログ保存先
    log_path = Path(os.environ.get("ISAAC_LOG_DIR", ".") / "contact_debug.csv")

    # ノード作成
    monitor = ContactMonitor(world, log_path, robot_prim_path="/World/humanoid")

    try:
        while sim_app.is_running():
            monitor.step()
            time.sleep(0.0) # headless 時は 0
    except KeyboardInterrupt:
        pass
    finally:

```

```

        monitor.shutdown()
        sim_app.close()
        rclpy.shutdown()

if __name__ == "__main__":
    main()

```

一般的な故障モードと的を絞った修正

- ・持続的な足の貫通：接触ソルバイテレーションを増やし、接触剛性を下げる、または複雑なメッシュコライダをプリミティブに置き換える。
- ・振動またはエネルギー増大：dt を減らす、またはサブステップを増やす；ばねベース拘束の剛性を下げる。
- ・コントローラ起因の爆発：指令トルクをクランプし、レートリミットを追加する、または PD ダンピングを用いて高周波励振を除去する。
- ・ずれたセンサ：ステージ単位（メートル）、座標系規則、USD アセット内のオフセット変換を確認する。

技術的影響、トレードオフ、運用上のリスク

- ・dt を下げる、またはソルバイテレーションを増やすと CPU/GPU コストが上昇し実験スループットが低下する。学習速度と物理的リアリズムをバランスさせるシミュレーション精度を選択する。
- ・コライダを単純化（凸カプセル対メッシュ）は安定性を向上させるが、微細な操作や足のコンプライアンスに重要な微妙な接触挙動を失う可能性がある。
- ・シミュレーション不安定性を修正するためにトルクを過度にクランプすると、実機で失敗するコントローラを隠蔽する可能性がある；常に保守的な安全制限で実機を検証する。
- ・学習中のログ不足は不安定エピソードが学習を汚染させる可能性がある。物理的受け入れ基準に違反するエピソードをログしフィルタリングする。

上記の診断ワークフローに従って不安定化の支配的な原因を隔離する。決定論的再現、固定ステップ積分、数値指標を優先してソルバとモデル調整を導く。

17 データ収集と学習

17.1 合成トレーニングデータの生成

前節の仮想ツインの作成と Isaac Sim の物理忠実度の活用に基づき、本項ではヒューマノイドロボットのための合成トレーニングデータの生成方法を詳述する。焦点は実用的であり、知覚・制御モデルの学習を加速し、sim-to-real ギャップのリスクを最小化する多様でラベル付きされたマルチモーダルデータセットを生成することである。

問題定義と工学的的重要性。物理的ヒューマノイドでの大規模かつ適切にラベル付けされたデータ

セットの収集は、コストが高く、時間がかかり、危険を伴う。合成データは、センススイートの高速なイテレーション、故障モードの安全な探索、教師あり学習のための正確な正解データを可能にすることで、これらの制約に対処する。重要な工学要件は、重要な部分でのリアルizm、制御可能な多様性、再現性、スケーラブルなスループットである。

技術分析：モダリティ、変化軸、ラベル付け。

- 合成対象モダリティ：

1. 視覚：RGB、深度、セマンティックセグメンテーション、オプティカルフロー。ビジョン、姿勢推定、物体操作に必須。
2. 本体感覚：関節角度、速度、トルク。モデルベース制御と模倣学習に重要。
3. 慣性：IMU 線形加速度と角速度、バイアスおよびノイズモデル付き。
4. 接触・力：足部接触イベントと地面反力、バランス・歩容学習用。
5. 触覚・触覚フィードバック（シミュレートされる場合）：操作タスクと把持ポリシー用。

- ランダマイズ対象の変化軸：

- 環境ジオメトリ、テクスチャ、照明。
- 物体姿勢と物理特性（質量、摩擦）。
- 製造公差内でのヒューマノイド形態。
- センサ内部・外部パラメータ、キャリブレーション誤差、ノイズ統計。
- シナリオレベル要因：床コンプライアンス、傾斜、障害物、動的アクター。

- シミュレーション独自のラベル付け機能：

- すべてのボディの正確な 6-DoF 姿勢。
- ピクセル単位のセマンティックラベルとインスタンスセグメンテーション。
- 正解接触タイミングと符号付き距離場。
- 制御タイムステップに同期した合成運動学トレース。

サンプリング戦略と重要度重み付け。一様ランダム化は、有益でないサンプルにシミュレーションバジェットを浪費する可能性がある。層化サンプリングと重要度サンプリングを用いて、転倒直前状態や遷移動作などの重要な領域に集中する。

$p_{\text{sim}}(x)$ をシミュレータサンプリング分布、 $p_{\text{target}}(x)$ を実機で観測される分布とする。重要度重みは、学習中の分布ミスマッチを補正する：

$$[H]w(x) = \frac{p_{\text{target}}(x)}{p_{\text{sim}}(x)}. \quad (144)$$

実際には、小規模な実データセットまたはカーネル密度推定を用いて p_{target} を近似する。重み付き学習はバイアスを減らす分散を増加させる；勾配を安定化させるため重みをクリップする。

結合データセット用の損失混合。実データと合成データを混合して学習する際、損失の凸結合を用いる：

$$[H]\mathcal{L} = \alpha \mathbb{E}_{x \sim \mathcal{D}_{\text{real}}} [\ell(\theta; x)] + (1 - \alpha) \mathbb{E}_{x \sim \mathcal{D}_{\text{synth}}} [w(x) \ell(\theta; x)], \quad (145)$$

ここで $\alpha \in [0, 1]$ は実データ忠実度と合成スループットのトレードオフを制御する。

実装パイプラインと実用的コードパターン。パイプラインは以下のステップを含む：

1. シナリオ分布とパラメータ化を定義する。

2. 環境・ヒューマノイドパラメータをシードで決定的にランダム化する。
3. スループットのため並列でバッチシミュレーションを実行する。
4. 同期されたマルチモーダルストリームと正確なラベルをキャプチャする。
5. 後処理：センサノイズモデル追加、データ圧縮、学習シャード作成。
6. 指標と小規模実世界ホールドアウトテストで合成リアリズムを検証する。

Isaac Sim 用の Python スタイルパイプラインスニペット。リストはバッチ生成・ラベル付けの共通構造を示す。コメントは簡潔。

コードサンプル 61 Isaac Sim 擬似コードによる合成データ生成ループ

```
#!/usr/bin/env python3
import os
import json
import logging
from pathlib import Path
from typing import Dict, Any, Tuple

import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.executors import MultiThreadedExecutor
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy

# Isaac Sim
import carb
from omni.isaac.kit import SimulationApp
from omni.isaac.core import World
from omni.isaac.core.utils.nucleus import get_assets_root_path
from omni.isaac.core.utils.stage import open_stage
from omni.isaac.core.utils.prims import define_prim
from omni.isaac.nucleus import get_assets_root_path
from pxr import UsdGeom, Gf

# 自作ユーティリティ（同ディレクトリに配置）
from utils.record_io import RecordWriter
from utils.randomizer import ScenarioRandomizer
from utils.policy import PolicyBase, ScriptedPolicy

# ログ設定
logging.basicConfig(
    level=logging.INFO,
```

```

        format="%(%asctime)s_[(levelname)s]_%(name)s:_%(message)s",
        handlers=[logging.StreamHandler()],
    )
    logger = logging.getLogger("data_gen")

```

```

class DataGenNode(Node):

```

```

    """ROS2ノードとして動作するIsaac_Simデータ収集器"""

```

```

    def __init__(self, config_path: str) -> None:
        super().__init__("isaac_data_gen")

```

```

        # パラメータ読み込み

```

```

        with open(config_path, "r") as f:
            self.cfg: Dict[str, Any] = json.load(f)

```

```

        # Isaac Sim起動

```

```

        self.sim_app = SimulationApp({"headless": self.cfg["headless"]})
        self.world = World(physics_dt=self.cfg["physics_dt"])

```

```

        # シーン読み込み

```

```

        assets_root_path = get_assets_root_path()
        if assets_root_path is None:
            raise RuntimeError("Nucleusサーバに接続できません")
        stage_path = assets_root_path + self.cfg["stage_path"]
        open_stage(stage_path)
        self.world.reset()

```

```

        # Humanoidスポン

```

```

        humanoid_prim = define_prim("/World/Humanoid", "Xform")
        # ここにUSDパスを解決してロードする実装を追加
        # usd_path = ...
        # humanoid_prim.GetReferences().AddReference(usd_path)

```

```

        # ランダマイザ・ポリシー・ライター初期化

```

```

        self.randomizer = ScenarioRandomizer(self.cfg["randomizer"])
        policy_cls = ScriptedPolicy if self.cfg["use_scripted_policy"] else PolicyBase
        self.policy = policy_cls(self.cfg["policy"])
        output_dir = Path(self.cfg["output_dir"]).expanduser()
        output_dir.mkdir(parents=True, exist_ok=True)

```

```

self.writer = RecordWriter(output_dir, self.cfg["record"])

# タイマー登録 (ROS 2スピン内で実行)
self.timer = self.create_timer(
    self.cfg["physics_dt"], self.step_callback
)

# シードループ制御用
self.seed_idx = 0
self.t = 0
self.steps_per_seed = self.cfg["steps_per_seed"]
self.num_seeds = self.cfg["num_seeds"]

def step_callback(self) -> None:
    """ROS 2タイマーコールバック：1ステップ進める"""
    if self.seed_idx >= self.num_seeds:
        self.finalize()
        return

    # シード切り替え
    if self.t == 0:
        seed = self.cfg["base_seed"] + self.seed_idx
        np.random.seed(seed)
        params = self.randomizer.sample(seed)
        self.world.reset()
        logger.info(f"Seed_{seed} 開始")

    # 行動決定・ステップ
    action = self.policy.get_action(self.t, params)
    self.world.step(render=not self.cfg["headless"])

    # 観測取得
    rgb = self.capture_rgb()
    depth = self.capture_depth()
    seg = self.capture_segmentation()
    imu = self.read_imu()
    joints = self.read_joint_states()
    contacts = self.read_contact_forces()

    # 記録

```

```

self.writer.write(
    seed=self.cfg["base_seed"] + self.seed_idx,
    step=self.t,
    rgb=rgb,
    depth=depth,
    seg=seg,
    imu=imu,
    joints=joints,
    contacts=contacts,
    meta=params,
)

self.t += 1
if self.t >= self.steps_per_seed:
    self.seed_idx += 1
    self.t = 0

def capture_rgb(self) -> np.ndarray:
    # ここにViewportからRGB取得
    return np.zeros((480, 640, 3), dtype=np.uint8)

def capture_depth(self) -> np.ndarray:
    return np.zeros((480, 640), dtype=np.float32)

def capture_segmentation(self) -> np.ndarray:
    return np.zeros((480, 640), dtype=np.uint8)

def read_imu(self) -> Dict[str, np.ndarray]:
    return {"accel": np.zeros(3), "gyro": np.zeros(3)}

def read_joint_states(self) -> Dict[str, np.ndarray]:
    return {"pos": np.zeros(23), "vel": np.zeros(23)}

def read_contact_forces(self) -> np.ndarray:
    return np.zeros(4)

def finalize(self) -> None:
    logger.info("収集完了")
    self.writer.close()
    self.sim_app.close()

```

```

rclpy.shutdown()

def main():
    rclpy.init()
    node = DataGenNode(
        config_path=os.path.join(
            os.path.dirname(__file__), "config", "data_gen.json"
        )
    )
    executor = MultiThreadedExecutor()
    executor.add_node(node)
    try:
        executor.spin()
    except KeyboardInterrupt:
        pass
    finally:
        node.finalize()

if __name__ == "__main__":
    main()

```

評価指標と検証。合成リアリズムと有効性を以下で測定する：

- ・タスク固有性能差：合成学習・実機テスト精度低下。
- ・画像リアリズム用 Fréchet Inception Distance (FID)、ヒューリスティックとしてのみ使用。
- ・姿勢推定・IMU 残差のキャリブレーション誤差。
- ・カバレッジ指標：合成・実分布間の経験的サポートオーバーラップ。

プロトコル：小規模実検証セットでイテレート。実世界での安全でない動作を引き起こす故障モードの低減を優先する。

工学トレードオフと運用上のリスク。

- ・忠実度対スループット：高忠実度接触・ソフトボディシミュレーションはリアルなデータを生成するが GPU 時間あたりサンプル数を減らす。高・低忠実度バッチを混合してバランスを取る。
- ・シミュレータアーティファクトへの過学習：ランダム化されたセンサモデルとドメインランダム化を注入して脆いポリシーを回避する。
- ・展開時の安全リスク：偏った合成データで学習したコントローラは実機で危険に動作する可能性がある。制約付き安全コントローラと監督付き実機テストで必ず検証する。
- ・ラベル正確性：シミュレーションラベルは正確だが、不正確な物理パラメータは体系的に誤ったラベルを生む可能性がある。可能な限り測定された実機に仮想ツインパラメータをキャリブレーションする。

ションする。

具体的な運用上の推奨：

- ・層化サンプリングを用いて、重要な動作領域付近により多くサンプルを割り当てる。
- ・再現可能なシードとメタデータを維持して故障解析を可能にする。
- ・小規模実データを大規模合成データセットと式 (eq:mixloss) で組み合わせる。
- ・重要度重み分散を監視し、式 (eq:importance) に従って必要に応じてクリップする。

ヒューマノイドシステムへの設計影響：知覚の短いイテレーションサイクル、故障状態の安全な探索、まれなイベントの制御可能な生成が含まれる。トレードオフは計算コスト、忠実度チューニング、sim-to-real 検証オーバーヘッドである。

17.2 知覚モデルの学習

合成データ生成戦略に基づき、次のステップはこれらのデータセットを用いて人間型ロボットのための堅牢で展開可能な知覚モデルを学習させることである。以下の文章は工学問題を定式化し、解析と実践的な手法を提示し、生産指向の人間型知覚パイプラインに適したコンパクトな実装パターンを示す。

問題定義と運用上の関連性。人間型ロボットは全身姿勢推定、手-物体セグメンテーション、意味的シーン理解などのタスクのために知覚モデルを必要とする。知覚の失敗はバランス、操縦安全性、人間との相互作用に直接影響する。工学上の目的は、シミュレーションから多様な現実環境へ一般化しながら、オンボードの遅延と計算制約を満たすモデルを学習することである。

技術的解析。主要な課題はドメインシフトと学習分布外での過信である。これらに対処するために次を組み合わせる：

- ・ドメインランダムマイゼーションを伴う高忠実度合成データ。
- ・ファインチューニングとキャリブレーションのための小規模で厳選された現実データセット。
- ・特徴分布を再重み付けまたは整列させるドメイン適応学習手法。

ドメイン重み付けによる学習の定式化は実用的な目的関数を与える。 $p_{\text{sim}}(x, y)$ と $p_{\text{real}}(x, y)$ をそれぞれシミュレーションと現実のデータ分布とする。重要度重み付けは重み付き経験損失を生む：

$$[H]L(\theta) \approx \mathbb{E}_{(x,y) \sim p_{\text{sim}}} [w(x) \ell(f_{\theta}(x), y)], \quad (146)$$

ここで $w(x) = \frac{p_{\text{real}}(x)}{p_{\text{sim}}(x)}$ は標本重要度を近似し、 ℓ はタスク損失である。実際には、 $w(x)$ はドメイン分類器によって推定されるか敵対的に学習される。

検出とセグメンテーションタスクでは、安定性のためにタスク損失を組み合わせる：

$$[H]L(\theta) = L_{\text{task}}(\theta) + \lambda_{\text{dom}} L_{\text{domain}}(\theta) + \lambda_{\text{reg}} L_{\text{reg}}(\theta). \quad (147)$$

ここで L_{task} はクロスエントロピーと平滑化 L1 バウンディングボックス回帰を含み、 L_{domain} は特徴差異をペナルティし、 L_{reg} は重み減衰である。

モデル選択とアーキテクチャ。精度、遅延、熱制約をバランスさせるアーキテクチャを選択する：

- ・オンボード推論用：知識蒸留を伴う効率的バックボーン (MobileNetV3, EfficientNet-Lite)。
- ・開発用：オフライン学習と特徴転送実験のための大きなバックボーン (ResNet, Swin)。

- 多モーダルセンシング用：カメラフレーム，深度，IMU を早期融合層で統合し，低照度または遮蔽での堅牢性を向上させる．

学習ワークフローと実践的な手順．

1. データセット構成：
 - 幾何学と多様なセンサノイズのための写真写実的シミュレートシーンを使用する．
 - テクスチャ，照明，センサパラメータに対してドメインランダムマイゼーションを適用する．
 - アーリーストッピングとキャリブレーションのために小規模の現実検証セットを確保する．
2. 事前学習とファインチューニング：
 - 大規模合成コーパスで強力な拡張を施しながら事前学習する．
 - 合成と現実の両標本を含む混合バッチでファインチューニングする．
3. ドメイン適応手法：
 - 学習されたドメイン分類器による重要度重み付け．
 - 特徴周辺を一致させるための敵対的特徴整列（勾配反転）．
 - 表現整列のための現実のラベルなしデータでの自己教師付き前処理タスク．
4. キャリブレーションと不確実性：
 - 確率的出力のための温度スケーリングとモンテカルロドロップアウトを使用する．
 - バランスと操縦にストレスを与える現実世界シナリオでリスク指標を検証する．

実装パターン．リストは合成バッチと現実バッチを混合するコンパクトな PyTorch スタイルループを示す．ドメイン分類器からの標本重み付け，混合バッチスケジューリング，段階的学習率スケジュールを実演する．

コードサンプル 62 Mixed synthetic/real training loop with domain weighting

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from typing import Tuple

class ImportanceWeightedTrainer:
    def __init__(
        self,
        model: nn.Module,
        domain_clf: nn.Module,
        feature_extractor: nn.Module,
        optimizer: torch.optim.Optimizer,
        task_loss: nn.Module,
        domain_loss: nn.Module,
        reg_loss: nn.Module,
        lambda_dom: float = 1.0,
        lambda_reg: float = 1e-4,
```

```

max_weight: float = 10.0,
eps: float = 1e-6,
device: torch.device = torch.device("cuda" if torch.cuda.is_available() else "
"):
    self.model = model.to(device)
    self.domain_clf = domain_clf.to(device)
    self.feature_extractor = feature_extractor.to(device)
    self.optimizer = optimizer
    self.task_loss = task_loss
    self.domain_loss = domain_loss
    self.reg_loss = reg_loss
    self.lambda_dom = lambda_dom
    self.lambda_reg = lambda_reg
    self.max_weight = max_weight
    self.eps = eps
    self.device = device

def run_epoch(
    self,
    synth_loader: DataLoader,
    real_loader: DataLoader,
    steps_per_epoch: int
) -> Tuple[float, float, float]:
    self.model.train()
    self.domain_clf.train()
    self.feature_extractor.train()

    synth_iter = iter(synth_loader)
    real_iter = iter(real_loader)

    total_task = 0.0
    total_domain = 0.0
    total_reg = 0.0

    for _ in range(steps_per_epoch):
        xs, ys = next(synth_iter)
        xr, yr = next(real_iter)
        xs, ys, xr, yr = xs.to(self.device), ys.to(self.device), xr.to(self.device), yr.to(self.device)

        x_all = torch.cat([xs, xr], dim=0)

```

```

with torch.no_grad():
    feats = self.feature_extractor(x_all)
    d_logits = self.domain_clf(feats)
    p_real = torch.sigmoid(d_logits).squeeze()
    p_sim = 1.0 - p_real
    w = (p_real / (p_sim + self.eps)).clamp(max=self.max_weight)

preds = self.model(x_all)
loss_task = self.task_loss(preds, torch.cat([ys, yr], dim=0))
loss_weighted = (w * loss_task).mean()

feats_s = self.feature_extractor(xs)
feats_r = self.feature_extractor(xr)
loss_domain = self.domain_loss(self.domain_clf(feats_s), torch.zeros(xs.size(0)))
               self.domain_loss(self.domain_clf(feats_r), torch.ones(xr.size(0)))

loss_reg = self.reg_loss(self.model)

loss = loss_weighted + self.lambda_dom * loss_domain + self.lambda_reg * loss_reg

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

total_task += loss_weighted.item()
total_domain += loss_domain.item()
total_reg += loss_reg.item()

return total_task / steps_per_epoch, total_domain / steps_per_epoch, total_reg / steps_per_epoch

```

評価と検証. シミュレーションと現実の両ベンチマークを使用する. 次を追跡する:

- 現実ホールドアウトセットでのタスク精度 (mAP, IoU).
- 安全性指標: 障害物に対する誤陽性率, バランス損失を引き起こす故障モード.
- ターゲットハードウェアで測定された遅延とメモリフットプリント.

工学上の意味とトレードオフ.

- 忠実度対計算: 高いシミュレーション忠実度はドメインギャップを減らす但データ生成コストを上昇させる.
- 現実データ投資: エッジケースのための小規模標的現実データセットは大規模汎用コレクションより高いROIを生む.

- モデル複雑さ：重いモデルは精度を向上させるが人間型プロセッサで熱スロットリングと遅延のリスクがある。
- 運用上のリスク：ドメインシフト下での過信予測は安全でない動作を引き起こす；キャリブレーションとフェイルセーフを行う。

設計トレードオフは人間型の安全な劣化モードを優先しなければならない。重み係数 λ_{dom} と標本重要度クリッピングを調整し、ノイジーなドメイン推定への過適合を回避する。繰り返し学習と検証のための現実世界故障ケースの継続的収集を維持する。

17.3 強化学習の基礎

合成センサモデルと知覚ネットワークを基盤に、強化学習（RL）はシミュレートされた状態推定を用いてヒューマノイドの閉ループ制御方策を学習する。次のステップは、制御を逐次の決定問題として定式化し、Isaac Sim で収集された経験が安定した方策更新にどう寄与するかを記述することである。

我々は、立位、歩行、動的操縦などのタスクに対して期待累積報酬を最大化する方策を訓練する。形式的に、マルコフ決定過程（MDP）をタプル $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ として定義し、ここで \mathcal{S} は状態、 \mathcal{A} は行動、 $P(s'|s, a)$ は遷移確率、 $R(s, a)$ はスカラー報酬、 $\gamma \in (0, 1]$ は割引率である。パラメトリック方策 $\pi_\theta(a|s)$ に対する目的関数は

$$[H]J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^T \gamma^t R(s_t, a_t) \right]. \quad (148)$$

実用的な訓練では、サンプル効率、安定性、現実世界の安全性の間にトレードオフが必要である。

主要な技術要素と数式

- 価値関数：状態価値 $V^\pi(s) = \mathbb{E}_\pi[\sum_{t \geq 0} \gamma^t R(s_t, a_t) \mid s_0 = s]$ と行動価値 $Q^\pi(s, a)$ は、時間差分更新とブートストラッピングの基盤となる。
- V^π に対するベルマン方程式は、ブートストラップターゲットの再帰的推定を与える。
- 方策勾配定理は、目的関数の勾配を

$$[H]\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right], \quad (149)$$

として与え、ここで A はアドバンテージ関数である。アクタークリティックアーキテクチャは V を推定し、それを用いて勾配推定の分散を削減する。

アルゴリズムの選択とエンジニアリング上のトレードオフ

- オンポリシー手法（PPO, A2C）は安定性と非定常性に対する頑健性を提供するが、より多くの環境との相互作用を必要とする。
- オフポリシー手法（SAC, DDPG）は経験を再利用することでサンプル効率を改善するが、注意深い探索とリプレイバッファ管理が必要である。
- エントロピー正則化とクリッピング目的は更新を安定化させるが、追加のハイパーパラメータを導入する。
- 実機ヒューマノイドにおける部分的観測性は、再発方策やフィルタリングされた IMU とビジョン出力による状態拡張を必要とすることが多い。

シミュレーションでのデータ収集並列化シミュレーションは、高次元ヒューマノイド制御のための十分な経験を収集する上で不可欠である。多くの Isaac Sim レプリカをランダム化された物理パラメータで使用し、頑健性を向上させる。核心的な実践的ステップ：

1. 観測ベクトルを定義：関節角度、速度、IMU、接触センサ、知覚出力（例：足跡ターゲット）。
2. 行動パラメータ化を定義：トルク指令、所望関節位置、または低レベル PD セットポイント。
3. ランダマイゼーションを実装：質量、摩擦、遅延、センサノイズ、および sim-to-real ブリッジングのための視覚テクスチャ。
4. バッチロールアウトを収集し、オンポリシーリターンを計算するか、オフポリシー手法のための遷移をリプレイバッファに格納する。

効率的なアドバンテージ推定には、一般化アドバンテージ推定（GAE）が一般的に用いられる。GAE 数式は、バイアスと分散をバランスするパラメータ λ を用いて、時間差分残差でアドバンテージを計算する。

最小実装スケッチ

コードサンプル 63 Isaac Sim 内のヒューマノイドに対するロールアウト収集と単一方策勾配更新

```
import torch
import torch.nn as nn
from torch import Tensor
from typing import Tuple, List
import numpy as np

class RolloutBuffer:
    """ ロールアウトデータをGPU/CPUに連続で溜めておくバッファ """
    def __init__(self, device: torch.device):
        self.device = device
        self.reset()

    def reset(self):
        self.obs: List[Tensor] = []
        self.act: List[Tensor] = []
        self.rew: List[Tensor] = []
        self.done: List[Tensor] = []
        self.logp: List[Tensor] = []
        self.val: List[Tensor] = []

    def push(self, obs, act, rew, done, logp, val):
        self.obs.append(obs.detach().to(self.device))
        self.act.append(act.detach().to(self.device))
        self.rew.append(torch.as_tensor(rew, device=self.device))
```

```

        self.done.append(torch.as_tensor(done, device=self.device))
        self.logp.append(logp.detach().to(self.device))
        self.val.append(val.detach().to(self.device))

def tensorize(self) -> Tuple[Tensor, ...]:
    return (torch.cat(self.obs),
            torch.cat(self.act),
            torch.cat(self.rew),
            torch.cat(self.done),
            torch.cat(self.logp),
            torch.cat(self.val))

@torch.no_grad()
def compute_gae(rewards: Tensor,
                values: Tensor,
                dones: Tensor,
                next_value: Tensor,
                gamma: float = 0.99,
                lam: float = 0.95) -> Tuple[Tensor, Tensor]:
    """GAE- $\lambda$  で advantage/return を計算 (vectorized) """
    T = rewards.size(0)
    advantages = torch.zeros_like(rewards)
    gae = 0
    next_values = torch.cat([values[1:], next_value.unsqueeze(0)])
    deltas = rewards + gamma * next_values * (1 - dones) - values
    for t in reversed(range(T)):
        gae = deltas[t] + gamma * lam * (1 - dones[t]) * gae
        advantages[t] = gae
    returns = advantages + values
    return returns, advantages

def ppo_update(policy: nn.Module,
               optimizer: torch.optim.Optimizer,
               buffer: RolloutBuffer,
               T: int,
               batch_size: int = 512,
               clip_eps: float = 0.2,
               epochs: int = 10,

```

```

        entropy_coef: float = 0.01,
        value_coef: float = 0.5,
        max_grad_norm: float = 1.0):
    """PPO-Clip で複数epoch更新"""
    obs, act, rew, done, logp_old, val_old = buffer.tensorize()
    with torch.no_grad():
        next_value = policy.critic(obs[-1]).squeeze(-1)
    returns, adv = compute_gae(rew, val_old, done, next_value)
    adv = (adv - adv.mean()) / (adv.std() + 1e-8)

    dataset = torch.utils.data.TensorDataset(obs, act, logp_old, returns, adv)
    loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)

    for _ in range(epochs):
        for o, a, l_old, ret, ad in loader:
            dist, val = policy(o)
            logp = dist.log_prob(a).sum(-1)
            ratio = torch.exp(logp - l_old)
            surr1 = ratio * ad
            surr2 = torch.clamp(ratio, 1 - clip_eps, 1 + clip_eps) * ad
            policy_loss = -torch.min(surr1, surr2).mean()
            value_loss = 0.5 * (ret - val.squeeze(-1)).pow(2).mean()
            entropy = dist.entropy().sum(-1).mean()
            loss = policy_loss + value_coef * value_loss - entropy_coef * entropy

            optimizer.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(policy.parameters(), max_grad_norm)
            optimizer.step()

```

ヒューマノイドへの RL 適用時のベストプラクティス

- より単純なタスクから始め、カリキュラム学習を用いて複雑さをスケールする。
- ドメインランダムマイゼーションを用いて、実機移行前に方策をモデルミスマッチに晒す。
- 物理的リアリズムメトリクスを監視：関節限界、接触インパルス、およびエネルギー消費。
- 予期せぬ挙動に対して、実機で安全フィルタまたはフォールバックコントローラを含める。

パフォーマンスメトリクスと評価以下で方策を評価する：

- ランダムイズされた条件下での成功率。
- サンプル効率：報酬対環境ステップ。
- 頑健性：攪乱後の回復。

- アクチュエータ寿命を確保するためのエネルギーおよびトルク限界。

エンジニアリングへの影響、トレードオフ、および運用上のリスク設計者は、サンプル効率と安定性および頑健性をトレードしなければならない。重度のランダム化は転送を助けるが収束を遅くする。報酬 shaping は学習を加速するが、望ましくない近道のリスクを伴う。実機では、未テストの方策は高トルクと不安定性を引き起こす可能性がある；常に低剛性モードと外部安全モニタで検証すること。最後に、運用上のリスクを管理するため、シミュレーション改良、標的化されたランダム化、保守的な実機テストを交互に繰り返すイテレーションサイクルを計画すること。

17.4 モデル性能の評価

強化学習の基礎からの報酬設計の考慮事項と知覚モデル訓練における精度目標を踏まえ、評価はシミュレートされたヒューマノイドに対して制御能力と知覚信頼性の両方を定量化しなければならない。正確な評価は報酬の再設計、ドメインランダム化、ハードウェアへの展開の判断を促す。

問題定義. メトリクスの前に評価目標を定義せよ。シミュレーションで訓練されたヒューマノイドに対して、通常以下に答える必要がある：

1. ポリシーは環境のバリエーションにわたって確実にタスクを達成するか？
2. ポリシーはセンサノイズと駆動遅延に対してどれだけ頑健か？
3. ポリシーの実行による予想消費電力と摩耗はどれか？
4. 知覚モジュールは未経験の照明とジオメトリにどれだけ一般化するか？

主要なメトリクスのクラスとその算出方法.

- タスク成功と効率:
 - 成功率：タスク基準を満たしたエピソードの割合。
 - 平均エピソードリターン：試行全体での累積報酬 R の平均。
 - エピソード長：成功または終了までの平均タイムステップ数。
- 動的安定性と安全性:
 - ZMP（ゼロモーメントポイント）逸脱：足裏支持ポリゴンからの最大偏差。
 - ピーク接触力スパイク：突発的衝撃の回数と大きさ。
 - 最大関節トルクおよび速度制限違反回数。
- エネルギーと摩耗プロキシ:
 - エネルギー毎メートルまたはエネルギー毎タスク：アクチュエータ電力を時間で積分。
 - 累積ヒートインデックス：部品ストレスのプロキシ。
- 知覚メトリクス（ビジョン／姿勢モジュール用）:
 - オブジェクト検出の適合率、再現率、F1 スコア。
 - セグメンテーションおよび把持領域オーバーラップの Intersection-over-Union (IoU)。
 - 姿勢誤差：並進 RMSE と回転誤差（例：アングルアキシス）。
- 転移メトリクス:
 - シムツーリアル転移ギャップ：シミュレーションとハードウェア試験間のメトリクス値の差。

– ドメイン一般化スコア：ランダム化されたパラメータセット全体での成功率平均。

代表的な数式. モデル比較には標準的で解釈可能な指標を用いる。

$$[H]\text{SuccessRate} = \frac{1}{N} \sum_{i=1}^N \mathbf{1}\{\text{episode}_i \text{ succeeds}\}, \quad (150)$$

ここで $\mathbf{1}\{\cdot\}$ はインジケータ関数である。知覚性能について、F1 スコアは

$$[H]F1 = 2 \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \quad (151)$$

2 つのポリシー間の統計的比較には、Cohen の d が効果量を定量化する：

$$[H]d = \frac{\bar{x}_1 - \bar{x}_2}{s_p}, \quad s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}}. \quad (152)$$

評価プロトコルの推奨. 反復可能で仮説主導のパイプラインを用いる。

1. 成功基準を正確に定義する：姿勢許容差、必要な接触、タイムアウト。
2. 以下を変化させる評価スイートを作成する：
 - 環境ジオメトリ（傾斜、段差高さ）、
 - 物理パラメータ（質量、摩擦）、
 - センサ破損（ガウシアンノイズ、遅延）。
3. 大標本モンテカルロバッチを実行する。可能なら条件あたり数百エピソード以上を用いる。
4. 非ガウスメトリクスにはブートストラップを用いて信頼区間を算出する。

ブートストラップ信頼区間の例（技術注記）。 $\{\theta_i\}$ を M 個のメトリクス標本とする。復元抽出してブートストラップ平均 $\{\hat{\theta}_b^*\}_{b=1}^B$ を生成し、経験的分位点から CI を導出する。

実装：軽量評価ロガー. 以下のスニペットは成功率、平均リターン、タスク毎エネルギー、成功率のブートストラップ CI を算出する実用的な評価器を示す。Isaac Sim または RL ランナーからエクスポートされた CSV ロールアウトを読み込み、結果を集計する。

コードサンプル 64 Evaluation aggregator for humanoid rollouts.

```
import argparse
import json
import logging
import sys
from pathlib import Path
from typing import Dict, Tuple

import numpy as np
import pandas as pd
from sklearn.metrics import precision_recall_fscore_support

# ログ設定
```

```

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s|%(levelname)s|%(message)s",
    datefmt="%Y-%m-%d_%H:%M:%S",
)
logger = logging.getLogger(__name__)

```

```

def load_rollouts(csv_path: Path) -> pd.DataFrame:
    """CSVを読み込み、必要な列が揃っているか検証"""
    try:
        df = pd.read_csv(csv_path)
    except FileNotFoundError:
        logger.error(f"{csv_path}が見つかりません")
        sys.exit(1)

```

```

    required = {"return", "success", "energy", "precision", "recall"}
    missing = required - set(df.columns)
    if missing:
        logger.error(f"CSVに必要な列が不足しています:{missing}")
        sys.exit(1)
    return df

```

```

def bootstrap_ci(x: np.ndarray, B: int = 2000, alpha: float = 0.05) -> Tuple[float, float]:
    """Bootstrapで信頼区間を推定"""
    n = len(x)
    rng = np.random.default_rng(42)
    means = np.array([np.mean(rng.choice(x, n, replace=True)) for _ in range(B)])
    return tuple(np.percentile(means, [100 * alpha / 2, 100 * (1 - alpha / 2)]))

```

```

def compute_metrics(df: pd.DataFrame) -> Dict[str, float]:
    """主要指標を算出"""
    metrics = {
        "success_rate": df["success"].mean(),
        "mean_return": df["return"].mean(),
        "energy_per_task": df["energy"].mean(),
        "precision": df["precision"].mean(),
        "recall": df["recall"].mean(),
    }

```

```

    }
    # F1 計算（ゼロ除算防止）
    p, r = metrics["precision"], metrics["recall"]
    metrics["f1"] = 2 * p * r / (p + r + 1e-12)
    return metrics

def main() -> None:
    parser = argparse.ArgumentParser(description="Rollout_評価スクリプト")
    parser.add_argument("csv", type=Path, help="rollouts.csvへのパス")
    parser.add_argument("--output", type=Path, help="JSON_結果保存先（省略時はstdout）")
    args = parser.parse_args()

    df = load_rollouts(args.csv)
    metrics = compute_metrics(df)
    ci_low, ci_high = bootstrap_ci(df["success"].values)

    results = {
        "success_rate": metrics["success_rate"],
        "success_ci": [ci_low, ci_high],
        "mean_return": metrics["mean_return"],
        "energy_per_task": metrics["energy_per_task"],
        "precision": metrics["precision"],
        "recall": metrics["recall"],
        "f1": metrics["f1"],
    }

    if args.output:
        args.output.write_text(json.dumps(results, indent=2))
        logger.info(f"結果を_{args.output}_に保存しました")
    else:
        print(json.dumps(results, indent=2))

if __name__ == "__main__":
    main()

```

結果の解釈と設計選択.

- 成功率が控えめなパラメータシフトで低下する場合、訓練中のドメインランダムマイゼーションを増やす。

- ・タスク毎エネルギーが高い場合、トルクをペナルティするかエネルギー項を追加する報酬 shaping を検討。
- ・シムツールリアル転移ギャップが大きい場合、ミスマッチ変数への標的ランダムマイゼーションまたは実データファインチューニングが必要。
- ・特定照明条件に知覚失敗が集中する場合、標的合成拡張を実施。

運用上のリスクとトレードオフ。

- ・極めて大規模な評価スイートを実行すると計算コストとイテレーション時間が増大する。
- ・狭い成功基準は脆いポリシーを生み、広い基準は安全でない動作を隠すリスクがある。
- ・平均リターンの過最適化は最悪ケースの失敗を隠すことがある。分布の両端を常に検査せよ。
- ・信頼区間は試行数が少ないと誤解を招くことがある。仮定が崩れる場合は非パラメトリック検定を用いる。

具体的なエンジニアリングへの影響。評価を運用条件に合わせて設計し、展開リスクに応じてテストに予算を配分し、ハードウェア転移判断には制御+知覚統合メトリクスを用いる。徹底性、計算コスト、安全制約のバランスが、成功で信頼できるヒューマノイド展開を決定する。

18 実ロボットへの移行

18.1 学習済みモデルの転送

前述のシミュレーション設定とデータ収集戦略は、モデル転送の基盤を提供する。合成データセットと強化学習によるコントローラを基に、本小節ではこれらのモデルを実機ヒューマノイドに展開する実践的な手法に焦点を当てる。

問題定義と運用上の意義。学習済みポリシーまたは知覚モデルは、安全性、遅延、ハードウェア制約の下で実機ヒューマノイドで確実に動作しなければならない。核心的な課題は、シミュレーションと実機の動特性、センシング、タイミングの差を埋めることである。効果的な転送は立ち上げ時間を短縮し、ハードウェアの摩耗を最小化し、初期テストエピソードでの失敗リスクを低減する。

sim-to-real ミスマッチの原因：

- ・モデリング誤差：質量、慣性、アクチュエータ動特性、接触摩擦の不正確さ。
- ・センシング誤差：カメラキャリブレーション、レンズ歪、IMU バイアス、量子化。
- ・タイミングと遅延：ネットワーク遅延、制御ループジッタ、センササンプリングミスマッチ。
- ・モデル化されていないコンプライアンス：ソフトコンタクト、ケーブル伸び、アクチュエータギアバックラッシュ。
- ・環境ばらつき：照明、表面不規則性、予期しない障害物。

頑健な最適化目的。ポリシー π を、考えられる実世界モデルの分布にわたる期待報酬を最大化するよう設計する。形式的には、

$$[H]\pi^* = \arg \max_{\pi} \mathbb{E}_{\phi \sim p(\Phi)} [R(\pi; \mathcal{M}(\phi))], \quad (153)$$

ここで ϕ は動特性およびセンサばらつきをパラメータ化し、 $p(\Phi)$ は事前不確実性を符号化し、 $\mathcal{M}(\phi)$

はシミュレータインスタンスを表す。

システム同定としての最適化。少数の実機軌跡が利用可能な場合、軌跡誤差を最小化することでシミュレータパラメータ ϕ を同定する：

$$[H]\hat{\phi} = \arg \min_{\phi} \sum_{t=0}^T \|x_t^{\text{real}} - x_t^{\text{sim}}(\phi)\|^2 + \lambda \|\phi - \phi_0\|^2. \quad (154)$$

ここで x_t は状態ベクトル、 ϕ_0 は事前知識、 λ は推定値を正則化する。

実践的な転送手法と実装手順：

1. 高忠実度デジタルツイン：Isaac Sim において運動学、質量特性、センサ配置を複製する。静的キャリブレーションテストで検証する。
2. オフラインシステム同定：実機に対して小規模かつ安全な励振信号（低振幅 PRBS またはスインスイープ）を実行する。(154) を用いてアクチュエータ帯域、減衰、遅延をフィッティングする。
3. 動特性ランダムマイゼーション：学習中に質量、重心オフセット、摩擦、アクチュエータ利得に対して現実的な範囲で ϕ をサンプリングする。 $p(\Phi)$ が同定されたパラメータを中心とするよう確保する。
4. 知覚適応：
 - ・光度ランダムマイゼーション：シミュレーション内で照明、テクスチャ、ノイズをランダム化する。
 - ・ドメイン適応ファインチューニング：実機で少数のラベル付きデータセットを収集し、バックボーンを低学習率でファインチューニングする。
 - ・特徴レベル適応：ラベルが乏しい場合は敵対的アライメントを用いる。
5. 残差学習と適応制御：
 - ・安定性のための保守的ベースラインコントローラを展開する。
 - ・ベースラインの上に学習済み残差ポリシーを適用し、小さな誤差を補正する。
 - ・推定ペイロードまたは剛性のためのオンライン適応モジュールを実装する。
6. 安全エンベロープと段階的テスト：
 - ・腕サポートを伴うつながれたバランステストから開始する。
 - ・支持付き歩行、低速自由運動、最終的には公称速度へと進める。
 - ・異常検出時に即座に安全コントローラに切り替えるハードウェアインザループフォールバックを用いる。

具体的なアルゴリズムパターン：シミュレーションで頑健ポリシーを学習し、ベースラインコントローラが安全性を確保しながら実機で小規模な残差ポリシーをファインチューニングする。これにより必要な実機サンプルを削減する。

評価指標とプロトコル：

- ・追従誤差：関節角度およびベースポーズの二乗平均平方根誤差。
- ・安定性余裕：ゼロモーメントポイントの逸脱および転倒までの時間。
- ・エネルギーおよびトルクピーク：RMS トルクおよび最大ピーク値。
- ・ N シードにわたるタスク試行の成功率。
- ・摩耗指標：測定された温度上昇およびモータ電流トレンド。

コード例：安全チェックを実装しながら学習済みポリシーを実機で実行するための最小限の ROS2 ベースポリシーラッパー。ラッパーは policy.pt をロードし、/joint_states とトルク指令を監視する。

コードサンプル 65 安全モニタ付きリアルタイムポリシーラッパー

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
import torch
import numpy as np
from typing import Optional, List

class PolicyNode(Node):
    def __init__(self) -> None:
        super().__init__('policy_node')

        # QoS設定：リアルタイム性と信頼性を両立
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        self.policy = torch.jit.load('policy.pt', map_location='cpu')
        self.policy.eval() # 推論モード固定

        self.joint_names: List[str] = [] # 初期化後に設定
        self.create_subscription(
            JointState, '/joint_states', self.js_cb, qos)

        self.cmd_pub = self.create_publisher(
            JointTrajectory, '/joint_commands', qos)

        # 安全制限 (Nm)
        self.torque_max = 50.0
        self.torque_limit = np.array([self.torque_max], dtype=np.float32)

        self.last_state: Optional[JointState] = None
```

```

def js_cb(self, msg: JointState) -> None:
    self.last_state = msg
    if not self.joint_names:
        self.joint_names = msg.name # 初回のみ記憶

    action = self.infer_action(msg)
    if self.check_safety(action):
        traj = self.mk_traj(action)
        self.cmd_pub.publish(traj)
    else:
        self.get_logger().warn('Safety limit exceeded, fallback to safe.')
        safe_traj = self.safe_controller(msg)
        self.cmd_pub.publish(safe_traj)

def infer_action(self, js: JointState) -> np.ndarray:
    obs = torch.tensor(js.position + js.velocity, dtype=torch.float32).unsqueeze(0)
    with torch.no_grad():
        a = self.policy(obs)
    return a.squeeze(0).cpu().numpy()

def check_safety(self, action: np.ndarray) -> bool:
    return np.all(np.abs(action) <= self.torque_limit)

def mk_traj(self, action: np.ndarray) -> JointTrajectory:
    traj = JointTrajectory()
    traj.joint_names = self.joint_names
    point = JointTrajectoryPoint()
    point.effort = action.tolist()
    point.time_from_start.sec = 0
    point.time_from_start.nanosec = int(20e6) # 20 ms
    traj.points.append(point)
    return traj

def safe_controller(self, js: JointState) -> JointTrajectory:
    # 零トルク指令で安全停止
    traj = JointTrajectory()
    traj.joint_names = js.name
    point = JointTrajectoryPoint()
    point.effort = [0.0] * len(js.name)
    point.time_from_start.sec = 0

```

```

        point.time_from_start.nanosec = int(20e6)
        traj.points.append(point)
    return traj

def main(args=None):
    rclpy.init(args=args)
    node = PolicyNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

設計上のトレードオフと運用上のリスク：

- ランダマイゼーション範囲を広げると頑健性は向上するがピーク性能は低下する。範囲は保守的に選ぶ。
- 強力な光度ランダマイゼーションは知覚転送を容易にするが、タスクに関連する特徴を隠す可能性がある。
- オンラインファインチューニングは適応を加速するが、頑健な安全モニタがなければハードウェア損傷のリスクがある。
- 残差ポリシーはベースラインコントローラを必要とする；フォールバック制御則の設計が安全運用エンベロープを決定する。

エンジニアリング実践への示唆：早期にキャリブレーションと安全励振テストに投資する。各転送イテレーションのログを保持し、持続的なミスマッチを診断する。安全上重要なヒューマノイドを運用する場合、システム同定や残差制御のような明確な解釈可能性を与える手法を優先する。

18.2 シミュレーションと現実のギャップを橋渡しする

ポリシーと知覚モデルのロボットへの転送が成功した後、次の実用的な課題はシミュレータとハードウェアの不整合による性能低下を軽減することである。本小節では、ヒューマノイドロボットに対するシミュレーションと現実のギャップを測定・軽減・監視するためのエンジニアリング手法を概説し、Isaac Sim ベースのワークフローに適した実装パターンを示す。

正確な問題定義：パラメータ θ でパラメータ化されたポリシーがシミュレーションで訓練された場合、物理ハードウェアにデプロイしながら許容できるタスク性能と安全性を確保することが目標である。2つの核心的技術的タスクが生じる：

1. sim-to-real への移行時に予想される性能劣化を定量化し、
2. ポリシーまたはモデル（または両方）を更新して、デプロイされたコントローラが性能および安全性指標を満たすようにする。

不整合の原因

- ・ダイナミクスモデル誤差：関節摩擦、伝達コンプライアンス、モデル化されていない慣性が系統的な相違をもたらす。
- ・接触・地形不確実性：足-地面間摩擦と微細な幾何形状がバランスと歩容に影響する。
- ・センサおよび駆動アーティファクト：遅延、サンプリングジッタ、バイアス、量子化、センサ軸の不整合が観測忠実度を変化させる。
- ・環境差異：照明、反射、物体テクスチャがビジョンシステムに影響する。
- ・ハードウェアばらつき：アクチュエータの経年変化と部品公差がユニット間で異なる。

エンジニアリングアプローチと解析

1. ランダム化パラメータによる頑健な最適化訓練目的をシミュレータパラメータの分布にわたる期待損失を最小化するように設計する。 p を分布 P からサンプルされたシミュレータ物理パラメータとする。頑健な訓練目的は

$$[H] \min_{\theta} \mathbb{E}_{p \sim P} [L(\pi_{\theta}; p)], \quad (155)$$

であり、ここで L はタスク損失、 π_{θ} はポリシーである。実際には摩擦、質量、遅延、センサノイズの範囲で p をサンプルする。ドメインランダムマイゼーションは1つの公称モデルへの脆い過学習を減らし、最悪場合の現実世界性能を改善する。

2. システム同定とパラメータ推定ハードウェアで短いキャリブレーション試行を収集してシミュレータパラメータ ϕ をフィットさせる。システム同定を非線形最小二乗問題として定式化する：

$$[H] \min_{\phi} \sum_{t=1}^T \|x_{\text{sim}}(t; \phi) - x_{\text{real}}(t)\|^2, \quad (156)$$

ここで x_{sim} は既知の入力列の下でのシミュレートされた状態である。勾配ベースの最適化手法または ϕ についての情報を与えた事前分布として定式化されたベイズ最適化を用いる。正確な ϕ は公称モデルバイアスを減らす。

3. 残差ダイナミクス学習現実とシミュレートされたダイナミクスの相違を残差関数 $r(x, u)$ としてモデル化する。軽量の回帰器 r_{ψ} をペアになった sim/real 軌道差分で訓練する。デプロイ時には、予測可能な誤差を補償するためにコントローラープ内のシミュレータモデルに $r_{\psi}(x, u)$ を加える。
4. 特権教師-教え子と観測シェイピングシミュレータが特権状態を持つ場合、特権入力を用いて教師ポリシーを訓練し、オンボードセンサのみを受け取る教え子を蒸留する。教え子の観測に対してノイズ注入と画像拡張を用いて知覚ギャップを閉じる。
5. 遅延補償とアクション平滑化往復制御遅延を測定し、予測バッファリングまたはスミス予測器を制御ループに適用する。高周波制御指令を平滑化してアクチュエータ帯域幅を尊重し、モデル化されていないモードの励起を回避する。
6. ハードウェアインザループ (HIL) と段階的展開リアルセンサまたはアクチュエータとリンクされたシミュレート環境でロボットコントローラを動作させる HIL セットアップを用いる。以下を経て現実世界への曝露を段階的に増やす：
 - ・単関節のベンチテスト、
 - ・拘束タスク（テザーまたは荷重軽減）、

- 安全停止付き監督試行、
- 完全非テザー運用。

実装パターン（実践的パイプライン）

- ステップ 1：センサとアクチュエータをキャリブレーションし、短いトルク/位置スイープを実行する。
- ステップ 2：システム ID を実行し、シミュレータパラメータ ϕ を更新する。
- ステップ 3： ϕ を中心としたダイナミクスランダムマイゼーションでポリシーを訓練する。
- ステップ 4：オンボード観測用に教え子ポリシーを蒸留する。
- ステップ 5：安全モニターとオンライン残差適応を伴ってデプロイする。

コード例：ランダム化パラメータ収集と訓練ループ

コードサンプル 66 Isaac Sim インターフェースを用いた sim-to-real ランダム化訓練ループ

```
import numpy as np
import torch
import torch.nn as nn
from typing import Dict, Tuple, Optional
import rclpy
from rclpy.node import Node
from isaacsim import SimulationApp, Robot
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
import yaml
import time
import logging

# ログ設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class IsaacSimEnv:
    """Isaac Sim環境ラッパー"""
    def __init__(self, config_path: str):
        with open(config_path, 'r') as f:
            self.cfg = yaml.safe_load(f)

        # Isaac Sim初期化
        self.sim_app = SimulationApp({"headless": self.cfg["headless"]})
        self.robot = Robot(self.cfg["robot_usd_path"])
        self.stage = self.sim_app.get_stage()
```

```

# 物理シーン設定
self._setup_physics_scene()

def _setup_physics_scene(self):
    """物理シーンの初期設定"""
    from pxr import UsdPhysics, PhysxSchema
    UsdPhysics.Scene.Define(self.stage, "/physicsScene")
    physxSceneAPI = PhysxSchema.PhysxSceneAPI.Apply(self.stage.GetPrimAtPath("/physicsSceneAPI").CreateEnableCCDAttr().Set(True))

def randomize_parameters(self, param_ranges: Dict[str, Tuple[float, float]]):
    """物理パラメータのランダムマイゼーション"""
    for param, (min_val, max_val) in param_ranges.items():
        if param == "torso_mass":
            self.robot.set_body_mass("torso", np.random.uniform(min_val, max_val))
        elif param == "foot_friction":
            for foot in ["left_foot", "right_foot"]:
                self.robot.set_material_friction(foot, np.random.uniform(min_val, max_val))
        elif param == "actuator_latency":
            self.robot.set_actuator_delay(np.random.uniform(min_val, max_val))

def reset(self) -> np.ndarray:
    """環境リセット"""
    self.sim_app.reset()
    return self._get_observation()

def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, Dict]:
    """1ステップ実行"""
    self.robot.apply_action(action)
    self.sim_app.step()

    obs = self._get_observation()
    reward = self._calculate_reward()
    done = self._check_termination()
    info = {}

    return obs, reward, done, info

def _get_observation(self) -> np.ndarray:

```

```

    """観測値取得"""
    joint_pos = self.robot.get_joint_positions()
    joint_vel = self.robot.get_joint_velocities()
    base_ori = self.robot.get_base_orientation()
    base_ang_vel = self.robot.get_base_angular_velocity()

    return np.concatenate([joint_pos, joint_vel, base_ori, base_ang_vel])

def _calculate_reward(self) -> float:
    """報酬計算"""
    velocity_reward = np.dot(self.robot.get_base_linear_velocity(), [1, 0, 0])
    energy_penalty = np.sum(np.square(self.robot.get_joint_efforts()))
    return velocity_reward - 0.001 * energy_penalty

def _check_termination(self) -> bool:
    """終了条件チェック"""
    base_height = self.robot.get_base_position()[2]
    return base_height < 0.3 # 転倒判定

class HardwareLogger(Node):
    """実ハードウェアデータ収集ノード"""
    def __init__(self):
        super().__init__('hardware_logger')
        self.data_buffer = []

    def collect_trajectories(self, duration: float = 10.0) -> Dict:
        """実機データ収集"""
        start_time = time.time()
        while (time.time() - start_time) < duration:
            # ROS 2 トピックからデータ取得
            joint_states = self._get_joint_states()
            imu_data = self._get_imu_data()

            self.data_buffer.append({
                'joint_positions': joint_states.position,
                'joint_velocities': joint_states.velocity,
                'imu_orientation': imu_data.orientation,
                'timestamp': time.time()
            })

```

```

        return {'trajectories': self.data_buffer}

def _get_joint_states(self):
    """ 関節状態取得 (ダミー実装) """
    from sensor_msgs.msg import JointState
    return JointState()

def _get_imu_data(self):
    """ IMUデータ取得 (ダミー実装) """
    from sensor_msgs.msg import Imu
    return Imu()

def apply_action_filter(action: np.ndarray, cutoff_freq: float = 10.0) -> np.ndarray:
    """ ローパスフィルタでアクションを平滑化 """
    # 簡単な指数移動平均フィルタ
    alpha = 2 * np.pi * cutoff_freq * 0.001 # 1msタイムステップ想定
    filtered_action = alpha * action + (1 - alpha) * action
    return filtered_action

def run_system_id(sim_env: IsaacSimEnv, real_data: Dict) -> Dict:
    """ システム同定でシムパラメータを更新 """
    # 簡単な最小二乗法によるパラメータ推定
    sim_trajectory = collect_sim_trajectory(sim_env, len(real_data['trajectories']))

    # パラメータ誤差を計算
    param_error = {}
    for key in ['joint_positions', 'joint_velocities']:
        sim_vals = np.array([t[key] for t in sim_trajectory])
        real_vals = np.array([t[key] for t in real_data['trajectories']])
        param_error[key] = np.mean(np.square(sim_vals - real_vals))

    # 誤差に基づいてシムパラメータを調整
    correction_factor = 1.0 + 0.1 * np.tanh(param_error['joint_positions'])
    return {'mass_correction': correction_factor}

def collect_sim_trajectory(sim_env: IsaacSimEnv, num_steps: int) -> list:
    """ シムから軌道データ収集 """
    trajectory = []
    obs = sim_env.reset()

```

```

for _ in range(num_steps):
    action = np.zeros(sim_env.robot.num_dof) # ゼロアクション
    obs, _, _, _ = sim_env.step(action)
    trajectory.append({
        'joint_positions': obs[:sim_env.robot.num_dof],
        'joint_velocities': obs[sim_env.robot.num_dof:2*sim_env.robot.num_dof]
    })

return trajectory

def main():
    """メイン訓練ループ"""
    rclpy.init()

    # 環境初期化
    env = DummyVecEnv([lambda: IsaacSimEnv("config/isaac_cfg.yaml")])

    # PPOエージェント設定
    policy = PPO(
        "MlpPolicy",
        env,
        learning_rate=3e-4,
        n_steps=2048,
        batch_size=64,
        n_epochs=10,
        gamma=0.99,
        gae_lambda=0.95,
        clip_range=0.2,
        verbose=1
    )

    # ハードウェアロガー初期化
    hw_logger = HardwareLogger()

    # 訓練ループ
    for episode in range(1000):
        # 物理パラメータランダムマイゼーション
        env.env_method(
            "randomize_parameters",
            {

```

```

        "torso_mass": (8.0, 12.0),
        "foot_friction": (0.6, 1.2),
        "actuator_latency": (0.005, 0.02),
    }
)

# PPO更新
policy.learn(total_timesteps=2048)

# 定期的に実機データでキャリブレーション
if episode % 50 == 0:
    real_data = hw_logger.collect_trajectories(duration=5.0)
    phi = run_system_id(env.envs[0], real_data)
    logger.info(f"Episode_{episode}:_System_ID_correction={phi}")

rclpy.shutdown()

if __name__ == "__main__":
    main()

```

実践的測定と検証

- 定量的成功指標を定義する：歩行安定性、ゼロモーメントポイント（ZMP）マージン、歩行あたりエネルギー、知覚検出率。
- システム ID またはドメインランダムマイゼーションの前後で性能を比較する A/B テストを実行する。
- 失敗モードを監視する：高周波振動、アクチュエータ過熱、安全でない重心逸脱。

設計上のトレードオフと運用上のリスク

- ランダムマイゼーションを広げると頑健性は向上するが学習が遅くなり公称タスクのピーク性能が低下する。
- 残差学習補正への過度の依存は収集データへの過学習リスクがある；オンライン適応保護を維持する。
- 安全エンベロープとウォッチドッグタイマはハードウェア制御ループに残す必要がある；ソフトウェアレベルの緩和策はハードウェア安全の代わりにならない。
- キャリブレーションサイクルはダウンタイムと同定中の損傷を避けるための慎重な実験設計を要する。

エンジニアリングへの影響：sim-to-real の実践を開発ライフサイクルに統合する。シミュレーションを継続的にキャリブレーションされるコンポーネントとして扱い、静的なオラクルとはしない。高速システム ID ツールと安全優先の段階的展開に投資し、フィールド故障を減らしヒューマノイドの信頼性を維持する。

18.3 実環境でのテスト

実環境でのテストは、シミュレーション上の仮説をハードウェア上で検証された挙動に変換するエンジニアリングプロセスである。問題定義は単純である：訓練済みポリシーや知覚モデルが、物理的な外乱を受けた際に性能・安全性・頑健性要件を満たすかを測定する。制約は実用的である：限られた試験時間、人員とハードウェアの安全、反復改良のための高忠実度ロギングの必要性。

技術解析は、測定可能な受け入れ基準の定義から始まる。ヒューマノイドの歩行・操縦における典型的な指標は以下の通り：

- ・関節角度または重心軌跡の二乗平均平方根追従誤差（RMSE）。
- ・スクリプト化されたタスクの完遂成功率。
- ・ピークトルク、モータ温度、エネルギー消費。
- ・安全違反：転倒回数、衝突回数、非常停止回数。

離散的にサンプルされた連続エラー信号 $e(t)$ に対して、RMSE は

$$[H]RMSE = \sqrt{\frac{1}{N} \sum_{k=1}^N e[k]^2}. \quad (157)$$

として計算する。ここで N はサンプル数である。センサ間で同期されたタイムスタンプを用い、 $e[k]$ が整列した信号同士を比較できるようにする。

稀な故障確率を区切るために必要な試行回数を推定する。二項分布の故障確率 p 、所望の半幅 ε 、信頼度に対する標準正規分布の分位点 z に対して、

$$[H]n \approx \frac{z^2 p(1-p)}{\varepsilon^2}. \quad (158)$$

と近似する。これは破局的故障率を定量化する際の実騫回数の実用的な下限を与える。

実装指向のプラクティス。低リスクから完全自律へ段階的に進める：

1. ベンチ・コンポーネント試験
 - ・アクチュエータ、エンコーダ、各センサを独立して検証。
 - ・想定デューティサイクルに対して静的トルク・熱試験を実施。
2. ハードウェアインザループ（HIL）
 - ・一部センサをシミュレート入力で置換しコントロールスタックに負荷をかける。
 - ・実地試験前にタイミング、レイテンシ、パケットロスの影響を試験。
3. テザーまたは拘束試験
 - ・転倒用セーフティハーネス、関節リミット用ソフトメカニカルストップを使用。
 - ・アクチュエータ指令の大きさと速度を制限。
4. 段階的タスク複雑度
 - ・初期はフラットで既知の路面にフィジューシャル付きで局所化。
 - ・徐々に傾斜、障害物、移動物体を導入。
5. データ収集とオンライン適応
 - ・システム同定・ドメイン適応微調整のための豊富なデータを記録。

計測・ロギング詳細：

- 同期クロック：PTP または ROS タイムスタンプでサブミリ秒整列。
- データレート：IMU は ≥ 200 Hz、関節エンコーダはネイティブコントローラレート、ビジョンはモデル訓練レートでサンプリング。
- ストレージ：生センサストリームをローカルにバッファし、要約テレメトリをリモートサーバへストリーム。
- メタデータ：コントローラバージョン、モデルチェックポイント、キャリブレーションパラメータを含める。

センサキャリブレーションは試験前に実施。以下を実行：

- 静止・回転動作を用いた IMU バイアス・スケールキャリブレーション。
- 既知の機械的ポーズでの関節エンコーダゼロ合わせ。
- 既知の荷重による力・トルク（FT）センサオフセットキャリブレーション。
- チェッカーボードまたは AprilTag リグを用いたカメラ内部・外部パラメータ。

制御・安全層：複数の独立ウォッチドッグを実装。

- 故障時にリンプまたはブレーキを指令する高レートハードウェアウォッチドッグ。
- 安全述語をチェックする中レートソフトウェア安全監視。
- 即時手動オーバーライド用の低レートヒューマンインザループチャンネル。

故障モード解析と故障注入実験で脆い条件を特定。制御遅延、センサドロップアウト、摩擦係数変更を注入。故障トレースを収集し、教師ありまたは RL ベース回復ポリシー学習用にラベル付け。

オンラインファインチューニングと継続訓練。収集した実環境データを用いて：

- システム同定パラメータを再計算し物理モデルを更新。
- 実画像・センサノイズパターンで知覚モジュールを再訓練。
- 安全探索制約付きで保守的 RL ファインチューニングを適用。

エンジニアリングトレードオフはサンプリングと安全性に現れる。例えば、網羅的故障試験はリスクを低減するが時間とハードウェア寿命を消費する。逆に、最小限試験は展開を高速化するが運用リスクを増大させる。

式 (2) による最小標本数計画はコストと統計的信頼性のバランスに貢献。重点サンプリングまたは高 σ シナリオへの集中ストレステストを用いて必要試験回数を削減。

テストハーネス例。以下のリストはタスクを循環し安全トピックを監視しテレメトリをログする簡潔な ROS ベーステストハーネスを示す。インラインコメントは簡潔で実用的。

コードサンプル 67 ROS2 test harness for staged real-world trials

```
#!/usr/bin/env python3
import os
import signal
import threading
import time
from pathlib import Path
```

```

from typing import Callable, Optional

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Bool
from sensor_msgs.msg import Imu

# QoS設定：センサデータ用
IMU_QOS = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=100,
)

class TestHarness(Node):
    def __init__(self) -> None:
        super().__init__('test_harness')
        self._shutdown_evt = threading.Event()

        # 緊急停止監視
        self.emergency_sub = self.create_subscription(
            Bool, '/safety/emergency', self._emergency_cb, 10)

        # IMU受信
        self.imu_sub = self.create_subscription(
            Imu, '/imu/data', self._imu_cb, IMU_QOS)

        # ログディレクトリ作成
        log_dir = Path('/var/log/humanoid_tests')
        log_dir.mkdir(parents=True, exist_ok=True)
        self.log_path = log_dir / f'session_{int(time.time())}.log'
        self.log_file = self.log_path.open('a', buffering=1)

# line buffering

        self.emergency = False
        self._lock = threading.Lock()

        # シグナルハンドラ登録
        signal.signal(signal.SIGINT, self._signal_handler)

```

```

signal.signal(signal.SIGTERM, self._signal_handler)

def _signal_handler(self, signum, frame) -> None:
    self.get_logger().info('Shutdown_signal_received')
    self._shutdown_evt.set()

def _emergency_cb(self, msg: Bool) -> None:
    with self._lock:
        self.emergency = msg.data
    if self.emergency:
        self.get_logger().warn('Emergency_stop_triggered')

def _imu_cb(self, msg: Imu) -> None:
    with self._lock:
        if self.emergency:
            return
        # タイムスタンプと加速度をログ
        ts = msg.header.stamp.sec + msg.header.stamp.nanosec * 1e-9
        self.log_file.write(
            f'{ts},{msg.linear_acceleration.x}, '
            f'{msg.linear_acceleration.y},{msg.linear_acceleration.z}\n')

def run_trial(self, task_fn: Callable[[], None], timeout_s: float = 30.0) -> bool:
    if self._shutdown_evt.is_set():
        return False

    start = self.get_clock().now()
    task_fn()

    while (self.get_clock().now() - start).nanoseconds * 1e-9 < timeout_s:
        if self._shutdown_evt.wait(0.01):
            return False
        with self._lock:
            if self.emergency:
                self.get_logger().error('Abort:_emergency_stop')
                return False
    return True

def destroy_node(self) -> None:
    self.log_file.close()

```

```

        super().destroy_node()

def main(args=None) -> None:
    rclpy.init(args=args)
    node = TestHarness()
    try:
        # 実際のtask_fnは呼び出し側で定義
        # node.run_trial(task_fn)
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

運用への影響、トレードオフ、リスク：

- トレードオフ：積極的オンライン適応は性能を改善するが、強力な安全制約なしでは不安定化リスクを増大させる。
- 設計への影響：単一故障点を減らすため高レート・冗長センシングと独立ウォッチドッグに投資する。
- 運用リスク：故障確率の推計が不適切だとエッジケースの試験不足につながる。稀だが破局的なモードに対して標的ストレステストを用いる。

これらのプラクティスを採用し検証ループを閉じる：センサをキャリブレーションし、段階的に試験をステージングし、幅広くログし、実環境データをモデル改良に活用する。これらのステップは展開リスクを低減し、試験カバレッジ・ハードウェア寿命・運用安全性のエンジニアリングトレードオフを浮き彫りにする。

18.4 継続的なトレーニングサイクル

実世界での実用的なテストとシミュレーション・リアリティミスマッチを低減する手法に続き、継続的なトレーニングサイクルは、人型コントローラがドリフト、摩耗、環境変化に対して頑健であることを維持するフィードバックループを形式化する。このサイクルは、構造化されたデータキャプチャ、ドメイン適応型シミュレーション更新、オンラインまたはオフラインでのファインチューニング、そして安全ゲート付きデプロイメントを結びつける。

問題定義。人型ロボットは非定常かつ部分的に観測可能な環境で動作する。センサは劣化し、接触は変化し、タスクは進化する。シミュレーションからハードウェアへの単一の転移は時間とともに劣化する。継続的なトレーニングサイクルは、知覚、状態推定、および制御ポリシーを、現実世界のデータをシミュレーションおよびトレーニングパイプラインに繰り返し組み込むことで最新の状態に保つことを目指す。エンジニアリングの目標は、ハードウェアおよび人間へのリスクを限定しながら、長期的な運用信頼性を最大化することである。

技術的分析。実用的な継続的サイクルには、以下のモジュラーな段階がある：

- データ取得：同期されたセンサログ、作動指令、および文脈メタデータ。
- データ検証およびラベリング：自動化された健全性チェック、自己教師付きラベル（例：接触イベント）、およびエッジケースに対する人間のレビュー。
- ドメイン差分推定：実観測がシミュレートされた分布とどれほど異なるかを定量的に測定。
- シミュレーション改良：シミュレータパラメータまたはランダム化範囲を更新して、観測された実状態をよりよくカバーする。
- モデル適応：シミュレートおよび実データセットを組み合わせて使用し、知覚および制御コンポーネントをファインチューニングする。
- 安全検証およびデプロイメント：段階的なロールアウト、シャドウテスト、およびランタイムコントローラへのゲート付きデプロイメント。

差分を測定することは重要である。実観測分布とシミュレート観測分布間の発散メトリクスを使用する。2つの分布 p_{real} および p_{sim} に対して、カルバック-ライブラー発散は

$$[H]D_{\text{KL}}(p_{\text{real}}\|p_{\text{sim}}) = \int p_{\text{real}}(x) \log \frac{p_{\text{real}}(x)}{p_{\text{sim}}(x)} dx, \quad (159)$$

であり、これは実世界サンプルの下での期待対数尤度比を定量化する。実際には、生の高次元センサデータではなく、コンパクトで代表的な特徴埋め込み上で D_{KL} を推定する。最大平均差分（MMD）または分類器2標本検定は、限られたサンプルサイズの下で頑健な代替手段を提供する。

重要度重み付けは、ドメイン間のサンプリングバイアスを補償する。 $w(x) = p_{\text{real}}(x)/p_{\text{sim}}(x)$ によってシミュレート損失項を再重み付けすることで、教師付きファインチューニングにおけるバイアスを低減する。強化学習を使用してポリシーを更新する場合、実世界の動作を不安定化することを避けるために更新を制約する。典型的な制約付き目的は

$$\begin{aligned} [H] \quad & \underset{\pi}{\text{maximize}} \quad \mathbb{E}_{s \sim \mathcal{D}}[J(\pi)] \\ & \text{subject to} \quad D_{\text{KL}}(\pi\|\pi_{\text{old}}) \leq \delta, \end{aligned} \quad (160)$$

であり、これは信頼領域制約を課す。実装は、安定した漸進的更新のためにラグランジュまたは KL ペナルティ形式（PPO スタイル）を使用する。

実装ブループリント。継続的サイクルは、再現可能なパイプラインと頑健なツールを必要とする：

1. データバージョンニング：データセットにロボットシリアル、タイムスタンプ、ファームウェア、および環境メタデータをタグ付けする。
2. シミュレーションパラメータ化：摩擦、遅延、センサノイズ、およびコンプライアンスのためのコンパクトなパラメータベクトル θ_{sim} を公開する。
3. モデルトレーニングワークフロー：シミュレートおよび実サンプルの両方を含む混合リプレイバッファから段階的ファインチューニングをサポートする。
4. 安全ゲート：自動化されたチェックには、トルクの有界テスト、シャドウモード比較、およびフェイルセーフコントローラが含まれる。

Python での最小限のオーケストレーションループが、コアシーケンスを示す。このスニペットは、Isaac Sim およびモデルトレーニングバックエンドとの統合を目的としている。

コードサンプル 68 継続的トレーニングループ（簡略化）

```

#!/usr/bin/env python3
import random
import time
from dataclasses import dataclass
from typing import List, Tuple

import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset as TorchDataset, DataLoader

# ROS 2 (オプション)
try:
    import rclpy
    from rclpy.node import Node
    ROS2_AVAILABLE = True
except ImportError:
    ROS2_AVAILABLE = False

# ロギング
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# 定数
DIVERGENCE_THRESHOLD = 0.25
MAX_REPLAY_SIZE = 100_000
BATCH_SIZE = 256
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# データ構造
@dataclass
class SensorCommand:
    timestamp: float
    sensor: np.ndarray
    command: np.ndarray

@dataclass
class Batch:
    seed: int

```

```

data: List[SensorCommand]
stats: dict

@property
def emb(self) -> torch.Tensor:
    # 簡易埋め込み：sensorとcommandを連結
    return torch.tensor(np.vstack([np.hstack([d.sensor, d.command]) for d in self.
                                   dtype=torch.float32, device=DEVICE])

class ReplayBuffer(TorchDataset):
    def __init__(self, max_size: int = MAX_REPLAY_SIZE):
        self.buffer: List[Tuple[Batch, Batch]] = []
        self.max_size = max_size

    def add(self, real: Batch, sim: Batch):
        self.buffer.append((real, sim))
        if len(self.buffer) > self.max_size:
            self.buffer.pop(0)

    def __len__(self):
        return len(self.buffer)

    def __getitem__(self, idx):
        real, sim = self.buffer[idx]
        return real.emb, sim.emb

    def sample(self, batch_size: int = BATCH_SIZE):
        real_embs, sim_embs = zip(*random.sample(self.buffer, batch_size))
        return torch.stack(real_embs), torch.stack(sim_embs)

# モデル
class DivergenceEstimator(nn.Module):
    def __init__(self, input_dim: int):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)

```

```

        ).to(DEVICE)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)

# ロボットインターフェース
class RealRobot:
    def __init__(self, node_name: str = "real_robot_node"):
        if ROS2_AVAILABLE:
            rclpy.init()
            self.node = Node(node_name)
        else:
            self.node = None

    def record(self, duration: float) -> Batch:
        # 実機からセンサ+指令を収集
        data = []
        start = time.time()
        while (time.time() - start) < duration:
            # ダミー読み取り
            sensor = np.random.randn(10)
            command = np.random.randn(5)
            data.append(SensorCommand(time.time(), sensor, command))
            time.sleep(0.01)
        stats = {"mean_sensor": np.mean([d.sensor for d in data], axis=0)}
        seed = random.randint(0, 2**31 - 1)
        return Batch(seed=seed, data=data, stats=stats)

# シミュレータ
class Simulator:
    def __init__(self):
        self.params = {"friction": 0.5, "delay": 0.0}

    def generate(self, seed: int) -> Batch:
        # シード固定で再現性確保
        random.seed(seed)
        np.random.seed(seed)
        data = []
        for _ in range(6000): # 60秒分 @ 100Hz
            sensor = np.random.randn(10) * self.params["friction"]

```

```

        command = np.random.randn(5) * (1 + self.params["delay"])
        data.append(SensorCommand(time.time(), sensor, command))
    stats = {"mean_sensor": np.mean([d.sensor for d in data], axis=0)}
    return Batch(seed=seed, data=data, stats=stats)

def adapt(self, real_stats: dict):
    # 摩擦・遅延を更新
    self.params["friction"] *= np.random.uniform(0.9, 1.1)
    self.params["delay"] = max(0.0, self.params["delay"] + np.random.uniform(-0.01, 0.01))

# 損失関数
def compute_divergence(real_emb: torch.Tensor, sim_emb: torch.Tensor) -> float:
    # MMD簡易実装
    xx = torch.mean(torch.mm(real_emb, real_emb.T))
    yy = torch.mean(torch.mm(sim_emb, sim_emb.T))
    xy = torch.mean(torch.mm(real_emb, sim_emb.T))
    mmd = xx + yy - 2 * xy
    return mmd.item()

# ファインチューニング
def fine_tune(model: nn.Module, dataset: ReplayBuffer, lr: float = 1e-4, steps: int = 10000):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    loader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
    for epoch in range(steps // len(loader) + 1):
        for real_emb, sim_emb in loader:
            optimizer.zero_grad()
            loss = compute_divergence(real_emb, sim_emb)
            torch.tensor(loss).backward()
            optimizer.step()
    return model

# 安全検証
def safety_validate(model: nn.Module, shadow_logs: List[Batch], thresholds: dict) -> bool:
    for log in shadow_logs:
        sim_log = Simulator().generate(seed=log.seed)
        div = compute_divergence(log.emb, sim_log.emb)
        if div > thresholds.get("max_div", 0.3):
            return False
    return True

```

```

# デプロイ
def deploy(model: nn.Module):
    logger.info("モデルを本番環境へデプロイ")
    torch.save(model.state_dict(), "/tmp/deployed_model.pt")

# メイン
def main():
    robot = RealRobot()
    sim = Simulator()
    dataset = ReplayBuffer()
    model = DivergenceEstimator(input_dim=15)
    shadow_logs = [robot.record(10) for _ in range(5)] # シャドウ用ログ

    while True:
        real_batch = robot.record(duration=60)
        sim_batch = sim.generate(seed=real_batch.seed)
        div = compute_divergence(real_batch.emb, sim_batch.emb)
        if div > DIVERGENCE_THRESHOLD:
            sim.adapt(real_batch.stats)
            dataset.add(real_batch, sim_batch)
            model = fine_tune(model, dataset)
            if safety_validate(model, shadow_logs, {"max_div": 0.3}):
                deploy(model)
            time.sleep(1) # 次サイクルまで待機

if __name__ == "__main__":
    main()

```

実用的な考慮事項とトレードオフ：

- 再トレーニングの頻度は、計算コストと適応速度のバランスを取る。高頻度更新はラグを減らす
が、一時的条件への過学習リスクを増加させる。
- リプレイバッファは、壊滅的忘却を避けるために多様なシミュレートデータを含まなければならない。
弾性重み固執などの正則化技術は、以前に学習した動作を保持する。
- 安全ゲーティングは必須である。常にフォールバックコントローラと保守的な運用制限を維持
する。
- ハードウェアストレス：継続的サイクルは作動時間を増加させる。累積作動メトリクスに基づい
て校正およびメンテナンスをスケジュールする。
- 解釈可能性および監査可能性：認証およびインシデント分析をサポートするために、モデル
チェックポイント、データセットスナップショット、および検証テストスイートをバージョン管
理する。

運用上のリスクおよびエンジニアリングへの影響：

- p_{sim} の誤推定は、重要度重みにバイアスを生じ、ポリシーを劣化させる。
- 最近の実データへの過学習は、ロングテール条件への汎化を減少させる。
- 不十分な安全テストは、不安定化ポリシー更新がハードウェアに到達することを許可する。
- 計算およびストレージコストはデータ量に比例してスケールする。インフラを適切に計画する。

設計者は、適応速度、安全マージン、および運用コストのトレードオフを取らなければならない。規律ある継続的トレーニングサイクルは、ハードウェアの寿命および人間の安全性を尊重しながら、長期的な故障率を最小化する。

知覚とビジョンシステム

19 コンピュータビジョンの基礎

19.1 物体検出と分類

これまでに紹介した視覚処理の基礎を踏まえ、本小節ではヒューマノイドロボット向けの実用的な物体検出と分類に焦点を当てる。センシング問題を定式化し、主要指標とアルゴリズムを導出し、リアルタイム推論の実装スケッチを示し、エンジニアリング上のトレードオフをまとめる。

問題定義：ヒューマノイドは、操縦、ナビゲーション、またはインタラクションを実行するために、雑然とした動的環境で物体を検出し分類しなければならない。検出は物体を局所化する（バウンディングボックスまたはマスク）。分類は意味ラベルを割り当てる。ヒューマノイド特有の制約として、限られた計算資源、バランスと操縦のための厳しい遅延制約、頭部搭載カメラによる至近距離の視点歪みがある。

技術分析：

- タスク分解：
 1. ステレオまたは RGB-D センサからの画像取得と前処理。
 2. 候補生成（領域提案またはシングルショット予測器）。
 3. 局所化回帰とクラス予測。
 4. 後処理：非最大値抑制（NMS）、信頼度閾値処理、深度の関連付け。
- 局所化品質は Intersection-over-Union（IoU）で量化される。予測ボックス P と正解ボックス G に対する IoU を定義する：

$$[H]\text{IoU}(P, G) = \frac{\text{area}(P \cap G)}{\text{area}(P \cup G)}. \quad (161)$$

IoU 閾値は評価時の真陽性受容と NMS に用いられる。

- 1 段階および 2 段階検出器の一般的な損失分解：

$$[H]L = L_{\text{cls}} + \lambda L_{\text{loc}}, \quad (162)$$

ここで L_{cls} はクラス不均衡に対するクロスエントロピーまたは focal 損失、 L_{loc} は Smooth- L_1 または IoU ベースの局所化損失である。 λ を調整することで、分類信頼度とボックス精度のバランスを取る。

- 検出パイプラインとトレードオフ：
 - 2 段階 (Faster R-CNN)：高精度，遅い遅延．把持計画で精度が臨界的な場合に有用．
 - 1 段階 (YOLO, SSD)：低遅延，ピーク精度は低い．閉ループ反応性を要するヒューマノイドに好まれる．
 - アンカーフリーメソッドはハイパーパラメータを削減するが，ヒューマン環境に多い極端なアスペクト比で苦勞することがある．

実装上の考慮事項：

- ヒューマノイドでのリアルタイム推論は，オンボード NVIDIA GPU または Jetson モジュールで実行されることが多い．TensorRT への変換で遅延を削減する．
- 深度アライメントを用いて 2D 検出を 3D 物体重心に変換する．バウンディングボックス中心 (u, v) と深度 z に対し，カメラ座標系での点は

$$[H]\mathbf{p} = zK^{-1}[u \ v \ 1]^T, \quad (163)$$

カメラ内部パラメータ行列 K による．

- 時空間フィルタリング（カルマンまたは単純指数平滑化）は操縦のための検出を安定化する．IMU または運動学的事前情報を融合し，動作中の偽検出を拒絶する．

実用的アルゴリズムパイプライン：

- 前処理：色正規化，アスペクト保持リサイズ，照明ばらつきのための拡張．
- 推論：検出器を実行；score_threshold を適用し低信頼度出力をフィルタ．
- 後処理：IoU 閾値 nms_iou で NMS を実行し重複を除去．
- 深度関連付け：ボックス内の中央値深度をサンプリングし 3D 姿勢を計算；信頼できる深度がないボックスを拒絶．

ヒューマノイドベンチマークで監視すべき評価指標：

- クラスごとの精度，再現率，平均精度 (AP)．
- ロボット動作下での推論時間の遅延と分散．
- 偽陽性の空間分布（安全性に臨界的）．

オンボード GPU 向けの最適化推論ループを示すコードリスト．このスニペットは TensorRT 最適化検出器と深度画像ストリームを仮定する．

コードサンプル 69 組込み GPU でのリアルタイム検出推論 (TensorRT)

```
import time
import numpy as np
import cv2
import torch
import torchvision
import rclpy
from rclpy.node import Node
```

```

from sensor_msgs.msg import Image
from vision_msgs.msg import Detection3D, Detection3DArray, ObjectHypothesisWithPose
from cv_bridge import CvBridge
from geometry_msgs.msg import Point
import threading
import queue

class TrtDetector(Node):
    def __init__(self, engine_path, K, score_th=0.3, nms_iou=0.45, target_size=(640, 3
        super().__init__('trt_detector')
        self.K = K
        self.score_th = score_th
        self.nms_iou = nms_iou
        self.target_size = target_size
        self.bridge = CvBridge()

        # TensorRT初期化
        import tensorrt as trt
        import pycuda.driver as cuda
        import pycuda.autoinit
        with open(engine_path, 'rb') as f, trt.Runtime(trt.Logger()) as runtime:
            self.engine = runtime.deserialize_cuda_engine(f.read())
        self.context = self.engine.create_execution_context()
        self.d_input = cuda.mem_alloc(np.zeros((1, 3, *target_size), np.float32).nbytes)
        self.d_output = [cuda.mem_alloc(np.zeros((1, *shape), np.float32).nbytes)
                           for shape in [(25200, 4), (25200,), (25200,)]]

# YOLOv5例
        self.stream = cuda.Stream()

        # 同期キュー
        self.rgb_q = queue.Queue(maxsize=5)
        self.depth_q = queue.Queue(maxsize=5)

        self.pub = self.create_publisher(Detection3DArray, 'detections_3d', 10)
        self.create_subscription(Image, 'rgb', self.rgb_cb, 1)
        self.create_subscription(Image, 'depth', self.depth_cb, 1)
        threading.Thread(target=self.infer_loop, daemon=True).start()

    def rgb_cb(self, msg):
        try:

```

```

        self.rgb_q.put(self.bridge.imgmsg_to_cv2(msg, 'bgr8'), block=False)
except queue.Full:
    pass # ドロップ

def depth_cb(self, msg):
    try:
        self.depth_q.put(self.bridge.imgmsg_to_cv2(msg, '32FC1'), block=False)
    except queue.Full:
        pass

def preprocess(self, rgb):
    img = cv2.resize(rgb, self.target_size)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).astype(np.float32) / 255.0
    return img.transpose(2, 0, 1)[None]

def postprocess(self, boxes, scores, classes):
    keep = scores > self.score_th
    boxes, scores, classes = boxes[keep], scores[keep], classes[keep]
    idx = torchvision.ops.nms(torch.from_numpy(boxes), torch.from_numpy(scores), scores)
    return boxes[idx].numpy(), scores[idx].numpy(), classes[idx].numpy()

def infer_loop(self):
    import pycuda.driver as cuda
    while rclpy.ok():
        try:
            rgb = self.rgb_q.get(timeout=0.1)
            depth = self.depth_q.get(timeout=0.1)
        except queue.Empty:
            continue
        t0 = time.time()
        x = self.preprocess(rgb)
        np.copyto(cuda.pagelocked_empty((1,3,*self.target_size), np.float32), x)
        cuda.memcpy_htod_async(self.d_input, x, self.stream)
        self.context.execute_async_v2([int(self.d_input)] + [int(d) for d in self.d_output])
        out = []
        for d in self.d_output:
            tmp = np.empty((1, 25200), np.float32)
            cuda.memcpy_dtoh_async(tmp, d, self.stream)
            out.append(tmp)
        self.stream.synchronize()

```

```

boxes, scores, classes = self.postprocess(out[0].reshape(-1,4), out[1].ravel())
msg = Detection3DArray()
msg.header.stamp = self.get_clock().now().to_msg()
for b, s, c in zip(boxes, scores, classes):
    u0, v0, u1, v1 = map(int, b)
    roi = depth[v0:v1, u0:u1]
    z = np.median(roi[np.isfinite(roi)])
    if z > 0.2:
        xyz = z * np.linalg.inv(self.K) @ np.array([(u0+u1)/2, (v0+v1)/2, 1])
        det = Detection3D()
        det.results.append(ObjectHypothesisWithPose())
        det.results[0].hypothesis.class_id = str(int(c))
        det.results[0].hypothesis.score = float(s)
        det.bbox.center.position = Point(x=float(xyz[0]), y=float(xyz[1]), z=float(xyz[2]))
        msg.detections.append(det)
self.pub.publish(msg)
self.get_logger().debug(f'latency:{time.time()-t0:.3f}s')

def main():
    rclpy.init()
    K = np.array([[610, 0, 320], [0, 610, 240], [0, 0, 1]]) # 例
    node = TrtDetector('yolo.engine', K)
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計トレードオフと運用上のリスク：

- 遅延対精度：低遅延モデルは制御ラグを減らす、把持中の誤分類リスクを増やす。
- 信頼度較正：過信モデルは安全でない動作を引き起こす；低信頼モデルはタスクスループットを低下させる。
- 深度信頼性：反射または透明表面は深度を劣化させ、誤った 3D 局所化を引き起こす。
- ヒューマノイドの電力および熱制約は連続高レート推論を制限し、運動状態に連動した可変レートで知覚をスケジュールする。

エンジニアは `score_threshold` と `nms_iou` をタスクごとに調整し、エンドツーエンド遅延をプロファイリングし、一時的な偽陽性を軽減するための時空間融合を展開すべきである。堅牢性は、実世界ヒューマン環境、合成拡張、安全なフォールバック動作のための明示的故障モードを組み合わせた評価を要する。

19.2 奥行き知覚とステレオビジョン

物体検出・分類を基盤とし、奥行き知覚は人型ロボットに対して確実な把持、衝突回避、バランス制御を可能にする空間的な文脈を供給する。以下の記述は、頭部搭載センサリグ、低遅延制御ループ、慣性センシングとの統合に関連するステレオビジョンの詳細に焦点を当てる。

問題定義と運用要件：

- ・人型ロボットは典型的なインタラクション範囲（0.3–5 m）にわたる高密度かつメトリックな奥行き推定を必要とする。
- ・制約としては、頭部ベースラインの制限、組み込み GPU の計算予算、動く人を含む動的シーン、可変照明が含まれる。
- ・安全上重要な動作（例：歩行、ハンドオーバー）は、既知の奥行き不確実性とリアルタイム信頼度指標を要求する。

技術分析 — 幾何学的基礎と誤差伝播：ステレオ奥行きは、2 台の校正済みピンホールカメラ間の三角測量から生じる。整直後、対応する水平画素座標 x_L と x_R は視差 $d = x_L - x_R$ を生じる。奥行き z は

$$z = \frac{f B}{d}, \quad (164)$$

ここで、 f はピクセル単位の焦点距離、 B はメートル単位のベースラインである。視差検出誤差 σ_d は奥行き不確実性に二次関数的に伝播する：

$$\sigma_z \approx \frac{f B}{d^2} \sigma_d = \frac{z^2}{f B} \sigma_d. \quad (165)$$

設計上の示唆：小さなベースラインは遠方で σ_z を劇的に増大させ、より大きな焦点距離は視野を狭める代償で奥行き分解能を改善する。

人型頭部のためのパイプラインとアルゴリズム：

1. 校正と整直

- ・各カメラの内部パラメータとカメラ間の外部パラメータを推定する。チェッカーボードまたは AprilTag をバンドル調整で使用する。camera_matrix と distortion_coeffs を保存する。
- ・対極線整列を強制する整直マップを計算する。これにより対応探索を 1 次元に削減する。

2. 対応付けマッチング

- ・古典的な選択肢：ブロックマッチングと半大域マッチング（SGM）。SGM は精度と計算のバランスを取り、OpenCV の StereoSGBM はリアルタイムパイプラインで一般的。
- ・学習ベース手法（PSMNet、GANet）は低テクスチャ・反射領域で改善された結果を提供するが、GPU と注意深いドメイン適応を要求する。

3. 後処理

- ・左右一致性チェックで遮蔽をマスクする。
- ・マッチングコスト、一意性比率、正規化相互相関を用いた信頼度推定。
- ・制御ループ向けに奥行きを安定化する時間フィルタリング（例：指数平滑化またはカルマンフィルタリング）。

4. 融合と上位レベル利用

- ステレオ奥行きを IMU 駆動の運動事前情報と融合し、高速頭部動作中の対応を解決する。
- ステレオを能動的奥行きセンサと統合し、無テクスチャまたは鏡面表面を処理する。

実装上の注意：

- ベースライン選択：6–12 cm は人間スケールのインタラクションに適し、長距離タスクでは機械的マウントと美的制約を保ちながらベースラインを増加させる。
- 露光と同期：グローバルシャッターカメラとハードウェア同期は動作誘起視差誤差を劇的に削減する。
- 校正維持：熱ドリフトと機械的たわみは定期的な再校正または運動連鎖内でのオンライン自己校正ルーチンを要求する。

実装例（組み込み GPU 上の OpenCV；最小パイプライン）：

コードサンプル 70 OpenCV StereoSGBM を用いた基本的なステレオ奥行きパイプライン

```
import cv2
import numpy as np
from pathlib import Path
from typing import Tuple, Optional

class StereoDepthEstimator:
    """
    ステレオカメラからの深度推定クラス
    """
    def __init__(self, calib_path: str, max_disp: int = 128):
        # キャリブレーションデータ読み込み
        calib = np.load(calib_path)
        self.left_map1 = calib['left_map1']
        self.left_map2 = calib['left_map2']
        self.right_map1 = calib['right_map1']
        self.right_map2 = calib['right_map2']
        self.Q = calib['Q']

        # SGBMマッチャ初期化（人型頭部用パラメータ）
        self.matcher = cv2.StereoSGBM_create(
            minDisparity=0,
            numDisparities=max_disp,
            blockSize=5,
            P1=8 * 3 * 5 ** 2,
            P2=32 * 3 * 5 ** 2,
            mode=cv2.STEREO_SGBM_MODE_HH
        )
```

```

# WLSフィルタ（平滑化用）
self.wls_filter = cv2.ximgproc.createDisparityWLSFilter(self.matcher)
self.wls_filter.setLambda(8000)
self.wls_filter.setSigmaColor(1.5)

# 視差信頼度閾値
self.conf_thresh = 1.0 # 無効視差閾値

def compute_depth(self, left_img: np.ndarray, right_img: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """
    ステレオ画像から深度マップを計算
    Returns:
    depth: 奥行き画像（信頼度低いピクセルはnan）
    disp: 視差画像（信頼度低いピクセルはnan）
    """
    # 整直変換
    left_rect = cv2.remap(left_img, self.left_map1, self.left_map2, cv2.INTER_LINEAR)
    right_rect = cv2.remap(right_img, self.right_map1, self.right_map2, cv2.INTER_LINEAR)

    # 視差計算
    disp_left = self.matcher.compute(left_rect, right_rect).astype(np.float32) / 16

    # 右視差（信頼度マスク用）
    right_matcher = cv2.ximgproc.createRightMatcher(self.matcher)
    disp_right = right_matcher.compute(right_rect, left_rect).astype(np.float32) / 16

    # WLSフィルタ適用
    disp_filtered = self.wls_filter.filter(disp_left, left_rect, None, disp_right)

    # 信頼度マスク生成
    conf_mask = self.wls_filter.getConfidenceMap()
    valid_mask = (disp_filtered > self.conf_thresh) & (conf_mask > 0.0)

    # 無効ピクセルをnanに
    disp_filtered[~valid_mask] = np.nan

    # 3D再投影
    points = cv2.reprojectImageTo3D(disp_filtered, self.Q)
    depth = points[..., 2]

```

```

        # 深度無効化
        depth[~valid_mask] = np.nan

    return depth, disp_filtered

# 使用例
if __name__ == "__main__":
    estimator = StereoDepthEstimator("stereo_rect_maps.npz")

    left = cv2.imread("left.png", cv2.IMREAD_GRAYSCALE)
    right = cv2.imread("right.png", cv2.IMREAD_GRAYSCALE)

    depth, disp = estimator.compute_depth(left, right)

    # 深度可視化（有効ピクセルのみ）
    valid_depth = depth[~np.isnan(depth)]
    if valid_depth.size > 0:
        depth_vis = cv2.normalize(depth, None, 0, 255, cv2.NORM_MINMAX, dtype=cv2.CV_8U)
        cv2.imshow("Depth", depth_vis)
        cv2.waitKey(0)

```

設計上のトレードオフと運用上のリスク：

- 精度 vs. 遅延：高品質マッチャとニューラル手法は遅延を増大させる。歩行には低遅延・適度な精度と時間平滑化を優先する。
- ベースライン vs. 遮蔽：大きなベースラインは遠方奥行きを改善するが、近傍遮蔽と首へのマウントトルクを増加させる。
- 照明と表面特性：ステレオは無特徴または鏡面表面で失敗する；冗長性のためフォールバックセンサ（TOF、構造化光）を含める。
- 校正と機械的摩耗：誤校正は系統的奥行きバイアスを生じ、バランスと操縦タスクを損なう。オンライン一貫性チェックと監督再校正トリガを実装する。

人型システムへの具体的影響：

- 下流プランナにピクセル単位信頼度マップを提供し、信頼できない奥行きに基づく制御動作を回避する。
- 高速動作にはステレオ・慣性融合を用い、短期間の対応喪失を許容する。
- オンザフライ左右チェックと時間フィルタリングのための計算予算を確保し、障害物検出の誤陽性を削減する。

19.3 視覚処理における主要な課題

ステレオ深度キューもフレームごとの物体分類器についての前述の議論は、相補的な 2 つの要求、すなわちセンサ雑音下での頑健な幾何推定と外観変動下での信頼できる意味解釈を浮き彫りにした。本小節では、ヒューマノイド視覚システムにおいてそれらの要求を橋渡しする運用上の課題を分析する。

ヒューマノイドロボットは制約のあるセンサエンベロープと高度に変動する環境に直面する。主要な課題は知覚、計算、統合のカテゴリに集約される。知覚上の問題は、シーン反射率、照明、センサ雑音が結合した非理想的な画像形成に起因する。実用的な画像モデルは誤差源を定量化するのに役立つ：

$$I(u) = \rho(u) S(u) + n(u), \quad (166)$$

ここで $I(u)$ は画素 u における観測放射照度、 $\rho(u)$ は表面アルベド、 $S(u)$ は照明項（鏡面成分と指向性成分を含む）、 $n(u)$ はセンサ雑音である。堅牢なエンジニアリングには、与えられた運用エンベロープでどの項が支配的かを推定し、それに応じた緩和策を設計することが求められる。

主要な知覚上の課題：

- ・照明変動と鏡面反射。屋内照明と屋外の太陽光は局所的な飽和と非ランバートハイライトを引き起こす。これらの効果はオプティカルフローやテンプレートマッチングで用いられる輝度一定性仮定を破壊する。
- ・モーションブラーとローリングシャッターアーティファクト。ヒューマノイドはしばしば頭部にカメラを搭載し、頭部の高速動作はブラーを引き起こし、特徴検出器と動作からの深度の両方を破損させる。
- ・オクルージョンとクラッター。人間の環境では物体や手の部分的なオクルージョンが頻繁に発生する。部分的観測可能性は検出、追跡、把持計画を複雑化する。
- ・スケールと視点変化。操縦タスクでは広いスケール範囲で物体を認識する必要がある。分類器は遠方と近接の視点にわたって一般化しなければならない。
- ・無模様または繰り返し表面。ステレオや特徴ベース手法は低テクスチャ領域で失敗し、深度ホールと曖昧なデータ対応付けを引き起こす。
- ・敵対的または予期しない視覚入力。反射、透明材料、意図的な敵対的パターンは分類器を誤導することがある。

計算およびシステムレベルの課題：

- ・リアルタイム制約。ヒューマノイドのバランスと操縦は制御ループのデッドライン以下の知覚遅延を要求する。知覚パイプラインはタイミング制約 $\tau_{\text{percept}} + \tau_{\text{comm}} < \tau_{\text{control}}$ を満たさなければならない。
- ・電力および熱制限。高性能 GPU は複雑なモデルを可能にするが、モバイルヒューマノイドでの電力スロットリングは持続的スループットを制約する。
- ・メモリおよび帯域幅。高解像度カメラは大きなデータレートを生成する。オンボードバスおよびインターコネクトはマルチカメラリグのピークスループットを制限する。
- ・Sim-to-real のミスマッチ。シミュレーションで学習したモデルはハードウェアへの展開時に分布シフトを示す。ドメインランダムマイゼーションとフォトリアリズムはギャップを減らす但除去は

できない。

統合および融合上の課題：

- ・センサキャリブレーションのドリフト。カメラと IMU 間の相対パラメータは温度および機械的ストレスにより変化する。キャリブレーションを固定すると再投影誤差が蓄積する。
- ・時系列データ対応付け。追跡はオクルージョンや視点変化にもかかわらずフレーム間で検出を対応付けることを要求する。誤った対応付けは安全でない動作計画につながる。
- ・クロスモーダル競合。ライダーや深度カメラは単眼意味の手がかりと矛盾することがある。決定ロジックは信頼性を定量化し一貫した推定値を選択しなければならない。

簡潔なエンジニアリング分析は、これらの要因が推定器誤差にどう影響するかを示す。カメラ内部パラメータ K と姿勢 (R, t) を用いた再投影ベース推定器について、3D 点 X の再投影残差は

$$e = \|u - \pi(K [R | t] X)\|_2, \quad (167)$$

ここで $\pi(\cdot)$ は透視除算である。 u の観測雑音と (R, t) の不確実性は 3D 位置誤差に非線形に伝搬する。共分散を明示的にモデル化することで、状態推定器は低信頼度測定値を重み付けを下げることができる。

実用的な緩和策とアーキテクチャ選択：

1. 頑健なフロントエンド前処理

- ・動的露出制御とローカルトーンマッピングは飽和を軽減する。
- ・時系列デブラーとローリングシャッター補正は IMU タイムスタンプを用いてフレームを安定化する。
- ・信頼度マスクは下流での使用のために低品質画素をフラグ付けする。

2. 異種センサ融合

- ・RGB、深度、IMU を明示的雑音モデルで融合する。
- ・深度信頼度とステレオ視差妥当性チェックを用いて外れ値を拒否する。

3. 計算を意識した知覚スタック

- ・マルチ解像度パイプラインを実装：粗い意味的セグメンテーションの後、操縦のための高解像度 ROI 処理。
- ・リアルタイムタスク（障害物検出、足場配置）を低遅延スレッドで優先する。

4. ドメイン頑健な学習

- ・学習データに光度補強、合成鏡面反射、センサ雑音を用いる。
- ・シミュレートカメラ雑音をハードウェアと経験的に一致するようにキャリブレーションする。

実装例：カラーと深度を読み取り、露出補償カラー画像を計算し、深度信頼度マスクを公開する非同期ビジョンワーカー。コードは頑健性のための最小限の実用的ステップを示す。

コードサンプル 71 露出を意識した処理のための非同期ビジョンワーカー

```
import cv2
import numpy as np
```

```

import threading
import time
from typing import Callable, Tuple, Optional

Frame = np.ndarray
Depth = np.ndarray
Confidence = np.ndarray
Timestamp = float
PublishFn = Callable[[Frame, Depth, Confidence, Timestamp], None]

class VisionWorker(threading.Thread):
    def __init__(self,
                 frame_source,
                 publish_fn: PublishFn,
                 fps: float = 30.0,
                 timeout: float = 1.0):
        super().__init__(daemon=True)
        self._src = frame_source
        self._publish = publish_fn
        self._running = threading.Event()
        self._running.set()
        self._period = 1.0 / fps
        self._timeout = timeout

    def run(self) -> None:
        while self._running.is_set():
            t0 = time.monotonic()
            color, depth, ts = self._src.read() # ブロッキング読み出し
            if color is None or depth is None:
                continue

            # ヒストグラム均一化で露出補正
            lab = cv2.cvtColor(color, cv2.COLOR_BGR2LAB)
            l, a, b = cv2.split(lab)
            l = cv2.equalizeHist(l)
            color_out = cv2.cvtColor(cv2.merge((l, a, b)), cv2.COLOR_LAB2BGR)

            # 深度信頼度マスク生成
            valid = (depth > 0.1) & (depth < 5.0)
            median = cv2.medianBlur(depth, 5)

```

```

conf = np.abs(depth - median) < 0.05
depth_conf = valid & conf

self._publish(color_out, depth, depth_conf, ts)

# FPS制御
elapsed = time.monotonic() - t0
sleep = max(0.0, self._period - elapsed)
time.sleep(sleep)

def stop(self) -> None:
    self._running.clear()
    self.join(timeout=self._timeout)

```

設計上のトレードオフと運用上のリスク：

- モデル複雑性を増やすと精度は向上するが遅延と消費電力が増加する。最悪ケースの制御デッドラインを満たす最小モデルを選択する。
- 不確実な測定値を積極的に拒否すると誤検出は減るが見逃しが増え、追跡の持続性を損なう。
- キャリブレーションされた雑音モデルなしにシミュレーションへの過度の依存は現実世界で脆い挙動を引き起こす。
- 失敗モードには、遅延した知覚による不安定化、誤対応付けによる危険な操縦、未処理の鏡面反射による壊滅的な深度誤差が含まれる。

エンジニアは要求仕様の取り込み中にこれらのトレードオフを定量化しなければならない。遅延予算、故障に耐性のあるフォールバック挙動、安全で信頼できるヒューマノイド運用を維持するための定期的な再キャリブレーションを指定する。

20 ビジョンとロボティクスの統合

20.1 ビジョンセンサとハードウェア

検出、分類、深度推定といったコアビジョントaskを確立したうえで、ここでは人型ロボットでそれらを実現する物理センサとハードウェアの選択について考察する。以下の議論では、センシング能力を実装上の設計制約やオンボード知覚パイプラインの統合要件と結びつける。

ビジョン・ハードウェアの選択は、運用上の要求を満たす必要がある：閉ループバランスのための遅延、人間との相互作用のための視野 (FOV)、操縦のための深度精度、移動型ヒューマノイドにとっての電力・重量予算。代表的なセンサクラスとそのエンジニアリング上のトレードオフは以下の通り：

- 単眼 RGB：コンパクト、低消費電力、高解像度。セマンティック知覚や顔・ジェスチャ認識に使用。制限：計測深度は Structure-from-Motion または外部深度センサが必要となり、計算負荷が

増大。

- ステレオ RGB：2 台の校正済みカメラによる受動的深度。利点：単純な光学系、能動光源を使わない設計では太陽光に強い。重要パラメータ：ベースライン B と焦点距離 f が深度感度を決定。視差 d から深度 z は

$$[H]z = \frac{f B}{d}. \quad (168)$$

深度精度は視差分解能に依存；ベースラインを大きくすると長距離精度が向上するが、機械的統合の複雑さが増す。

- RGB-D（構造光、TOF）：高密度深度を直接取得。構造光は屋内に優れ；ToF は照明に強いがマルチパス・環境光ノイズに弱い。環境と要求深度レンジに応じて選択。
- イベントカメラ：非同期・高時間分解能センサで高速移動・滑り検出に最適。低遅延・高ダイナミックレンジを提供するが、専用アルゴリズムが必要でセマンティックタスクには直感的でない。
- LiDAR：走査 3D 点群で長距離・高精度。通常はマッピング・グローバル自己位置推定に使用。高コスト、重量、胴体・脚部搭載では機械的または電子的に複雑。
- サーマル・多スペクトルカメラ：雑踏や悪条件下での人間検出、災害救助ヒューマノイド用途に有用。

センサ選択には、知覚性能に直接影響するハードウェアレベルの特性を考慮する：

- シャッター方式：グローバルシャッターは頭部高速動作時のモーション歪を回避。ローリングシャッターは高速動作で深刻なアーティファクトを引き起こす。
- ダイナミックレンジ・露光制御：高ダイナミックレンジは屋内・屋外を往復する際のクリッピングを軽減。
- フレームレート・遅延：閉ループバランス・反射動作には 50 ms 以下のエンドツーエンド遅延が必要。パイプラインバッファリングを最小化し、ハードウェアタイムスタンプを優先。
- 解像度 vs 帯域：高解像度は計算・バス負荷を増大。GPU/CPU リソースやネットワーク帯域が限られる場合は ROI または圧縮エンコードを使用。
- インターフェース・ドライバ成熟度：量産グレードドライバ、標準プロトコル（USB3 Vision、GigE Vision、MIPI CSI-2）および ROS/ROS2 ドライバが用意されたセンサを優先し、迅速な統合を実現。
- 物理的制約：重量、重心影響、放熱、配線経路、EMI シールド、機械的堅牢性。

搭載と外部パラメータ：センサ配置と剛体変換により、画像座標からロボット座標へのマッピングが決まる。内部・外部パラメータは既存ツールで校正。ステレオやマルチセンサリグでは動作中に外部パラメータが安定するよう機械的剛性を確保。代表的な配置：

1. 頭部：ナビゲーション・インタラクション用広角ステレオまたは RGB-D。
2. 胸部・胴体：マッピング用 LiDAR または長ベースラインステレオ。
3. 手部：近接操作小型 RGB または構造光深度。

センサと IMU 間の同期・タイミングは重要。ハードウェアトリガ、Precision Time Protocol (PTP)、または GPS 同期クロックを分散システムに使用。ソフトウェアのみのタイムスタンプはばらつきを生じ、センサ融合を劣化させる。

センサ融合・処理アーキテクチャ：遅延・スループット目標を満たすようタスクを適切なハードウェアに割り当てる。

- 画素処理（デノイズ、補正）は利用可能な ISP または FPGA にオフロード。
- ニューラルネットワーク推論は GPU（NVIDIA Jetson/Isaac）に、リアルタイム制御ループは CPU に割り当てる。
- 深度ストリームに対して初期フィルタ（時間中央値、バイラテラル）を適用し、下流推定器のノイズを削減。

ベースライン選択・深度分解能の設計方程式はトレードオフを定量化する。ステレオでは視差量子化 Δd (pixel) を定義。深度不確かさ Δz は概ね

$$[H]\Delta z \approx \frac{fB}{d^2} \Delta d. \quad (169)$$

目標距離 z_t と許容深度誤差 Δz_{\max} に対し、ベースライン B を

$$[H]B \geq \frac{z_t^2 \Delta d}{f \Delta z_{\max}}. \quad (170)$$

と選ぶ。これにより、距離増加に伴う深度精度への 2 次のペナルティを明確にする。

実装例：OpenCV ステレオブロックマッチングで視差・深度を計算。スニペットは頭部搭載ステレオペアで補正済み視差を計測深度に変換する方法を示す。コードは較正済み内部パラメータとステレオ補正行列を仮定。

コードサンプル 72 Compute depth from stereo disparity using OpenCV.

```
import cv2
import numpy as np
from pathlib import Path
import logging

# ログ設定
logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')

# パラメータ定義
FOCAL_LENGTH_PX = 700.0          # 焦点距離 [px]
BASELINE_M = 0.12                 # ベースライン長 [m]
MIN_DISPARIITY = 0.1              # 無効視差閾値
STEREO_PARAMS = {
    'numDisparities': 128,
    'blockSize': 15,
    'preFilterType': cv2.STEREO_BM_PREFILTER_NORMALIZED_RESPONSE,
    'preFilterSize': 9,
    'preFilterCap': 31,
    'textureThreshold': 10,
```

```

        'uniquenessRatio': 15,
        'speckleWindowSize': 100,
        'speckleRange': 32,
        'disp12MaxDiff': 1
    }

def load_rectified_pair(left_path: str, right_path: str) -> tuple[np.ndarray, np.ndarray]:
    """左右整流済画像を読み込む"""
    left = cv2.imread(left_path, cv2.IMREAD_GRAYSCALE)
    right = cv2.imread(right_path, cv2.IMREAD_GRAYSCALE)
    if left is None or right is None:
        raise FileNotFoundError('画像読み込み失敗')
    return left, right

def compute_depth(left: np.ndarray, right: np.ndarray) -> np.ndarray:
    """視差→奥行へ変換"""
    stereo = cv2.StereoBM_create(**STEREO_PARAMS)
    disp = stereo.compute(left, right).astype(np.float32) / 16.0
    valid = disp > MIN_DISPARITY
    depth = np.zeros_like(disp)
    depth[valid] = (FOCAL_LENGTH_PX * BASELINE_M) / disp[valid]
    return depth

def main():
    left_img, right_img = load_rectified_pair('left_rect.png', 'right_rect.png')
    depth_map = compute_depth(left_img, right_img)
    cv2.imwrite('depth_meters.exr', depth_map)
    logging.info('depth_meters.exr 保存完了')

if __name__ == '__main__':
    main()

```

エンジニアリング上の影響、トレードオフ、運用上のリスク：

- 機械的剛性とコネクタ信頼性は外部パラメータの安定性、ひいてはマッピング精度に直接影響。
- 大きなベースラインを選ぶと長距離深度が向上するが、頭部慣性が増え動的バランスに影響。
- イベントカメラは遅延を削減するが専用アルゴリズムが必要；成熟度不足はシステム統合リスクを高める。
- 環境光・反射面は ToF・構造光深度に誤差を生じる；センサ冗長性を計画。
- 同期失敗はセンサ融合の不整合を引き起こし、バランス臨界動作で致命的な制御エラーに至る。

設計者はセンサ精度、計算負荷、電力、物理的制約をバランスさせる必要がある。ベースライン、

焦点距離、遅延目標をシステム設計の初期段階で明示的に定量化し、後期統合失敗を回避する。

20.2 ビジョンパイプラインの構築

前のサブセクションでは、カメラの選択、深度モダリティ、センサ配置の制約を確立し、それらがパイプラインの複雑さとタイミングに直接影響することを示した。ここでは、これらのハードウェアの決定を人間型ロボット向けの具体的で展開可能なビジョンパイプラインに変換し、キャリブレーション、レイテンシ予算、モジュラーなソフトウェア設計を重視する。

問題定義と運用目標：信頼性の高い 6-DoF ポーズ手がかりと意味的検出を、制限付きレイテンシでモーションプランナーに届けるビジョンパイプラインを構築する。パイプラインは可変照明、動的全身運動によるモーションブラー、四肢による断続的な遮蔽に耐えなければならない。主要な性能指標は以下の通り：

- ・エンドツーエンドレイテンシ（センサキャプチャからメッセージ公開まで） $\leq 50\text{ ms}$ で反応的歩行調整を実現；
- ・タスク許容範囲内の検出精度とポーズ誤差、例えば操作では位置 3 cm 、姿勢 5° ；
- ・オンボード NVIDIA ハードウェアと互換性のある CPU/GPU リソース予算。

技術的分解。実用的なパイプラインは明確な段階に分かれる：

1. 同期とキャプチャ：
 - ・ハードウェアタイムスタンプと Precision Time Protocol (PTP) でカメラと IMU を整列；
 - ・高速胴体運動に対して露光制御とローリングシャッター緩和を使用。
2. キャリブレーション：
 - ・各カメラの内部パラメータとカメラ・ロボットベース間の外部パラメータ；
 - ・視差から深度を得るためのステレオ整列、またはアクティブセンサ用の深度カメラ内部パラメータ。
3. 前処理：
 - ・デノイズ、デバイヤ、作業色空間への変換；
 - ・計算とネットワーク帯域削減のための画像 ROI 選択。
4. 知覚モジュール：
 - ・検出/分類 (TensorRT による CNN 推論加速)；
 - ・ステレオと ToF センサ間の深度推定または深度融合；
 - ・ポーズ推定 (PnP、ICP、または学習ベース 6-DoF 推定器)。
5. 融合とトラッキング：
 - ・カルマンまたはパーティクルフィルタで IMU と運動学的事前情報と検出を融合；
 - ・時間的関連付けで遮蔽下でも物体 ID を維持。
6. プランナーへのインタフェース：
 - ・共分散考慮ポーズ推定を ROS2 メッセージで公開；
 - ・レイテンシと信頼性の診断チャンネルを提供。

キャリブレーションと再投影誤差。カメラ内部パラメータ K とロボットベースに対する外部パラメータ (R, t) について、3D ランドマーク X の再投影は観測画像点 u と一致すべきである。再投影残

差を最小化：

$$[H]e(u, X) = \|u - \Pi(K[R \mid t]X)\|_2, \quad (171)$$

ここで $\Pi(\cdot)$ は同次正規化を行う。キャリブレーション手順はすべての観測にわたり $e(u, X)^2$ の和を最小化する。機械的ドリフトや温度変化を検出するため、常時キャリブレーション残差モニタを維持する。

ステレオ深度とベースラインのトレードオフ。焦点距離 f 、ベースライン b のステレオペアでは、視差 d からの深度 Z は：

$$[H]Z = \frac{fb}{d}. \quad (172)$$

設計上の含意：

- 小さい b は長距離での三角測定感度を低下；
- 大きい b は機械的クリアランス要求とキャリブレーション感度を増加；
- 短距離精度とボディ制約のトレードオフで b を選択。

レイテンシ予算とスケジューリング。段階に時間を割り当ててレイテンシ予算を構築。50 ms 総量の例：

- キャプチャ＋転送：8 ms；
- 前処理＋整列：6 ms；
- 推論（GPU）：20 ms；
- 融合＋公開：8 ms；
- マージン：8 ms。

ROS2 で適切な QoS プロファイルによりソフトリアルタイムスケジューリングを実施。過負荷時に決定的ドロップを可能にするため、センサフレーム用リングバッファを使用。

実装パターン：明確なトピック契約を持つモジュラー ROS2 ノード。効率的な圧縮ストリーム用に image transport を使用。CNN 推論を Jetson または PCIe GPU 上の TensorRT エンジンにオフロード。簿記と共分散伝搬用に軽量 CPU スレッドを維持。

カメラを購読し TensorRT 検出器を実行し、ポーズ風メッセージを公開する ROS2 Python ノードの例。以下のスニペットは主要な統合ポイントを示すが、簡潔のためモデル構築詳細は省略。

コードサンプル 73 ROS2 subscriber node running GPU-accelerated detector and publishing pose

```
import os
import time
from typing import List, Tuple

import cv2
import numpy as np
import rclpy
from cv_bridge import CvBridge
from geometry_msgs.msg import PoseStamped
```

```

from rclpy.node import Node
from sensor_msgs.msg import Image
import tensorrt as trt
import pycuda.driver as cuda
import pycuda.autoinit

TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
ENGINE_PATH = '/models/det.trt'
INPUT_SHAPE = (3, 480, 640) # C, H, W
OBJ_POINTS = np.load('/models/object_points.npy') # 3D model points
CAM_MATRIX = np.load('/models/camera_matrix.npy')
DIST_COEFFS = np.load('/models/dist_coeffs.npy')

def load_trt_engine(engine_path: str) -> trt.ICudaEngine:
    """TensorRT_エンジンを読み込み返す。"""
    if not os.path.isfile(engine_path):
        raise RuntimeError(f'{engine_path}_が存在しません')
    with open(engine_path, 'rb') as f, trt.Runtime(TRT_LOGGER) as runtime:
        return runtime.deserialize_cuda_engine(f.read())

class TrtWrapper:
    """TensorRT_推論ラッパー：メモリ確保～推論までをカプセル化。"""
    def __init__(self, engine: trt.ICudaEngine):
        self.engine = engine
        self.context = engine.create_execution_context()
        # 入出力バッファ確保
        self.inputs, self.outputs, self.bindings, self.stream = self._alloc_buf()

    def _alloc_buf(self):
        inputs, outputs, bindings = [], [], []
        stream = cuda.Stream()
        for binding in self.engine:
            size = trt.volume(self.engine.get_binding_shape(binding))
            dtype = trt.nptype(self.engine.get_binding_dtype(binding))
            host_mem = cuda.pagelocked_empty(size, dtype)
            device_mem = cuda.mem_alloc(host_mem.nbytes)
            bindings.append(int(device_mem))

```

```

        if self.engine.binding_is_input(binding):
            inputs.append({'host': host_mem, 'device': device_mem})
        else:
            outputs.append({'host': host_mem, 'device': device_mem})
    return inputs, outputs, bindings, stream

def infer(self, img: np.ndarray) -> np.ndarray:
    """前処理済画像を入力し生の推論結果を返す。"""
    np.copyto(self.inputs[0]['host'], img.ravel())
    cuda.memcpy_htod_async(
        self.inputs[0]['device'], self.inputs[0]['host'], self.stream)
    self.context.execute_async_v2(
        bindings=self.bindings, stream_handle=self.stream.handle)
    cuda.memcpy_dtoh_async(
        self.outputs[0]['host'], self.outputs[0]['device'], self.stream)
    self.stream.synchronize()
    return self.outputs[0]['host']

def postprocess(raw: np.ndarray, conf_th: float = 0.5) -> List[Tuple[int, int, int, int, float, str]]:
    """生推論結果をフィルタし[(x1,y1,x2,y2),...]を返す。"""
    # 簡易例：出力が [N,6] (x1,y1,x2,y2,conf,class) と仮定
    raw = raw.reshape(-1, 6)
    keep = raw[:, 4] > conf_th
    return [tuple(map(int, box[:4])) for box in raw[keep]]

def estimate_pose_2d_to_3d(box: Tuple[int, int, int, int]) -> Tuple[np.ndarray, np.ndarray]:
    """2Dバウンディングボックスから姿勢推定し(rvec,tvec)を返す。"""
    # 簡易：矩形中心を画像座標とし対応する3D点をOBJ_POINTSから選択
    x1, y1, x2, y2 = box
    pts2d = np.array([[x1, y1], [x2, y1], [x2, y2], [x1, y2]], dtype=np.float32)
    success, rvec, tvec = cv2.solvePnP(
        OBJ_POINTS, pts2d, CAM_MATRIX, DIST_COEFFS, flags=cv2.SOLVEPNP_IPPE)
    if not success:
        raise RuntimeError('PnP失敗')
    return rvec, tvec

def to_pose_msg(rvec: np.ndarray, tvec: np.ndarray, stamp, frame_id: str) -> PoseStamped:

```

```

""" rvec/tvec を PoseStamped に変換。 """
pose = PoseStamped()
pose.header.stamp = stamp
pose.header.frame_id = frame_id
rot_mat, _ = cv2.Rodrigues(rvec)
qw, qx, qy, qz = cv2.RQDecomp3x3(rot_mat)[1] # 簡易クォータニオン化
pose.pose.orientation.w = qw
pose.pose.orientation.x = qx
pose.pose.orientation.y = qy
pose.pose.orientation.z = qz
pose.pose.position.x, pose.pose.position.y, pose.pose.position.z = tvec.flatten()
return pose

```

```

class VisionNode(Node):
    def __init__(self):
        super().__init__('vision_node')
        self.bridge = CvBridge()
        self.trt = TrtWrapper(load_trt_engine(ENGINE_PATH))

        self.sub = self.create_subscription(
            Image, 'camera/image_raw', self.image_cb, 10)
        self.pub = self.create_publisher(PoseStamped, 'vision/pose', 10)

        self.get_logger().info('VisionNode_初期化完了')

    def image_cb(self, msg: Image):
        try:
            cv_img = self.bridge.imgmsg_to_cv2(msg, 'bgr8')
        except Exception as e:
            self.get_logger().error(f'cv_bridge_変換失敗: {e}')
            return

        # 前処理
        blob = cv2.dnn.blobFromImage(
            cv_img, scalefactor=1/255.0, size=(INPUT_SHAPE[2], INPUT_SHAPE[1]),
            swapRB=True, crop=False)
        raw = self.trt.infer(blob)
        dets = postprocess(raw)
        if not dets:

```

```

        return

    # 先頭検出のみ利用
    rvec, tvec = estimate_pose_2d_to_3d(dets[0])
    pose_msg = to_pose_msg(rvec, tvec, msg.header.stamp, 'base_link')
    self.pub.publish(pose_msg)

def main(args=None):
    rclpy.init(args=args)
    node = VisionNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

運用上の考慮とリスク：

- ・機械振動はポーズ推定器を劣化；マウントダンピングが必須。
- ・温度ドリフトは外部パラメータを変化；定期的な再キャリブレーションまたはオンライン外部適応をスケジュール。
- ・単一モダリティへの過度依存は故障モードを引き起こす；可能な限りステレオ、ToF、IMU を融合。
- ・知覚と制御計画間の GPU 競合はレイテンシ予算に違反；ワークロードを分割または推論を優先。

設計トレードオフの概要：

- ・より高い精度はより広いベースライン、より多くの計算、より厳格なキャリブレーションを要求。
- ・より低いレイテンシはより単純なモデル、ROI クロップ、ハードウェア加速推論を好む。
- ・遮蔽に対する頑健性はトラッキングと時間的融合に利益があり、状態複雑性のコストが伴う。

エンジニアは統合中にパイプラインレイテンシ、キャリブレーション残差、故障率を定量化すべきである。これらの指標を PTP に整列したタイムスタンプでログし、回帰をセンサ、ソフトウェア、または機械的要因まで追跡する。

Perception and Vision Systems

21 Fundamentals of Computer Vision

21.1 物体検出と分類

これまでに紹介した視覚処理の基礎を踏まえ、本小節ではヒューマノイドロボット向けの実用的な物体検出と分類に焦点を当てる。センシング問題を定式化し、主要指標とアルゴリズムを導出し、リアルタイム推論の実装スケッチを示し、エンジニアリング上のトレードオフをまとめる。

問題定義：ヒューマノイドは、操縦、ナビゲーション、または対話を実行するために、雑然とした動的環境で物体を検出し分類しなければならない。検出は物体を位置付ける（バウンディングボックスまたはマスク）。分類は意味ラベルを割り当てる。ヒューマノイド特有の制約として、限られた計算資源、バランスおよび操縦のための厳しい遅延制約、頭部搭載カメラによる至近距離の視点歪みがある。

技術分析：

- タスク分解：
 1. ステレオまたは RGB-D センサからの画像取得および前処理。
 2. 候補生成（領域提案またはシングルショット予測器）。
 3. 位置回帰およびクラス予測。
 4. 後処理：非最大値抑制（NMS）、信頼度閾値処理、および深度関連付け。
- 位置精度は Intersection-over-Union（IoU）で定量化される。予測ボックス P と正解ボックス G に対する IoU を定義：

$$[H] \text{IoU}(P, G) = \frac{\text{area}(P \cap G)}{\text{area}(P \cup G)}. \quad (173)$$

IoU 閾値は評価時の真陽性受容および NMS に用いられる。

- 一段および二段検出器に共通する損失分解：

$$[H] L = L_{\text{cls}} + \lambda L_{\text{loc}}, \quad (174)$$

ここで L_{cls} はクラス不均衡に対するクロスエントロピーまたは focal 損失、 L_{loc} は Smooth- L_1 または IoU ベースの位置損失である。 λ を調整することで分類信頼度とボックス精度のバランスを取る。

- 検出パイプラインとトレードオフ：
 - 二段（Faster R-CNN）：高精度、遅い遅延。把持計画で精度が重要な場合に有用。
 - 一段（YOLO, SSD）：低遅延、ピーク精度は低い。閉ループ反応性を要するヒューマノイドに好まれる。
 - アンカーフリー手法はハイパーパラメータを減らすが、人間環境に多い極端なアスペクト比で苦労することがある。

実装上の考慮事項：

- ヒューマノイドでのリアルタイム推論は通常、オンボード NVIDIA GPU または Jetson モジュールで実行される。TensorRT への変換で遅延を削減する。

- 深度アライメントを用いて 2D 検出を 3D 物体重心に変換する。バウンディングボックス中心 (u, v) と深度 z に対し、カメラ座標系での点は

$$[H]\mathbf{p} = zK^{-1}[u \ v \ 1]^T, \quad (175)$$

カメラ内部パラメータ行列 K を用いる。

- 時空間フィルタ（カルマンまたは単純指数平滑化）は操縦のための検出を安定化する。IMU または運動学的事前知識を融合し、動作中の誤検出を除去する。

実用的アルゴリズムパイプライン：

- 前処理：色正規化，アスペクト保持リサイズ，照明ばらつきのための拡張。
- 推論：検出器を実行；score_threshold を適用し低信頼度出力を除去。
- 後処理：IoU 閾値 nms_iou で NMS を実行し重複を除去。
- 深度関連付け：ボックス内の中央深度をサンプリングし 3D 姿勢を計算；信頼できる深度がないボックスを棄却。

ヒューマノイドベンチマークで監視すべき評価指標：

- クラスごとの適合率，再現率，および平均適合率（AP）。
- ロボット動作下での推論時間の遅延と分散。
- 誤検出の空間分布（安全性に重要）。

オンボード GPU 向けに最適化された推論ループを示すコードリスト。このスニペットは TensorRT 最適化検出器と深度画像ストリームを前提とする。

コードサンプル 74 組み込み GPU でのリアルタイム検出推論（TensorRT）。

```
import time
import cv2
import numpy as np
import torch
import torchvision
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from vision_msgs.msg import Detection3D, Detection3DArray, ObjectHypothesisWithPose
from cv_bridge import CvBridge
from geometry_msgs.msg import Point
import threading
import queue

class TrtDetector(Node):
    def __init__(self, engine_path, K, score_th=0.3, nms_iou=0.45, target_size=(640, 3
```

```

        self.engine = self._load_engine(engine_path)
# TensorRTエンジン読み込み
        self.K = np.array(K)
        self.score_th = score_th
        self.nms_iou = nms_iou
        self.target_size = target_size
        self.bridge = CvBridge()
        self.q_rgb = queue.Queue(maxsize=2)
        self.q_depth = queue.Queue(maxsize=2)
        self.pub = self.create_publisher(Detection3DArray, '/detections_3d', 10)
        self.create_subscription(Image, '/camera/color/image_raw', self._cb_rgb, 1)
        self.create_subscription(Image, '/camera/aligned_depth_to_color/image_raw', self._cb_depth, 1)
        threading.Thread(target=self._infer_loop, daemon=True).start()

def _load_engine(self, path):
    import tensorrt as trt
    with open(path, 'rb') as f, trt.Runtime(trt.Logger()) as runtime:
        return runtime.deserialize_cuda_engine(f.read())

def _cb_rgb(self, msg):
    if self.q_rgb.full():
        self.q_rgb.get_nowait()
    self.q_rgb.put(self.bridge.imgmsg_to_cv2(msg, 'bgr8'))

def _cb_depth(self, msg):
    if self.q_depth.full():
        self.q_depth.get_nowait()
    self.q_depth.put(self.bridge.imgmsg_to_cv2(msg, '16UC1'))

def _preprocess(self, rgb):
    img = cv2.resize(rgb, self.target_size)
    img = img.astype(np.float32) / 255.0
    return img.transpose(2, 0, 1)[None, ...]

def _postprocess(self, boxes, scores, classes):
    keep = scores > self.score_th
    boxes, scores, classes = boxes[keep], scores[keep], classes[keep]
    idxs = torchvision.ops.nms(torch.tensor(boxes), torch.tensor(scores), self.nms_iou)
    return boxes[idxs].numpy(), scores[idxs].numpy(), classes[idxs].numpy()

```

```

def _infer_loop(self):
    while rclpy.ok():
        try:
            rgb = self.q_rgb.get(timeout=1)
            depth = self.q_depth.get(timeout=1)
        except queue.Empty:
            continue
        t0 = time.perf_counter()
        x = self._preprocess(rgb)
        boxes, scores, classes = self._infer_trt(x)
        boxes, scores, classes = self._postprocess(boxes, scores, classes)
        msg = Detection3DArray()
        msg.header.stamp = self.get_clock().now().to_msg()
        for b, s, c in zip(boxes, scores, classes):
            u0, v0, u1, v1 = map(int, b)
            z = np.median(depth[v0:v1, u0:u1]) * 0.001 # mm→m
            if np.isfinite(z) and z > 0.2:
                xyz = z * np.linalg.inv(self.K) @ np.array([(u0 + u1) / 2, (v0 + v1) / 2, 1])
                det = Detection3D()
                det.bbox.center.position = Point(x=float(xyz[0]), y=float(xyz[1]), z=float(xyz[2]))
                hyp = ObjectHypothesisWithPose()
                hyp.hypothesis.class_id = str(int(c))
                hyp.hypothesis.score = float(s)
                det.results.append(hyp)
                msg.detections.append(det)
        self.pub.publish(msg)
        self.get_logger().debug(f'latency: {(time.perf_counter()-t0)*1000:.1f}')

def _infer_trt(self, x):
    import pycuda.autoinit
    import pycuda.driver as cuda
    with self.engine.create_execution_context() as ctx:
        d_in = cuda.mem_alloc(x.nbytes)
        cuda.memcpy_htod(d_in, x)
        out_shapes = [(100, 4), (100,), (100,)]
        outs = [np.empty(s, dtype=np.float32) for s in out_shapes]
        d_outs = [cuda.mem_alloc(o.nbytes) for o in outs]
        ctx.execute_v2([int(d_in)] + [int(d) for d in d_outs])
        for o, d in zip(outs, d_outs):
            cuda.memcpy_dtoh(o, d)

```

```

        return outs

def main():
    rclpy.init()
    K = [[615.0, 0, 320], [0, 615.0, 180], [0, 0, 1]]
    node = TrtDetector('model.trt', K)
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計上のトレードオフと運用上のリスク：

- ・遅延対精度：低遅延モデルは制御ラグを減らす、保持中の誤分類リスクを増やす。
- ・信頼度校正：過信モデルは安全でない動作を引き起こす；不信モデルはタスクスループットを低下させる。
- ・深度信頼性：反射または透明表面は深度を劣化させ、誤った 3D 位置付けを引き起こす。
- ・ヒューマノイドの電力および熱予算は連続高レート推論を制約し、動作状態に応じた可変レートで知覚をスケジュールする。

エンジニアは `score_threshold` および `nms_iou` をタスクごとに調整し、エンドツーエンド遅延をプロファイリングし、一時的な誤検出を軽減するための時空間融合を展開すべきである。堅牢性には、実世界の人間環境、合成拡張、および安全なフォールバック動作のための明示的故障モードを組み合わせた評価が必要である。

21.2 奥行き知覚とステレオビジョン

物体検出・分類を基盤とし、奥行き知覚は人型ロボットに対して確実な把持、衝突回避、バランス制御を可能にする空間的な文脈を供給する。以下の文章は、頭部搭載センサリグ、低遅延制御ループ、慣性センシングとの統合に関連するステレオビジョンの詳細に焦点を当てる。

問題定義と運用要件：

- ・人型ロボットは典型的なインタラクション範囲 (0.3–5 m) にわたる高密度かつメトリックな奥行き推定を必要とする。
- ・制約としては、頭部のベースライン制限、組み込み GPU の計算予算、移動する人を含む動的シーン、可変照明が含まれる。
- ・安全上重要な動作（例：歩行、ハンドオーバー）は、既知の奥行き不確実性とリアルタイム信頼度指標を要求する。

技術分析 — 幾何学的基礎と誤差伝播：ステレオ奥行きは、2 台の校正済みピンホールカメラ間の三角測量から生じる。整直後、対応する水平画素座標 x_L と x_R は視差 $d = x_L - x_R$ を生じる。奥行き

z は

$$z = \frac{f B}{d}, \quad (176)$$

に従い、ここで f はピクセル単位の焦点距離、 B はメートル単位のベースラインである。視差検出誤差 σ_d は奥行き不確実性へ二次的に伝播する：

$$\sigma_z \approx \frac{f B}{d^2} \sigma_d = \frac{z^2}{f B} \sigma_d. \quad (177)$$

設計上の含意：小さなベースラインは遠方で σ_z を劇的に増大させ、より大きな焦点距離は視野を狭める代償で奥行き分解能を改善する。

人型頭部のためのパイプラインとアルゴリズム：

1. 校正と整直

- 各カメラの内部パラメータとカメラ間の外部パラメータを推定する。チェッカーボードまたは AprilTag をバンドル調整と共に使用。camera_matrix および distortion_coeffs を保存。
- 対極線整列を強制するため整直マップを計算。これは対応探索を 1 次元に削減する。

2. 対応付けマッチング

- 古典的な選択：ブロックマッチングおよびセミグローバルマッチング (SGM)。SGM は精度と計算をバランス取り；OpenCV の StereoSGBM はリアルタイムパイプラインで一般的。
- 学習ベース手法 (PSMNet、GANet) は低テクスチャおよび反射領域で改善された結果を提供するが、GPU と注意深いドメイン適応を要求する。

3. 後処理

- 左右一貫性チェックによりオクルージョンをマスク。
- マッチングコスト、一意性比率、正規化相互相関を用いた信頼度推定。
- 制御ループ向けに奥行きを安定化するための時空間フィルタリング（例：指数平滑化またはカルマンフィルタリング）。

4. 融合と上位レベル利用

- 高速頭部動作中の対応を解決するため、ステレオ奥行きを IMU 駆動運動事前情報と融合。
- 無テクスチャまたは鏡面表面を扱うため、ステレオを能動的奥行きセンサと統合。

実装上の注意：

- ベースライン選択：6–12 cm は人間尺度インタラクションに適し；長距離タスクでは機械的マウントおよび美的制約を保ちながらベースラインを増加。
- 露光と同期：グローバルシャッターカメラおよびハードウェア同期は運動誘発視差誤差を劇的に削減。
- 校正維持：熱ドリフトおよび機械的たわみは定期的な再校正または運動学チェーン内でのオンライン自己校正ルーチンを要求。

実装例（組み込み GPU 上の OpenCV；最小パイプライン）：

コードサンプル 75 OpenCV StereoSGBM を用いた基本的ステレオ奥行きパイプライン

```
import cv2
import numpy as np
```

```

from pathlib import Path
from typing import Tuple, Optional

class StereoDepthEstimator:
    """
    ステレオカメラからの深度推定を行うクラス
    """

    def __init__(self, calibration_path: str,
                  num_disparities: int = 128,
                  block_size: int = 5) -> None:
        """
        キャリブレーションデータの読み込みとSGBMマッチャーの初期化

        Args:
            calibration_path: ステレオキャリブレーションデータのパス
            num_disparities: 視差数 (16の倍数)
            block_size: ブロックサイズ (奇数)
        """
        # キャリブレーションデータの読み込み
        self._load_calibration_data(calibration_path)

        # SGBMマッチャーの初期化
        self._init_matcher(num_disparities, block_size)

    def _load_calibration_data(self, calibration_path: str) -> None:
        """キャリブレーションデータの読み込み"""
        try:
            maps = np.load(calibration_path)
            self.left_map1 = maps['left_map1']
            self.left_map2 = maps['left_map2']
            self.right_map1 = maps['right_map1']
            self.right_map2 = maps['right_map2']
            self.Q = maps['Q']
        except (FileNotFoundError, KeyError) as e:
            raise ValueError(f"キャリブレーションデータの読み込みに失敗しました:{e}")

    def _init_matcher(self, num_disparities: int, block_size: int) -> None:
        """SGBMマッチャーの初期化"""
        if num_disparities % 16 != 0:

```

```

        raise ValueError("num_disparitiesは16の倍数である必要があります")
    if block_size % 2 == 0:
        raise ValueError("block_sizeは奇数である必要があります")

    self.matcher = cv2.StereoSGBM_create(
        minDisparity=0,
        numDisparities=num_disparities,
        blockSize=block_size,
        P1=8 * 3 * block_size ** 2,
        P2=32 * 3 * block_size ** 2,
        mode=cv2.STEREO_SGBM_MODE_HH
    )

    def compute_depth(self, left_img: np.ndarray, right_img: np.ndarray,
                      confidence_threshold: float = 0.6) -> Tuple[np.ndarray, np.ndarray,
        """
        """
        """ステレオ画像から深度マップを計算"""

        """Args:
        """
        """left_img: 左カメラ画像 (グレースケール)"""
        """right_img: 右カメラ画像 (グレースケール)"""
        """confidence_threshold: 信頼度閾値"""

        """Returns:
        """
        """depth: 深度マップ"""
        """disparity: 視差マップ"""
        """confidence_mask: 信頼度マスク"""
        """
        """
        """# 入力検証"""
        if left_img.shape != right_img.shape:
            raise ValueError("左右画像のサイズが一致しません")
        if len(left_img.shape) != 2 or len(right_img.shape) != 2:
            raise ValueError("入力画像はグレースケールである必要があります")

        """# 画像の補正"""
        left_rect = cv2.remap(left_img, self.left_map1, self.left_map2, cv2.INTER_LINEAR)
        right_rect = cv2.remap(right_img, self.right_map1, self.right_map2, cv2.INTER_LINEAR)

        """# 視差計算"""
        disparity = self.matcher.compute(left_rect, right_rect).astype(np.float32) / 16

```

```

# 3D点群の再投影
points_3d = cv2.reprojectImageTo3D(disparity, self.Q)
depth = points_3d[..., 2]

# 信頼度マスクの計算（視差の連続性と有効範囲をチェック）
confidence_mask = self._compute_confidence_mask(disparity, depth, confidence_t

# 無効な深度値を除外
depth[~confidence_mask] = np.nan

return depth, disparity, confidence_mask

def _compute_confidence_mask(self, disparity: np.ndarray, depth: np.ndarray,
                             threshold: float) -> np.ndarray:
    """
    視差マップから信頼度マスクを計算

    Args:
        disparity: 視差マップ
        depth: 深度マップ
        threshold: 信頼度閾値

    Returns:
        信頼度マスク (bool配列)
    """
    # 視差の有効範囲チェック
    valid_disparity = (disparity > 0) & (disparity < self.matcher.getNumDisparities())

    # 深度の有効範囲チェック（負の値や極端に大きな値を除外）
    valid_depth = (depth > 0) & (depth < 10.0) # 10m以内の深度のみ有効

    # 視差の連続性チェック（ラプラシアンフィルタでエッジを検出）
    ddx = cv2.Sobel(disparity, cv2.CV_32F, 1, 0, ksize=3)
    ddy = cv2.Sobel(disparity, cv2.CV_32F, 0, 1, ksize=3)
    disparity_smoothness = np.sqrt(ddx**2 + ddy**2)
    smooth_mask = disparity_smoothness < threshold

    return valid_disparity & valid_depth & smooth_mask

```

使用例

```
def main():  
    # 深度推定器の初期化  
    estimator = StereoDepthEstimator('stereo_rect_maps.npz')  
  
    # 画像の読み込み  
    left_img = cv2.imread('left.png', cv2.IMREAD_GRAYSCALE)  
    right_img = cv2.imread('right.png', cv2.IMREAD_GRAYSCALE)  
  
    if left_img is None or right_img is None:  
        raise FileNotFoundError("画像ファイルが見つかりません")  
  
    # 深度マップの計算  
    depth, disparity, confidence_mask = estimator.compute_depth(left_img, right_img)  
  
    # 結果の保存 (オプション)  
    cv2.imwrite('depth.png', (depth * 1000).astype(np.uint16))  
# mm単位で保存  
    cv2.imwrite('disparity.png', (disparity * 16).astype(np.uint16))  
    cv2.imwrite('confidence.png', (confidence_mask * 255).astype(np.uint8))  
  
if __name__ == "__main__":  
    main()
```

設計上のトレードオフと運用上のリスク：

- 精度 vs. 遅延：高品質マッチャおよびニューラル手法は遅延を増大。歩行には低遅延・適度な精度と時空間平滑化を優先。
- ベースライン vs. オクルージョン：大きなベースラインは遠方奥行きを改善するが近傍オクルージョンと首へのマウントトルクを増加。
- 照明と表面特性：ステレオは無特徴または鏡面表面で失敗；冗長性のためフォールバックセンサ（TOF、構造化光）を含める。
- 校正と機械的摩耗：誤校正は系統的奥行きバイアスを生じバランスおよび操作タスクを危険にする。オンライン一貫性チェックおよび監督再校正トリガを実装。

人型システムへの具体的含意：

- 下位プランナへピクセル単位信頼度マップを提供し、信頼できない奥行きに基づく制御動作を回避。
- 高速動作に対してはステレオ・慣性融合を用い、短期間対応喪失を許容。

- ・障害物検出における誤陽性を削減するため、オンザフライ左右チェックおよび時空間フィルタリングのため計算予算を割り当てる。

21.3 視覚処理における主要な課題

ステレオ深度キューやフレーム単位の物体分類器に関する前述の議論は、2つの相補的な必要性、すなわちセンサノイズ下での頑健な幾何推定と外観変動下での信頼できる意味解釈を浮き彫りにした。本小節では、ヒューマノイド視覚システムにおいてこれらの必要性を橋渡しする動作上の課題を分析する。

ヒューマノイドロボットは制約のあるセンサエンベロップと高度に変動する環境に直面する。主要な課題は知覚、計算、統合のカテゴリに集約される。知覚上の問題は、シーン反射率、照明、センサノイズが結合した非理想的な画像形成に起因する。実用的な画像モデルは誤差源を定量化するのに役立つ：

$$I(u) = \rho(u) S(u) + n(u), \quad (178)$$

ここで $I(u)$ は画素 u における観測放射照度、 $\rho(u)$ は表面アルベド、 $S(u)$ は照明項（鏡面成分と指向性成分を含む）、 $n(u)$ はセンサノイズである。堅牢なエンジニアリングには、与えられた動作エンベロップでどの項が支配的かを推定し、それに応じた緩和策を設計する必要がある。

主要な知覚上の課題：

- ・照明変動と鏡面反射。屋内照明と屋外の太陽は局所的な飽和と非ランバートハイライトを引き起こす。これらの効果はオプティカルフローやテンプレートマッチングで使用される輝度一定性仮定を破壊する。
- ・モーションブラーとローリングシャッターアーティファクト。ヒューマノイドはしばしば移動する頭部にカメラを搭載する。高速な頭部運動は特徴検出器とモーションからの深度の両方を破損させるブラーを誘発する。
- ・オクルージョンとクラッター。人間の環境では物体や手の部分的なオクルージョンが頻繁に発生する。部分的観測可能性が検出、追跡、把持計画を複雑化する。
- ・スケールと視点変化。操縦タスクでは広いスケール範囲で物体を認識する必要がある。分類器は遠方と近接の視点にわたって一般化しなければならない。
- ・無模様または繰り返し表面。ステレオや特徴ベース手法は低テクスチャ領域で失敗し、深度ホールと曖昧なデータ関連付けを引き起こす。
- ・敵対的または予期しない視覚入力。反射、透明材料、意図的な敵対的パターンが分類器を誤導する可能性がある。

計算上およびシステムレベルの課題：

- ・リアルタイム制約。ヒューマノイドのバランスと操縦は制御ループのデッドラインを下回る知覚遅延を要求する。知覚パイプラインはタイミング制約を満たさなければならない：

$$\tau_{\text{percept}} + \tau_{\text{comm}} < \tau_{\text{control}}$$
- ・電力と熱制限。高パフォーマンス GPU は複雑なモデルを可能にするが、モバイルヒューマノイドでの電力スロットリングは持続的スループットを制約する。
- ・メモリと帯域幅。高解像度カメラは大きなデータレートを生成する。オンボードバスと相互接続

はマルチカメラリグのピークスループットを制限する。

- シムツリアルミスマッチ。シミュレーションで訓練されたモデルはハードウェアへの展開時に分布シフトを示す。ドメインランダムマイゼーションとフォトリアリズムはギャップを減少させるが除去はできない。

統合とフュージョンの問題：

- センサキャリブレーションのドリフト。カメラと IMU 間の相対外部パラメータは温度と機械的ストレスで変化する可能性がある。キャリブレーションが固定されていると再投影誤差が蓄積する。
- 時間的データ関連付け。追跡はオクルージョンと視点変化にもかかわらずフレーム間で検出を関連付けることを要求する。誤った関連付けは安全でない運動計画につながる。
- クロスモーダルコンフリクト。ライダや深度カメラは単眼意味の手がかりと矛盾する可能性がある。決定ロジックは信頼性を定量化し、一貫した推定値を選択しなければならない。

簡潔なエンジニアリング分析は、これらの要因が推定器誤差にどう影響するかを示す。カメラ内部パラメータ K と姿勢 (R, t) を使用する再投影ベース推定器の場合、3D 点 X の再投影残差は

$$e = \|u - \pi(K [R \mid t] X)\|_2, \quad (179)$$

ここで $\pi(\cdot)$ は透視除算である。 u の観測ノイズと (R, t) の不確実性は非線形に 3D 位置誤差に伝播する。共分散を明示的にモデル化することで、状態推定器は低信頼性測定値の重みを下げることができる。

実用的な緩和策とアーキテクチャ選択：

1. 頑健なフロントエンド前処理

- 動的露出制御とローカルトーンマッピングが飽和を減少させる。
- 時間的デブラーリングとローリングシャッタ補正は IMU タイムスタンプを使用してフレームを安定化する。
- 信頼度マスクが低品質画素を下流での使用に対してフラグ付けする。

2. 異種センサフュージョン

- RGB、深度、IMU を明示的なノイズモデルでフューズする。
- 深度信頼度とステレオ視差妥当性チェックを使用して外れ値を拒否する。

3. 計算を意識した知覚スタック

- マルチレゾリューションパイプラインを実装する：粗い意味的セグメンテーションの後に縦ののための高解像度 ROI 処理を行う。
- リアルタイムタスク（障害物検出、足底配置）を低遅延スレッドで優先する。

4. ドメイン頑健な訓練

- 訓練データでフォトメトリック拡張、合成鏡面反射、センサノイズを使用する。
- シミュレートされたカメラノイズをハードウェアと経験的に一致するようにキャリブレーションする。

実装例：カラーと深度を読み取り、露出補償カラー画像を計算し、深度信頼度マスクを公開する非同期ビジョンワーク。コードは頑健性のための最小限の実用的ステップを示す。

コードサンプル 76 露出を意識した処理のための非同期ビジョンワーカー

```
import cv2
import numpy as np
import threading
import time
from typing import Callable, Tuple, Optional

Frame = np.ndarray
Depth = np.ndarray
Confidence = np.ndarray
Timestamp = float
PublishFn = Callable[[Frame, Depth, Confidence, Timestamp], None]

class VisionWorker(threading.Thread):
    def __init__(
        self,
        frame_source: "FrameSource", # 依存性注入でテストブルに
        publish_fn: PublishFn,
        fps: float = 30.0,
        name: str = "VisionWorker",
    ) -> None:
        super().__init__(name=name, daemon=True)
        self._src = frame_source
        self._publish = publish_fn
        self._running = threading.Event()
        self._running.set()
        self._period = 1.0 / fps
        self._lock = threading.Lock()
        self._latest: Optional[Tuple[Frame, Depth, Timestamp]] = None

    def run(self) -> None:
        while self._running.is_set():
            t0 = time.perf_counter()
            color, depth, ts = self._src.read()
            if color is None or depth is None:
                continue

            # ガンマ補正+CLAHEで露出補正 (HDR対策)
            lab = cv2.cvtColor(color, cv2.COLOR_BGR2LAB)
```

```

l, a, b = cv2.split(lab)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
l = clahe.apply(l)
color_out = cv2.cvtColor(cv2.merge((l, a, b)), cv2.COLOR_LAB2BGR)

# 深度信頼度マスク生成
valid = (depth > 0.1) & (depth < 5.0)
median = cv2.medianBlur(depth, 5)
conf = np.abs(depth - median) < 0.05
depth_conf = valid & conf

self._publish(color_out, depth, depth_conf, ts)

# FPS制御
elapsed = time.perf_counter() - t0
sleep = max(0.0, self._period - elapsed)
time.sleep(sleep)

def stop(self) -> None:
    self._running.clear()
    self.join(timeout=1.0)

```

設計上のトレードオフと運用上のリスク：

- モデルの複雑さを増加させると精度は向上するが遅延と電力消費が増加する。最悪ケースの制御デッドラインを満たす最小限のモデルを選択する。
- 不確実な測定値の積極的な拒否は誤陽性を減少させるが、偽陰性を増加させて追跡の持続性を損なう。
- キャリブレーションされたノイズモデルなしでシミュレーションへの過度の依存は現実世界での脆い動作につながる。
- 失敗モードには、遅延した知覚による不安定化、誤関連付けによる危険な操縦、処理されない鏡面反射による壊滅的な深度誤差が含まれる。

エンジニアは要件キャプチャ中にこれらのトレードオフを定量化しなければならない。遅延予算、故障に耐性のあるフォールバック動作、安全で信頼できるヒューマノイド動作を維持するための定期的な再キャリブレーションを指定する。

22 ビジョンとロボティクスの統合

22.1 ビジョンセンサとハードウェア

検出、分類、深度推定といったコアビジョントaskを確立したうえで、ここではヒューマノイドロボットでそれらの機能を実現する物理センサとハードウェアの選択について考察する。以下の議論では、センシング能力を実用的な設計制約やオンボード知覚パイプラインの統合要件と結びつける。

ビジョンハードウェアの選択は、運用要件を満たす必要がある：閉ループバランスのためのレイテンシ、ヒューマンインタラクションのための視野（FOV）、操作の深度精度、移動型ヒューマノイドのための電力／重量バジェット。代表的なセンサクラスとそのエンジニアリング上のトレードオフは以下の通り：

- ・単眼 RGB：コンパクト、低消費電力、高解像度。セマンティック知覚や顔／ジェスチャ認識に使用。制限事項：計測深度は Structure-from-Motion または外部深度センサが必要となり、計算負荷が増大する。
- ・ステレオ RGB：2 台の校正済みカメラから受動的に深度を取得。利点：シンプルな光学系、能動光源を使わない設計では太陽光に強い。重要パラメータ：ベースライン B と焦点距離 f が深度感度を決定する。視差 d から深度 z は

$$[H]z = \frac{fB}{d}. \quad (180)$$

深度精度は視差分解能に依存；ベースラインを大きくすると遠距離精度が向上するが、機械的統合の複雑さが増す。

- ・RGB-D（構造化光、ToF）：高密度深度を直接取得。構造化光は屋内に優れ；Time-of-Flight（ToF）は照明に強いがマルチパスや環境光ノイズに弱い。環境と要求深度レンジに応じて選択。
- ・イベントカメラ：非同期・高時間分解能センサで高速移動や滑り検出に最適。低レイテンシ・高ダイナミックレンジを提供するが、専用アルゴリズムが必要でセマンティックタスクには直感的でない。
- ・LiDAR：走査 3D 点群で長距離・高精度。通常、マッピングとグローバルローカリゼーションに使用。高コスト、重量、胴体／脚部搭載では機械的または電子的に複雑。
- ・サーマル・多スペクトルカメラ：クラッターや悪条件下での人検出に有用で、災害対応ヒューマノイドロールに適する。

センサ選択は、知覚性能に直接影響するハードウェアレベルの特性を考慮する必要がある：

- ・シャッター方式：グローバルシャッターは高速ヘッドモーション時のモーション歪を回避。ローリングシャッターは高速動作で深刻なアーティファクトを引き起こす可能性がある。
- ・ダイナミックレンジと露光制御：高ダイナミックレンジは屋内・屋外を往復する際のクリッピングを軽減。
- ・フレームレートとレイテンシ：閉ループバランスと反射動作にはエンドツーエンド 50 ms 未満のレイテンシが必要。パイプラインバッファリングを最小化；ハードウェアタイムスタンプを優先。

- 解像度 vs 帯域：高解像度は計算・バス負荷を増大。GPU/CPU リソースやネットワーク帯域が限られる場合は ROI または圧縮エンコーダを使用。
- インターフェースとドライバの成熟度：量産グレードドライバ、標準プロトコル（USB3 Vision、GigE Vision、MIPI CSI-2）および ROS/ROS2 ドライバを持つセンサを優先し、迅速な統合を実現。
- 物理的制約：重量、重心影響、熱放散、配線経路、EMI シールド、機械的堅牢性。

マウンティングと外部パラメータ：センサ配置と剛体変換により、画像座標からロボット座標へのマッピングが決まる。内部・外部パラメータは既存ツールで校正する。ステレオやマルチセンサリグでは、動作中に外部パラメータが安定するよう機械的剛性を確保。代表的な配置：

1. 頭部：ナビゲーションとインタラクション用の広 FOV ステレオまたは RGB-D。
2. 胸部／胴体：マッピング用 LiDAR または長ベースラインステレオ。
3. 手部：近接操作用小型 RGB または構造化光深度。

センサと IMU 間の同期とタイミングは重要。分散システムにはハードウェアトリガ、Precision Time Protocol (PTP)、または GPS 同期クロックを使用。ソフトウェアのみのタイムスタンプはオフセットがばらつき、センサ融合を劣化させる。

センサ融合と処理アーキテクチャ：レイテンシとスループット目標を満たすようタスクを適切なハードウェアに割り当てる。

- 利用可能な場合、画素処理（デノイズ、補正）を ISP または FPGA にオフロード。
- ニューラルネットワーク推論には GPU（NVIDIA Jetson/Isaac）を使用し、リアルタイム制御ループは CPU に割り当てる。
- 深度ストリームに対して初期フィルタ（時間中央値、バイラテラル）を適用し、下流推定器のノイズを削減。

ベースライン選択と深度分解能の設計方程式はトレードオフを定量化するのに役立つ。ステレオで視差量子化を Δd (pixel) と定義すると、深度不確実性 Δz は概算で

$$[H]\Delta z \approx \frac{fB}{d^2} \Delta d. \quad (181)$$

目標距離 z_t と許容深度誤差 Δz_{\max} に対し、ベースライン B を

$$[H]B \geq \frac{z_t^2 \Delta d}{f \Delta z_{\max}}. \quad (182)$$

と選ぶ。これにより、距離が増加するに従う深度精度への 2 次のペナルティを明示する。

実装例：OpenCV ステレオブロックマッチングで視差と深度を計算。スニペットは、頭部搭載ステレオペアで補正済み視差を計測深度に変換する方法を示す。コードは校正済み内部パラメータとステレオ補正行列を仮定。

コードサンプル 77 Compute depth from stereo disparity using OpenCV.

```
import cv2
import numpy as np
from pathlib import Path
```

```

from typing import Tuple

# 無効視差マスク値
INVALID_DISPARITY: float = -1.0

def load_rectified_pair(left_path: str, right_path: str) -> Tuple[np.ndarray, np.ndarray]:
    """左右整流済画像をGRAYSCALEで読み込む"""
    left = cv2.imread(left_path, cv2.IMREAD_GRAYSCALE)
    right = cv2.imread(right_path, cv2.IMREAD_GRAYSCALE)
    if left is None or right is None:
        raise FileNotFoundError("画像読み込み失敗")
    return left, right

def compute_depth_from_disparity(
    disp: np.ndarray, focal_length_px: float, baseline_m: float
) -> np.ndarray:
    """視差→奥行(m)変換。無効視差はNaNにする"""
    depth = np.full_like(disp, np.nan, dtype=np.float32)
    valid = disp > 0.1
    depth[valid] = (focal_length_px * baseline_m) / disp[valid]
    return depth

def main() -> None:
    # パラメータ
    focal_length_px: float = 700.0
    baseline_m: float = 0.12
    out_path: Path = Path("depth_meters.exr")

    # 画像読み込み
    left, right = load_rectified_pair("left_rect.png", "right_rect.png")

    # StereoBMパラメータ (高速・品質バランス)
    stereo = cv2.StereoBM_create(numDisparities=128, blockSize=15)
    disp = stereo.compute(left, right).astype(np.float32) / 16.0

    # 視差→奥行
    depth = compute_depth_from_disparity(disp, focal_length_px, baseline_m)

```

```
# 32-bit float EXR保存
cv2.imwrite(str(out_path), depth)

if __name__ == "__main__":
    main()
```

エンジニアリング上の影響、トレードオフ、運用上のリスク：

- ・機械的剛性とコネクタ信頼性は、外部パラメータの安定性、ひいてはマッピング精度に直接影響する。
- ・ベースラインを大きくすると遠距離深度が向上するが、頭部慣性が増え動的バランスに影響。
- ・イベントカメラはレイテンシを削減するが専用アルゴリズムが必要；成熟度不足はシステム統合リスクを高める。
- ・環境光や反射面は ToF や構造化光深度に誤差を生じさせる；センサ冗長性を計画する。
- ・同期失敗はセンサ融合の不整合を引き起こし、バランスクリティカルな動作で致命的な制御エラーに至る。

設計者は、センサ精度、計算負荷、電力、物理的制約をバランスさせる必要がある。ベースライン、焦点距離、レイテンシ目標をシステム設計の初期段階で明示的に定量化し、後期統合失敗を回避すること。

22.2 ビジョンパイプラインの構築

前の小節では、カメラの選択、深度モダリティ、センサ配置の制約を確立し、それらがパイプラインの複雑さとタイミングに直接影響することを示した。ここでは、これらのハードウェア決定を人型ロボットのための具体的で展開可能なビジョンパイプラインに変換し、キャリブレーション、レイテンシ予算、モジュラーなソフトウェア設計を重視する。

問題定義と運用目標：信頼性のある 6-DoF ポーズ情報と意味的検出をモーションプランナーに一定のレイテンシで届けるビジョンパイプラインを構築する。パイプラインは可変照明、全身動作によるモーションブラー、四肢による断続的な遮蔽に耐えなければならない。主要な性能指標は以下の通り：

- ・エンドツーエンドレイテンシ（センサキャプチャからメッセージ公開まで） $\leq 50\text{ ms}$ 、反動的歩行調整のため；
- ・検出精度とポーズ誤差はタスク許容範囲内、例えば操作では位置 3 cm、姿勢 5°；
- ・CPU/GPU リソース予算はオンボード NVIDIA ハードウェアと互換。

技術的分解。実用的なパイプラインは明確な段階に分かれる：

1. 同期とキャプチャ：

- ・ハードウェアタイムスタンプと Precision Time Protocol (PTP) でカメラと IMU を整列；
- ・露出制御とローリングシャッター緩和を高速胴体動作に対応。

2. キャリブレーション：

- ・各カメラの内部パラメータとカメラ・ロボットベース間の外部パラメータ；
- ・視差から深度を得るためのステレオ整列、またはアクティブセンサの深度カメラ内部パラメータ。

3. 前処理：

- ・デノイズ、デバイヤ、作業色空間への変換；
- ・計算量とネットワーク帯域削減のための画像 ROI 選択。

4. 知覚モジュール：

- ・検出/分類（TensorRT による CNN 推論加速）；
- ・深度推定またはステレオと ToF センサ間の深度融合；
- ・ポーズ推定（PnP、ICP、または学習ベース 6-DoF 推定器）。

5. 融合とトラッキング：

- ・カルマンまたは粒子フィルタで IMU と運動学的事前情報と検出を融合；
- ・時間的関連付けで遮蔽下でも物体 ID を維持。

6. プランナーへのインタフェース：

- ・共分散考慮ポーズ推定を ROS2 メッセージで公開；
- ・レイテンシと信頼性の診断チャンネルを提供。

キャリブレーションと再投影誤差。ロボットベースに対するカメラ内部パラメータ K と外部パラメータ (R, t) について、3D ランドマーク X の再投影は観測画像点 u と一致すべきである。再投影残差を最小化：

$$[H]e(u, X) = \|u - \Pi(K[R|t]X)\|_2, \quad (183)$$

ここで $\Pi(\cdot)$ は同次正規化を行う。キャリブレーション手順はすべての観測にわたり $e(u, X)^2$ の和を最小化する。衝撃や温度変化による機械的ドリフトを検出するため、常時キャリブレーション残差モニタを維持する。

ステレオ深度とベースラインのトレードオフ。焦点距離 f 、ベースライン b のステレオペアにおいて、視差 d からの深度 Z は：

$$[H]Z = \frac{fb}{d}. \quad (184)$$

設計上の意味：

- ・小さい b は長距離での三角測定感度を低下；
- ・大きい b は機械的クリアランス要求とキャリブレーション感度を増加；
- ・近距離精度と身体制約をトレードオフして b を選ぶ。

レイテンシ予算とスケジューリング。各段階に時間を割り当ててレイテンシ予算を構築。50 ms 総量の例：

- ・キャプチャ+転送：8 ms；
- ・前処理+整列：6 ms；
- ・推論（GPU）：20 ms；
- ・融合+公開：8 ms；
- ・マージン：8 ms。

ROS2 で適切な QoS プロファイルによりソフトリアルタイムスケジューリングを実施。過負荷時に決定的ドロップを可能にするため、センサフレーム用リングバッファを使用。

実装パターン：明確なトピック契約を持つモジュラー ROS2 ノード。効率的な圧縮ストリームのため image transport を使用。CNN 推論を Jetson または PCIe GPU 上の TensorRT エンジンにオフロード。簿記と共分散伝播のため軽量 CPU スレッドを維持。

カメラを購読し TensorRT 検出器を実行し、ポーズ風メッセージを公開する ROS2 Python ノードの例。以下のスニペットは主要な統合ポイントを示すが、簡潔のためモデル構築詳細は省略。

コードサンプル 78 ROS2 subscriber node running GPU-accelerated detector and publishing pose

```
import os
import time
from typing import List, Tuple

import cv2
import numpy as np
import rclpy
from cv_bridge import CvBridge
from geometry_msgs.msg import PoseStamped
from rclpy.node import Node
from sensor_msgs.msg import Image

# TensorRT ラッパー（自前実装 or torch2trt 等）
try:
    import tensorrt as trt
    import pycuda.driver as cuda
    import pycuda.autoinit
except ImportError:
    trt = None
    cuda = None

class TrtWrapper:
    """TensorRT エンジンを読み、推論を実行する簡易ラッパー"""
    def __init__(self, engine_path: str):
        self.logger = trt.Logger(trt.Logger.WARNING)
        with open(engine_path, "rb") as f, trt.Runtime(self.logger) as runtime:
            self.engine = runtime.deserialize_cuda_engine(f.read())
        self.context = self.engine.create_execution_context()
        self.stream = cuda.Stream()
```

```
# 入出力バッファ確保
```

```
self.bindings = []
```

```
for binding in self.engine:
```

```
    size = trt.volume(self.engine.get_binding_shape(binding))
```

```
    dtype = trt.nptype(self.engine.get_binding_dtype(binding))
```

```
    host_mem = cuda.pagelocked_empty(size, dtype)
```

```
    device_mem = cuda.mem_alloc(host_mem.nbytes)
```

```
    self.bindings.append(int(device_mem))
```

```
    if self.engine.binding_is_input(binding):
```

```
        self.host_input = host_mem
```

```
        self.device_input = device_mem
```

```
    else:
```

```
        self.host_output = host_mem
```

```
        self.device_output = device_mem
```

```
def infer(self, img: np.ndarray) -> np.ndarray:
```

```
    # 前処理済み画像をコピー
```

```
    np.copyto(self.host_input, img.ravel())
```

```
    cuda.memcpy_htod_async(self.device_input, self.host_input, self.stream)
```

```
    self.context.execute_async_v2(bindings=self.bindings, stream_handle=self.stream)
```

```
    cuda.memcpy_dtoh_async(self.host_output, self.device_output, self.stream)
```

```
    self.stream.synchronize()
```

```
    return self.host_output
```

```
class VisionNode(Node):
```

```
    def __init__(self):
```

```
        super().__init__("vision_node")
```

```
        self.declare_parameter("model_path", "/models/det.trt")
```

```
        self.declare_parameter("input_size", [640, 480])
```

```
        self.declare_parameter("conf_thresh", 0.7)
```

```
        self.bridge = CvBridge()
```

```
        self.sub = self.create_subscription(Image, "camera/image_raw", self.image_cb,
```

```
        self.pub = self.create_publisher(PoseStamped, "vision/pose", 10)
```

```
# TensorRT 初期化
```

```
model_path = self.get_parameter("model_path").value
```

```
if not os.path.isfile(model_path):
```

```
    self.get_logger().error(f"モデルファイルが見つかりません: {model_path}")
```

```

        raise RuntimeError("モデルファイルが見つかりません")
self.trt = TrtWrapper(model_path)
self.input_size = tuple(self.get_parameter("input_size").value)
self.conf_thresh = self.get_parameter("conf_thresh").value

# カメラ内部パラメータ (仮置き)
self.K = np.array([[500.0, 0.0, 320.0],
                   [0.0, 500.0, 240.0],
                   [0.0, 0.0, 1.0]])
self.dist = np.zeros((4, 1))

# 3D モデル点 (単位メートル)
self.obj_pts = np.array([
    [-0.05, -0.05, 0.0],
    [ 0.05, -0.05, 0.0],
    [ 0.05,  0.05, 0.0],
    [-0.05,  0.05, 0.0]
], dtype=np.float32)

self.get_logger().info("VisionNode_初期化完了")

def image_cb(self, msg: Image) -> None:
    try:
        cv_img = self.bridge.imgmsg_to_cv2(msg, desired_encoding="bgr8")
    except Exception as e:
        self.get_logger().warn(f"cv_bridge_変換エラー: {e}")
    return

# 前処理
blob = cv2.resize(cv_img, self.input_size)
blob = cv2.cvtColor(blob, cv2.COLOR_BGR2RGB)
blob = blob.astype(np.float32) / 255.0
blob = np.transpose(blob, (2, 0, 1)) # HWC→CHW
blob = np.expand_dims(blob, 0) # 1×C×H×W

# 推論
raw = self.trt.infer(blob) # 例: [1, 6, 8400] → xyxyconf
dets = self._postprocess(raw)

if len(dets) == 0:

```

```

        return

    # 最も信頼度の高い検出を使用
    best = max(dets, key=lambda x: x[4])
    corners = self._bbox_to_corners(best[:4])

    # PnP
    ok, rvec, tvec = cv2.solvePnP(
        self.obj_pts, corners, self.K, self.dist, flags=cv2.SOLVEPNP_IPPE
    )
    if not ok:
        return

    # PoseStamped 生成
    pose = PoseStamped()
    pose.header.stamp = msg.header.stamp
    pose.header.frame_id = "base_link"
    pose.pose.position.x, pose.pose.position.y, pose.pose.position.z = tvec.flatten()
    from geometry_msgs.msg import Quaternion
    q = self._rvec_to_quaternion(rvec)
    pose.pose.orientation = q
    self.pub.publish(pose)

# 以下、補助メソッド
def _postprocess(self, raw: np.ndarray) -> List[Tuple[float, float, float, float,
# 簡易 NMS なし版
dets = []
raw = raw.reshape(-1, 6) # x1,y1,x2,y2,conf,class
for r in raw:
    if r[4] > self.conf_thresh:
        dets.append(tuple(r[:5]))
return dets

def _bbox_to_corners(self, bbox: np.ndarray) -> np.ndarray:
    x1, y1, x2, y2 = bbox
    return np.array([[x1, y1], [x2, y1], [x2, y2], [x1, y2]], dtype=np.float32)

def _rvec_to_quaternion(self, rvec: np.ndarray) -> Quaternion:
    from transforms3d.quaternions import quatAboutAxis, normQ
    angle = np.linalg.norm(rvec)

```

```

        axis = rvec.flatten() / angle if angle > 1e-9 else [0, 0, 1]
        q = quatAboutAxis(angle, axis)
        q = normQ(q)
        return Quaternion(x=q[1], y=q[2], z=q[3], w=q[0])

def main(args=None):
    rclpy.init(args=args)
    node = VisionNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()

```

運用上の考慮とリスク：

- 機械的振動はポーズ推定器を劣化；マウントダンピングが必須。
- 熱ドリフトは外部パラメータをずらす；定期的再キャリブレーションまたはオンライン外部適応をスケジュール。
- 単一モダリティへの過度依存は故障モードを引き起こす；可能な限りステレオ、ToF、IMU を融合。
- 知覚と制御計画間の GPU 競合はレイテンシ予算に違反；ワークロードを分割または推論を優先。

設計トレードオフの概要：

- 高精度には広いベースライン、より多くの計算、厳格なキャリブレーションが必要。
- 低レイテンシは単純なモデル、ROI クロップ、ハードウェア加速推論を好む。
- 遮蔽への頑健性はトラッキングと時間融合に利益あり、状態複雑性の代償を伴う。

エンジニアは統合中にパイプラインレイテンシ、キャリブレーション残差、故障率を定量化すべき。これらの指標を PTP に整列したタイムスタンプでログし、回帰をセンサ、ソフトウェア、または機械的要因に遡及できるようにする。

22.3 ケーススタディ：ジェスチャ認識

センサ選択とパイプライン設計を踏まえ、本ケーススタディではそれらの選択を実践的なヒューマンノイドタスクに適用する。カメラタイプ、キーポイント抽出、リアルタイム制約がジェスチャ認識のためのアルゴリズムおよびシステム設計にどう影響するかを示す。

問題定義. サービスヒューマノイドは, 安全で直感的な HRI のために人間の腕と手の小規模なジェスチャ語彙を解釈しなければならない. 要件は, エンドツーエンド遅延 200 ms 未満, 屋内照明下で 95% の分類精度, 部分的遮蔽でのグレースフルデグラデーションを含む. システムは組込み GPU 上で動作し, Isaac Sim で学習したモデルを実機に転送しなければならない.

技術分析. ジェスチャ認識は以下の段階に分解される:

- センサ取得: ロボットクロックに同期した RGB または RGB-D ストリーム;
- 空間前処理: 歪み補正, 検出器を用いた人物周囲のクロップ;
- 特徴抽出: 2D/3D キーポイントまたは密画像特徴;
- 時間モデリング: 動的ジェスチャのための系列分類器;
- 決定と作動: 安全チェック付きロボット動作へのマッピング.

キーポイントベースのパイプラインはデータ次元を削減し解釈性を高める. 人体姿勢・手キーポイント検出器を用いてフレームごとに状態ベクトル $x_t \in \mathbb{R}^n$ を生成し, n は 2D キーポイント数の 2 倍または 3D の場合は 3 倍とする. 観測ノイズを削減するため, 各キーポイントにカルマンフィルタ (ガウスノイズに対する線形推定器) を適用する:

$$[H]\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t (z_t - H\hat{x}_{t|t-1}), \quad (185)$$

ここで z_t は観測キーポイントベクトル, H は観測行列, K_t はカルマンゲインである. 平滑化により小さなジッタによるジェスチャ分類器の誤発動を削減する.

時間分類には, 長短期記憶 (LSTM) ネットワークのような再帰型ニューラルネットワーク (RNN) が時間依存性を捉える. LSTM はクラスロジットを出力し, ソフトマックスで確率に変換する:

$$[H]p_i = \frac{\exp(s_i)}{\sum_j \exp(s_j)}. \quad (186)$$

学習は, 歪んだジェスチャ頻度に対処するためクラスバランスを考慮した交差エントロピー損失を用いる.

設計選択と制約:

1. 特徴モダリティ: RGB-D からの 3D キーポイントは体の向きに対する頑健性を向上させるが, 帯域幅と処理時間が増加する.
2. 時間ウィンドウサイズ: 長いウィンドウは遅いジェスチャでの精度を上げるが, 遅延とメモリ使用量を増やす.
3. モデル複雑さ: より深い LSTM または時間畳み込みネットワーク (TCN) は精度を上げるが, より多くの GPU サイクルを要求する.
4. シミュレーションから現実への転送: Isaac Sim でドメインランダム化された手と衣服をレンダリングし, ドメインギャップを削減する.

評価指標と遅延予算. エンドツーエンド遅延 T_{E2E} を定義する:

$$[H]T_{E2E} = T_{\text{capture}} + T_{\text{pre}} + T_{\text{infer}} + T_{\text{post}}. \quad (187)$$

ターゲットハードウェア上でコンポーネントごとの時間を測定する. 最大 T_{E2E} を 200 ms とする. 誤陽性率 (FPR) と遅延重み付き精度を用いて安全臨界動作を捉える.

実装ノート. リアルタイムキーポイントのため軽量検出器 (例: MediaPipe hands/pose) を用いる. 正規化されたキーポイントの系列で LSTM を学習する. 胴体座標を基準に正規化し, 肩幅でスケールして人物不変特徴を実現する. フリッカを回避するためヒステリシス決定ルールを実装し, 信頼度閾値 τ 以上で同じ予測ラベルが k 連続フレーム必要ならコマンドを発行する.

デプロイ例コード. スニペットは, フレームを読み込み, キーポイントを抽出し, PyTorch LSTM を実行し, ROS でジェスチャを配信するコンパクトな推論ループを示す. リスト内のコメントは簡潔に留める.

コードサンプル 79 リアルタイム推論ループ: キーポイント抽出, LSTM 推論, ROS 配信.

```
import cv2
import numpy as np
import torch
import rclpy
from rclpy.node import Node
from std_msgs.msg import Int32
import mediapipe as mp

class GestureNode(Node):
    def __init__(self):
        super().__init__('gesture_node')
        self.declare_parameters(
            namespace='',
            parameters=[
                ('model_path', 'gesture_lstm.pt'),
                ('window_size', 30),
                ('conf_thresh', 0.8),
                ('hysteresis', 3),
                ('camera_id', 0),
            ]
        )

        self.model = torch.jit.load(self.get_parameter('model_path').value)
        self.model.eval()

        self.W = self.get_parameter('window_size').value
        self.tau = self.get_parameter('conf_thresh').value
        self.k_req = self.get_parameter('hysteresis').value

        self.window = []
```

```

self.consensus = []

self.gest_pub = self.create_publisher(Int32, 'gesture_id', 10)

self.mp_hands = mp.solutions.hands.Hands(
    static_image_mode=False,
    max_num_hands=1,
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5
)

self.cap = cv2.VideoCapture(self.get_parameter('camera_id').value)
if not self.cap.isOpened():
    self.get_logger().error("カメラが開けません")
    rclpy.shutdown()

timer_period = 0.033 # 30 FPS
self.timer = self.create_timer(timer_period, self.timer_callback)

def extract_keypoints(self, frame):
    img_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = self.mp_hands.process(img_rgb)
    if results.multi_hand_landmarks:
        lm = results.multi_hand_landmarks[0].landmark
        pts = np.array([[p.x, p.y, p.z] for p in lm], dtype=np.float32)
        return pts
    return None

def normalize_keypoints(self, pts):
    # 手首を原点に移動＋スケール正規化
    pts = pts - pts[0]
    scale = np.linalg.norm(pts[9] - pts[0])
    if scale > 0:
        pts /= scale
    return pts.flatten()

def timer_callback(self):
    ret, frame = self.cap.read()
    if not ret:

```

```

        return

    keypoints = self.extract_keypoints(frame)
    if keypoints is None:
        self.window.clear()
        return

    vec = self.normalize_keypoints(keypoints)
    self.window.append(vec)
    if len(self.window) > self.W:
        self.window.pop(0)

    if len(self.window) == self.W:
        seq = torch.tensor(np.stack(self.window)).unsqueeze(0).float()
        with torch.no_grad():
            logits = self.model(seq)
            probs = torch.softmax(logits[0], dim=0).numpy()
            label = int(np.argmax(probs))
            conf = float(np.max(probs))

        self.consensus.append((label, conf))
        if len(self.consensus) > self.k_req:
            self.consensus.pop(0)

    # 連続フレームで閾値超えで確定
    if all(l == label and c >= self.tau for l, c in self.consensus):
        msg = Int32()
        msg.data = label
        self.gest_pub.publish(msg)
        self.consensus.clear()

def main(args=None):
    rclpy.init(args=args)
    node = GestureNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

```

```
if __name__ == '__main__':  
    main()
```

データセットおよび学習推奨. シミュレートされたデータと実験室データの両方を収集する. Isaac Sim を用いて多様な体型, 衣服, 照明, カメラ内部パラメータをレンダリングする. 実データにランダム遮蔽とモーションブラーを加えて拡張する. 重み付き交差エントロピーとアーリーストップングを用いる. デプロイされたカメラを用いて実機で記録した別個のテストセットで検証する.

運用上の影響, 設計トレードオフ, およびリスク.

- トレードオフ: 頑健性のため RGB-D を選ぶか, 計算負荷と較正の簡素化のため RGB を選ぶか.
- 安全性: 誤検出は安全でないロボット動作を引き起こす可能性がある. 作動前に拒否層と近接チェックを実装する.
- リアルタイム制限: モデル複雑さは式 (370) の遅延予算を尊重しなければならない. ターゲット GPU でプロファイリングする.
- 転送リスク: 不十分なドメインランダム化は性能低下を引き起こす. 実機での微調整で軽減する.
- 頑健性: 照明変化と遮蔽は依然として主要な故障モードである. 安全臨界の場合は多モーダルフュージョン (IMU, オーディオ) を用いる.

23 高度なビジョン技術

23.1 意味的セグメンテーション

前小節でのビジョンパイプラインの検討とセンサ選択のトレードオフを踏まえ, ここではヒューマノイドロボットの運用能力として意味的セグメンテーションを考察する. 焦点は, ピクセル単位のラベルを運動, 操縦, HRI タスクへの信頼できる入力に変換するアルゴリズムとエンジニアリング選択にある.

問題定義と運用上の関連性. ヒューマノイドロボットは安全な歩行計画, 操作可能な物体の識別, 人間の文脈解釈のために, シーンを意味的に意味のある領域に分割しなければならない. したがってセグメンテーションモジュールは3つのエンジニアリング制約をバランスさせる必要がある:

- リアルタイムスループットで制御ループのデッドライン (反応タスクでは通常 10–50 Hz) を満たすこと,
- 多様な照明, 遮蔽, 衣服に対する頑健性,
- オンボード計算 (例: Jetson Xavier, RTX クラスの組み込み GPU) のためのコンパクトなモデルサイズ.

技術分析: モデルファミリーとトレードオフ.

1. エンコーダ・デコーダ (UNet ライク) アーキテクチャは高密度な空間復元を提供し, きめ細かい物体境界に有効である. 指先と物体の接触を要する操縦シナリオに優れる.
2. DeepLab で用いられるアトラス空間ピラミッドプーリング (ASPP) は過度にダウンサンプリン

グすることなく受容野を増大させ、シーンレイアウトと障害物分類の文脈理解を改善する。

3. 軽量バックボーン (MobileNetV3、EfficientNet-lite) はオンボード展開のための FLOP を削減するが、ラベルの粒度を若干犠牲にする。
4. 多モーダルフュージョン (RGB + 深度または RGB + イベントカメラ) は倉庫や非構造化住宅に典型的な低テクスチャまたは低照度条件下でのセグメンテーションを改善する。フュージョンは初期 (入力結合)、ミッドレベル (特徴フュージョン)、または遅延 (CRF またはアテンションベースのリファインメント) で実行できる。

損失関数と評価指標。クラス不均衡タスクには、重み付きピクセル単位のクロスエントロピーを境界認識損失と組み合わせて用いる。クラス重み w_c を用いたピクセル単位クロスエントロピーを次のように定義する：

$$[H]\mathcal{L}_{CE} = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C w_c y_{i,c} \log p_{i,c}, \quad (188)$$

ここで N はピクセル数、 C はクラス数、 $y_{i,c} \in \{0, 1\}$ は正解ラベル、 $p_{i,c}$ は予測確率である。最終ベンチマークには mean Intersection over Union (mIoU) を用いる：

$$[H]\text{mIoU} = \frac{1}{C} \sum_{c=1}^C \frac{TP_c}{TP_c + FP_c + FN_c}, \quad (189)$$

ここで TP_c, FP_c, FN_c はクラス c の真陽性、偽陽性、偽陰性である。

データ戦略と sim-to-real 転送。ヒューマノイド視点を捉えた多様なエゴセントリックデータセットを収集する。Isaac Sim でドメインランダム化シーン (センサノイズモデル、多様な照明、衣服テクスチャ、人間と操作対象による遮蔽) を生成して拡張する。以下の実用的パイプラインを用いる：

- ランダム化された物体配置で合成 RGB-D シーケンスを生成する。
- 光度補強とモーションブラーを適用する。
- スタイル転送または敵対的ドメイン適応を用いてリアリティギャップを削減し、合成画像と実画像の混合で学習する。

実装と展開の考慮事項。

- モデルを TensorRT で FP16 または INT8 に量子化する。精度低下とレイテンシ利得を測定する。
- 制御ループでフレーム間安定性のため、時間的スムージング (ロジットの指数移動平均) を用いる： $p_t = \alpha p_{t-1} + (1 - \alpha) \hat{p}_t$ 。
- 深度による幾何学的整合性チェックを適用する：セグメンテーションマスクを 3D に再投影し、閾値体積未満の孤立浮遊領域を除去する。

実用的パイプラインチェックリスト (エンジニア向け)：

1. センサキャリブレーション：RGB と深度の内部キャリブレーション、マルチセンサリグの外部キャリブレーション。
2. 前処理：レンズ歪み補正、同期深度アライメント。
3. モデル選択：レイテンシと精度ターゲットに基づくバックボーンとデコーダの選択。
4. 最適化：剪定、量子化、TensorRT へのコンパイル、消費電力のベンチマーク。
5. 後処理：形態学的クリーンアップ、境界リファインメントのための CRF、3D 整合性チェック。

6. 統合：セグメンテーション出力をナビゲーションと操縦のためのビヘイビアツリーノードに統合。

コード例：組み込み GPU でのリアルタイムセグメンテーションのための TensorRT を用いた簡潔なランタイムループ。スニペットは前処理、推論、深度整合性チェックのための簡単な後処理を示す。

コードサンプル 80 組み込み GPU での TensorRT を用いたリアルタイムセグメンテーション推論。

```
import cv2
import numpy as np
import tensorrt as trt
import pycuda.driver as cuda
import pycuda.autoinit # 自動初期化
import threading
import queue
import time
from typing import Tuple, Optional

# TensorRT エンジン読み込み
def load_engine(engine_path: str) -> trt.ICudaEngine:
    with open(engine_path, "rb") as f, trt.Logger() as logger, trt.Runtime(logger) as runtime:
        return runtime.deserialize_cuda_engine(f.read())

# 非同期推論ラッパー
class TRTPredictor:
    def __init__(self, engine_path: str, stream: cuda.Stream):
        self.engine = load_engine(engine_path)
        self.context = self.engine.create_execution_context()
        self.stream = stream
        self.bindings = []
        for binding in self.engine:
            size = trt.volume(self.engine.get_binding_shape(binding))
            dtype = trt.nptype(self.engine.get_binding_dtype(binding))
            host_mem = cuda.pagelocked_empty(size, dtype)
            device_mem = cuda.mem_alloc(host_mem.nbytes)
            self.bindings.append((int(device_mem)))
            if self.engine.binding_is_input(binding):
                self.host_in = host_mem
                self.device_in = device_mem
            else:
                self.host_out = host_mem
                self.device_out = device_mem
```

```

def infer(self, img: np.ndarray) -> np.ndarray:
    np.copyto(self.host_in, img.ravel())
    cuda.memcpy_htod_async(self.device_in, self.host_in, self.stream)
    self.context.execute_async_v2(bindings=self.bindings, stream_handle=self.stream)
    cuda.memcpy_dtoh_async(self.host_out, self.device_out, self.stream)
    self.stream.synchronize()
    out_shape = self.engine.get_binding_shape(1)
    return self.host_out.reshape(out_shape)

# 前処理
def preprocess(frame: np.ndarray) -> np.ndarray:
    img = cv2.resize(frame, (512, 512))
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).astype(np.float32) / 255.0
    mean = np.array([0.485, 0.456, 0.406], dtype=np.float32)
    std = np.array([0.229, 0.224, 0.225], dtype=np.float32)
    img = (img - mean) / std
    return np.transpose(img, (2, 0, 1)).ravel()

# 後処理
def postprocess(logits: np.ndarray, depth: Optional[np.ndarray] = None, min_vol: float):
    seg = np.argmax(logits, axis=0).astype(np.uint8)
    if depth is not None:
        # 深度が小さい領域を除外
        mask = depth > 0.1
        seg *= mask
    return seg

# 可視化
def visualize_segmentation(frame: np.ndarray, seg: np.ndarray) -> np.ndarray:
    color = cv2.applyColorMap(seg * 50, cv2.COLORMAP_JET)
    return cv2.addWeighted(frame, 0.7, color, 0.3, 0)

# メイン
def main():
    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
        raise RuntimeError("カメラが開けません")
    stream = cuda.Stream()
    predictor = TRTPredictor("model.trt", stream)

```

```

while True:
    ret, frame = cap.read()
    if not ret:
        break
    inp = preprocess(frame)
    logits = predictor.infer(inp)
    seg = postprocess(logits)
    display = visualize_segmentation(frame, seg)
    cv2.imshow("seg", display)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()

if __name__ == "__main__":
    main()

```

エンジニアリングへの影響、トレードオフ、運用上のリスク。

- レイテンシ対精度：高解像度モデルは境界精度を改善するが、制御デッドラインを逃す可能性がある。タスクレベルのレイテンシ予算で選択する。
- 電力と熱制約：組み込み GPU は持続的負荷でスロットルする。期待されるデューティサイクルでベンチマークする。
- 安全と故障モード：人間や障害物の誤セグメンテーションは衝突を引き起こす。未知領域を非航行可能とみなす保守的なフォールバック動作を実装する。
- 継続的学習：展開ロボットで失敗例を収集し、定期的に再学習し、実世界の失敗サンプルに重みを付ける。

設計者は頑健な多モーダルフュージョン、量子化対応学習、ランタイムでのセグメンテーション信頼度モニタリングを優先すべきである。これらの対策により、多様な実世界環境における安全なヒューマノイド運用が維持される。

23.2 ヒューマノイドのためのビジュアル SLAM

意味的セグメンテーションに続き、ビジュアル SLAM はメトリックマップとエゴモーション推定を提供し、意味的ラベルをナビゲーションおよびマニピュレーションに対して実行可能にする。本小節では、ヒューマノイドの制約に適応したビジュアル SLAM 手法を説明し、コアな最適化定式化を提示し、リアルタイムヒューマノイド知覚のための実用的な実装パターンを示す。

ビジュアル SLAM（同時位置推定と地図構築）は、ヒューマノイド固有の要件を満たす必要がある。ヒューマノイドは重心の高い運動、歩行による頻繁な視点変化、およびアーティキュレートされた頭部関節を持つ。IMU、関節エンコーダ、および 1 つ以上のカメラを搭載するのが一般的である。問題

は、運動学的事前知識を用いてドリフトを削減し堅牢性を高めながら、ロボット状態とスパースメトリックマップをリアルタイムで推定することである。

問題設定と状態定義

- 目標：カメラ座標系のポーズ軌道 $T_{w,c}(t) \in \text{SE}(3)$ とマップ点集合 $X = \{x_i \in \mathbb{R}^3\}$ を計算する。
- 利用可能な観測：
 1. ビジュアルフレーム I_t (単眼、ステレオ、または RGB-D)。
 2. IMU 読み取り a_t, ω_t 。
 3. 頭部運動学を提供する関節エンコーダ角度 θ_t 。
- 運動学的事前知識：ロボットベースに対するカメラポーズは関節角度の関数である。ベースから頭部への順運動学変換を $T_{b,h}(\theta)$ とすると、

$$[H]T_{w,c}(t) = T_{w,b}(t)T_{b,h}(\theta_t)T_{h,c}, \quad (190)$$

ここで $T_{h,c}$ は固定されたマウント変換である。

最適化定式化フレーム・ツー・マップ推定は、ファクタグラフ上の最大事後確率 (MAP) 問題として定式化される。再投影残差と慣性残差を単一のコストに統合する：

$$[H]\hat{\mathcal{X}} = \arg \min_{\mathcal{X}} \sum_{k,i} \rho(\|r_{k,i}^{\text{proj}}\|_{\Sigma_{\text{cam}}}^2) + \sum_k \|r_k^{\text{imu}}\|_{\Sigma_{\text{imu}}}^2 + \sum_k \|r_k^{\text{kin}}\|_{\Sigma_{\text{kin}}}^2, \quad (191)$$

ここで

- $r_{k,i}^{\text{proj}}$ はキーフレーム k におけるマップ点 i の再投影誤差、
- r_k^{imu} は連続する状態間の事前積分 IMU 残差、
- r_k^{kin} はエンコーダ角度からの運動学的事前知識を (1) 式経由で課す、
- $\rho(\cdot)$ はロバストロスであり、 Σ は雑音共分散を表す。

ヒューマノイドのための主要アルゴリズム要素

1. センサ同期：カメラ、IMU、関節エンコーダを厳密にタイムスタンプする。小さな遅延は高速頭部運動中の投影誤差を削減する。
2. ビジュアルフロントエンド：フレーム間マッチングに ORB または SuperPoint 特徴を用いる。RGB-D では深度を用いて直接的な 3D 対応を計算する。
3. 慣性事前積分：フレーム間で IMU を積分し、ファクタグラフ用の r^{imu} 残差を形成する。
4. 運動学的ファクタ：ベースポーズと順運動学モデルから推定されるカメラポーズの不一致をペナルティとするファクタを作成する ((1) 式)。
5. キーフレーム選択とマップ管理：視差または特徴カバレッジが閾値を超えたときにキーフレームを選択する。
6. ループクロージャとポーズグラフ最適化：場所認識 (Bag-of-Words) を用いてループを検出し、グローバルポーズグラフ最適化を実行してドリフトを除去する。

運動学的残差の実用的な方程式推定ベースポーズ $T_{w,b}$ と観測関節角度 θ が与えられたとき、順運動学により予測カメラポーズを定義し、ビジュアル・慣性推定から得られたポーズと比較する。運動学

的残差は

$$[H]r^{\text{kin}} = \log(T_{w,c}^{\text{vis}-1} T_{w,b} T_{b,h}(\theta) T_{h,c}), \quad (192)$$

ここで \log は $\text{SE}(3)$ 対数写像で \mathbb{R}^6 へ写す。

実装パターンとコード例以下に、リアルタイムフロントエンドループを示す簡潔な Python スケルトンを示す。特徴抽出、RGB-D または三角測量ステレオによる solvePnP を用いたポーズ推定、事前積分 IMU 増分を用いた単純な EKF 風融合ステップを示す。本番用ライブラリ (GTSAM、ORB-SLAM3、Ceres 等) と置き換えて完全ファクタ最適化を行うこと。

コードサンプル 81 ヒューマノイドビジュアル・IMU フロントエンドループ (簡略化)

```
import cv2
import numpy as np
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image, Imu, JointState
from cv_bridge import CvBridge
import tf_transformations as tf
from typing import Optional, Tuple, List

class VisualInertialOdometry(Node):
    def __init__(self):
        super().__init__('vio_node')
        self.bridge = CvBridge()

        # パラメータ
        self.declare_parameter('camera_topic', '/camera/image_raw')
        self.declare_parameter('depth_topic', '/camera/depth/image_raw')
        self.declare_parameter('imu_topic', '/imu/data')
        self.declare_parameter('joint_topic', '/joint_states')

        # ORB特徴量
        self.orb = cv2.ORB_create(nfeatures=1000, scaleFactor=1.2, nlevels=8)
        self.matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

        # カメラ内部パラメータ (ROSパラメータサーバから読み込み)
        K_vec = self.declare_parameter('camera_matrix', [0.0]*9).value
        self.K = np.array(K_vec).reshape(3,3)

        # 初期化
        self.prev_des: Optional[np.ndarray] = None
        self.prev_kp: Optional[List[cv2.KeyPoint]] = None
```

```

self.prev_depth: Optional[np.ndarray] = None
self.last_time = self.get_clock().now()

# サブスクライバ
self.create_subscription(Image, self.get_parameter('camera_topic').value,
                           self.rgb_callback, 10)
self.create_subscription(Image, self.get_parameter('depth_topic').value,
                           self.depth_callback, 10)
self.create_subscription(Imu, self.get_parameter('imu_topic').value,
                           self.imu_callback, 100)
self.create_subscription(JointState, self.get_parameter('joint_topic').value,
                           self.joint_callback, 10)

# バッファ
self.imu_buffer: List[Imu] = []
self.latest_depth: Optional[np.ndarray] = None
self.latest_joints: Optional[JointState] = None

def rgb_callback(self, msg: Image):
    rgb = self.bridge.imgmsg_to_cv2(msg, 'bgr8')
    t_cam = rclpy.time.Time.from_msg(msg.header.stamp)

    # IMUバッチ取得
    imu_samples = self.get_imu_between(self.last_time, t_cam)

    # 関節角度取得
    if self.latest_joints is None:
        return
    theta = np.array(self.latest_joints.position)

    # 特徴検出
    kp, des = self.orb.detectAndCompute(rgb, None)
    if self.prev_des is not None and len(kp) > 0:
        matches = self.matcher.match(des, self.prev_des)
        pts2d = np.float32([kp[m.queryIdx].pt for m in matches]).reshape(-1,1,2)
        pts3d = np.float32([self.depth_to_3d(self.prev_kp[m.trainIdx].pt,
                                              self.prev_depth) for m in matches])

        if len(pts3d) >= 6:
            # PnP

```

```

        success, rvec, tvec, inliers = cv2.solvePnPRansac(
            pts3d, pts2d, self.K, None, reprojectionError=2.0, confidence=0.99)
        if success and inliers is not None and len(inliers) > 6:
            T_vis = self.se3_from_rvec_tvec(rvec, tvec)

            # IMU事前積分
            delta_T_imu = self.preintegrate_imu(imu_samples)

            # 運動学ポーズ
            T_base = self.ekf.state_pose()
            T_kin = T_base @ self.fk_head(theta) @ self.T_h_c

            # 融合
            T_fused = self.fuse_se3([T_vis, T_base @ delta_T_imu, T_kin],
                                    weights=[0.6, 0.3, 0.1])
            self.ekf.update_pose(T_fused)

        # バッファ更新
        self.prev_des, self.prev_kp, self.prev_depth, self.last_time = des, kp, self.l

def depth_callback(self, msg: Image):
    self.latest_depth = self.bridge.imgmsg_to_cv2(msg, '32FC1')

def imu_callback(self, msg: Imu):
    self.imu_buffer.append(msg)
    # 古いデータ削除
    cutoff = self.get_clock().now() - rclpy.duration.Duration(seconds=1.0)
    self.imu_buffer = [im for im in self.imu_buffer
                        if rclpy.time.Time.from_msg(im.header.stamp) > cutoff]

def joint_callback(self, msg: JointState):
    self.latest_joints = msg

def get_imu_between(self, t_start: rclpy.time.Time, t_end: rclpy.time.Time) -> List[Imu]:
    return [im for im in self.imu_buffer
            if t_start <= rclpy.time.Time.from_msg(im.header.stamp) <= t_end]

def depth_to_3d(self, uv: Tuple[float, float], depth: np.ndarray) -> np.ndarray:
    u, v = int(uv[0]), int(uv[1])
    if 0 <= u < depth.shape[1] and 0 <= v < depth.shape[0]:

```

```

        z = depth[v,u]
        if z > 0:
            x = (u - self.K[0,2]) * z / self.K[0,0]
            y = (v - self.K[1,2]) * z / self.K[1,1]
            return np.array([x, y, z])
        return np.array([0.0, 0.0, 0.0])

def se3_from_rvec_tvec(self, rvec: np.ndarray, tvec: np.ndarray) -> np.ndarray:
    R, _ = cv2.Rodrigues(rvec)
    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = tvec.flatten()
    return T

def preintegrate_imu(self, imu_samples: List[Imu]) -> np.ndarray:
    # 簡易事前積分（実装は置換可能）
    delta = np.eye(4)
    if len(imu_samples) < 2:
        return delta
    dt = 0.01 # 仮
    for im in imu_samples:
        acc = np.array([im.linear_acceleration.x,
                        im.linear_acceleration.y,
                        im.linear_acceleration.z])
        delta[:3, 3] += acc * dt * dt * 0.5
    return delta

def fk_head(self, theta: np.ndarray) -> np.ndarray:
    # 簡易FK（実装は置換可能）
    return np.eye(4)

@property
def T_h_c(self) -> np.ndarray:
    # 頭部→カメラ変換（ROSパラメータから読み込み）
    return np.array(self.declare_parameter('T_head_camera', np.eye(4).flatten()).to_numpy())

def fuse_se3(self, Ts: List[np.ndarray], weights: List[float]) -> np.ndarray:
    # 対数空間で加重平均
    from scipy.spatial.transform import Rotation as R
    ws = np.array(weights)

```

```

ws /= ws.sum()
t_mean = sum(w * T[:3,3] for w, T in zip(ws, Ts))
quats = [R.from_matrix(T[:3,:3]).as_quat() for T in Ts]
# 球面線形補間簡易版
q_mean = sum(w * q for w, q in zip(ws, quats))
q_mean /= np.linalg.norm(q_mean)
T_out = np.eye(4)
T_out[:3,:3] = R.from_quat(q_mean).as_matrix()
T_out[:3,3] = t_mean
return T_out

class EKF:
    def __init__(self):
        self._pose = np.eye(4)
    def state_pose(self) -> np.ndarray:
        return self._pose.copy()
    def update_pose(self, T: np.ndarray):
        self._pose = T.copy()

ekf = EKF()

def main():
    rclpy.init()
    node = VisualInertialOdometry()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

エンジニアリングにおける影響とトレードオフ

- センサ選択：単眼システムは軽量だがスケール曖昧性に悩まされる。ステレオまたは RGB-D はメトリックスケールとモーションブレンダーへの耐性を追加する。IMU とエンコーダは高ダイナミックなヒューマノイド動作に必須。
- 計算：完全バンドル調整は精度を生むが CPU/GPU リソースを要求する。重い最適化を低周波バックグラウンドスレッドにオフロードする。
- 堅牢性：運動学的事前知識はドリフトを削減するがモデル誤差を伝播する；関節・カメラ変換を正確に校正する。
- 故障モード：ローリングシャッターと強いモーションブレンダーは特徴追跡を破壊する。人間がいるダ

イナミックシーンはループクロージャを複雑にする。

- 運用上のリスク：SLAM 故障は安全でない動作を引き起こす。保守的なフォールバック動作とセンサウォッチドッグを統合する。

設計者は精度、レイテンシ、および電力制約を SLAM バリエーション選択時に衡量すべきである。ヒューマノイドには、敏捷な動作と低い視点に対処するため、ビジュアル・慣性・運動学的密結合システムを優先する。

23.3 事前学習済みモデルの利用

本小節では、事前学習済みネットワークが知覚パイプラインをどのように加速し、SLAM 姿勢推定とセグメンテーション済みシーン幾何とインターフェースするかを説明し、ビジュアル SLAM とセマンティックセグメンテーションの流れを継続する。先行手法は、ヒューマノイド固有の視点に事前学習済みモデルを適応する際に有用な幾何学的事前知識と合成訓練データを供給する。

問題定義と運用上の関連性。ヒューマノイドロボットは、操縦、ナビゲーション、インタラクションのために堅牢で低遅延の知覚を必要とする。知覚モデルをゼロから学習することはコストがかかりデータが大量に必要である。事前学習済みモデルは開発時間を短縮するが、ドメインシフト、計算、安全性の課題を引き起こす。工学上の課題は、精度、遅延、信頼性の目標をヒューマノイドプラットフォーム上で満たすように、事前学習済みモデルを選択・適応・展開することである。

技術分析：モデル選択と適応戦略。

- 選択基準：

1. タスク整合性：出力（分類、検出、セグメンテーション、深度）に一致するタスクで事前学習されたアーキテクチャを選ぶ。
2. バックボーン効率：エッジ展開向けにモバイルフレンドリーなバックボーン（MobileNetV3、ResNet-18、EfficientNet-Lite）を優先する。
3. 入力分布：ステレオ整流画像やイベントカメラなど、センサモダリティを一致させる。
4. 転移可能性：大規模で多様なデータセット（ImageNet、COCO）で学習されたモデルを優先し、特徴の汎化を改善する。

- 転移学習モード：

1. 特徴抽出：バックボーンパラメータを凍結し、タスクヘッドを再学習する。ロボットデータセットが小さい場合に使用。
2. ファインチューニング：一部の層を解凍し、低い学習率で適応する。適度なラベル付きロボットデータが存在する場合に使用。
3. 完全再学習：ドメイン内データが豊富に存在する場合のみ。

- 最適化目的。ファインチューニングはタスク損失に正則化を加えて最小化する：

$$[H]\theta^* = \arg \min_{\theta} \sum_{i=1}^N \ell(f(x_i; \theta), y_i) + \lambda \mathcal{R}(\theta). \quad (193)$$

ここで ℓ はタスク損失（クロスエントロピーまたは IoU ベース）、 \mathcal{R} は L2 正則化である。

- ドメイン適応。シミュレーションから実環境へのシフトや頭部搭載カメラのバイアスを補償するため：

1. データ拡張：現実的な照明、モーションブラー、センサノイズ。

2. 教師なし適応：分布差異を最小化、例：MMD または敵対的ドメイン分類器。
 3. 自己教師付きファインチューニング：疑似ラベルと信頼度閾値を使用。
- 不確実性とキャリブレーション。ヒューマノイドは安全な判断のために予測信頼度を定量化しなければならない。ロジット z_i に温度スケールリングを適用しソフトマックスをキャリブレーションする：

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}.$$

T をホールドアウト検証セットでキャリブレーションし、信頼性を改善する。

実装ガイダンスと実践的パイプライン。

- データパイプライン：
 1. 姿勢と照明を横断する代表的なロボットカメラフレームを収集する。
 2. Isaac Sim を使用してエッジケースシーンを拡張し、正解データを収集する。
 3. SLAM 幾何を使用してドメイン一致した深度またはセグメンテーションマスクを生成し、弱教師ラベルとする。
- 訓練レジメン：
 1. 事前学習済みバックボーンから初期化する。
 2. 初期 N エポックは早期層を凍結し、徐々に解凍する（層単位）。
 3. 事前学習済みパラメータには低い学習率を使用する。
 4. キャリブレーション指標（ECE）と遅延を監視する。
- 展開と加速：
 1. ONNX にエクスポートし、TensorRT で NVIDIA Jetson および Orin 向けに最適化する。
 2. 許容精度損失時は INT8 量子化する。
 3. CPU/GPU 利用率をプロファイリングし、推論が制御ループデッドラインを満たすことを確認する。

実践的ファインチューニングのための PyTorch スニペット例。これは事前学習済みセグメンテーションバックボーンを読み込み、分類器を置換し、エンコーダを凍結し、訓練を設定する。バッチサイズとデバイスはハードウェアに応じて調整する。

コードサンプル 82 Fine-tune pretrained segmentation head (PyTorch).

```
import os
import json
import logging
from pathlib import Path
from typing import Dict, Any

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
```

```

import torchvision.transforms as T
import segmentation_models_pytorch as smp # pip install segmentation-models-pytorch

# ログ設定
logging.basicConfig(level=logging.INFO, format="%(asctime)s-%(levelname)s-%(message)s")
logger = logging.getLogger(__name__)

# ハイパーパラメータ（外部JSONで上書き可能）
CFG = {
    "num_classes": 5,
    "backbone": "resnet50",
    "pretrained": True,
    "lr_head": 1e-3,
    "lr_finetune": 1e-4,
    "freeze_epochs": 10,
    "total_epochs": 30,
    "batch_size": 8,
    "num_workers": 4,
    "device": "cuda" if torch.cuda.is_available() else "cpu",
    "save_dir": "./runs/segmentation",
    "resume": None,
}

class SegDataset(torch.utils.data.Dataset):
    """簡易Dataset例（独自実装に差し替え）"""
    def __init__(self, root: str, split: str, transform=None):
        self.transform = transform
        # ここに画像/マスクパスリストを読み込む処理を実装
        self.samples = []

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        img_path, mask_path = self.samples[idx]
        img = T.ToTensor()(Image.open(img_path).convert("RGB"))
        mask = torch.load(mask_path).long() # (H,W)
        if self.transform:
            img = self.transform(img)
        return img, mask

```

```

def build_model(num_classes: int, backbone: str, pretrained: bool) -> nn.Module:
    # AuxClassifier無しで軽量化
    model = smp.FCN(
        encoder_name=backbone,
        encoder_weights="imagenet" if pretrained else None,
        classes=num_classes,
        activation=None,
    )
    return model

def freeze_encoder(model: nn.Module) -> None:
    # エンコーダ全凍結
    for p in model.encoder.parameters():
        p.requires_grad = False

def unfreeze_encoder(model: nn.Module) -> None:
    # 後半ブロックだけ解凍（メモリ節約）
    for p in model.encoder.layer4.parameters():
        p.requires_grad = True

def train_epoch(model, loader, criterion, optimizer, device):
    model.train()
    running = 0.0
    for img, mask in loader:
        img, mask = img.to(device), mask.to(device)
        logits = model(img)
        loss = criterion(logits, mask)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running += loss.item() * img.size(0)
    return running / len(loader.dataset)

def validate(model, loader, criterion, device):
    model.eval()
    running = 0.0
    with torch.no_grad():
        for img, mask in loader:
            img, mask = img.to(device), mask.to(device)

```

```

        logits = model(img)
        loss = criterion(logits, mask)
        running += loss.item() * img.size(0)
    return running / len(loader.dataset)

def main(cfg: Dict[str, Any]):
    os.makedirs(cfg["save_dir"], exist_ok=True)
    writer = SummaryWriter(cfg["save_dir"])

    train_ds = SegDataset(root="./data", split="train", transform=T.Normalize(mean=[0.
    val_ds = SegDataset(root="./data", split="val",
transform=T.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225]))
    train_loader = DataLoader(train_ds, batch_size=cfg["batch_size"], shuffle=True,
num_workers=cfg["num_workers"], pin_memory=True)
    val_loader = DataLoader(val_ds, batch_size=cfg["batch_size"], shuffle=False, n

    model = build_model(cfg["num_classes"], cfg["backbone"], cfg["pretrained"]).to(cfg
    criterion = nn.CrossEntropyLoss(ignore_index=255)

    # 1段階目：ヘッドのみ学習
    freeze_encoder(model)
    optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()))

    start_epoch = 0
    if cfg["resume"]:
        ckpt = torch.load(cfg["resume"], map_location=cfg["device"])
        model.load_state_dict(ckpt["model"])
        optimizer.load_state_dict(ckpt["optimizer"])
        start_epoch = ckpt["epoch"] + 1

    best_loss = float("inf")
    for epoch in range(start_epoch, cfg["total_epochs"]):
        if epoch == cfg["freeze_epochs"]:
            # 2段階目：fine-tune
            unfreeze_encoder(model)
            optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.paran

        tr_loss = train_epoch(model, train_loader, criterion, optimizer, cfg["device"])
        val_loss = validate(model, val_loader, criterion, cfg["device"])
        writer.add_scalar("Loss/train", tr_loss, epoch)

```

```

writer.add_scalar("Loss/val", val_loss, epoch)
logger.info(f"Epoch{epoch:03d}|train{tr_loss:.4f}|val{val_loss:.4f}")

if val_loss < best_loss:
    best_loss = val_loss
    torch.save({
        "model": model.state_dict(),
        "optimizer": optimizer.state_dict(),
        "epoch": epoch,
        "cfg": cfg,
    }, Path(cfg["save_dir"]) / "best.pth")

writer.close()

if __name__ == "__main__":
    # JSON上書き例: python train.py config.json
    import sys
    if len(sys.argv) > 1:
        with open(sys.argv[1]) as f:
            CFG.update(json.load(f))
    main(CFG)

```

評価指標と安全性テスト。

- 平均 IoU、適合率/再現率、キャリブレーション誤差 (ECE)、推論遅延、推論あたりエネルギー $E = P \cdot t$ を測定する。
- エッジケースで検証：高速頭部回転、遮蔽、反射面、低照度。
- 制御に不確実性を統合：信頼度が低い場合は保守的動作をトリガする。

工学への影響、トレードオフ、運用上のリスク。

- トレードオフ：
 1. 精度 vs 遅延：より深いモデルは精度を向上させるが推論時間を増加させる。
 2. キャリブレーション vs スループット：事後キャリブレーションはコストを追加するが壊滅的な判断を減らす。
 3. 合成データ速度 vs 実データ妥当性：シミュレーションは訓練を加速させるがドメインシフトのリスクがある。
- 運用上のリスク：
 1. 分布外入力への過信は安全でない行動を引き起こす可能性がある。
 2. 継続的学習中の壊滅的忘却は以前に検証された動作を劣化させる可能性がある。
 3. モデル更新は安全クリティカルタスクでの回帰を避けるため厳格な回帰テストを必要とする。

展開チェックリストを採用する：モデルをヒューマノイドセンサにキャリブレーションし、ターゲットハードウェアでプロファイリングし、不確実性駆動フォールバックを含め、継続的再検証のためのデータパイプラインを維持する。

23.4 ロボティックビジョンの将来動向

事前学習済みモデルによるセマンティック事前知識と Visual SLAM が生成するポーズ整合マップを基盤に、次なる進歩はヒューマノイドが厳しいリソース制約下で知覚・予測・行動をいかに洗練するかにかかっている。以下の動向は運用上の重要性を優先する：低遅延知覚，継続的適応，人間中心環境における信頼できるシーン理解。

将来動向とその技術的役割

- ・イベント・ニューロモーフィックビジョンによる低遅延知覚。問題：従来のフレームカメラはモーションブラと固定レート取得を強い，高速な反応的バランスおよび操作と衝突する。分析：イベントカメラは輝度変化に比例した非同期スパイクを出力。処理は密なフレーム畳み込みから疎な時空間フィルタリングとスパイクニューラルネットワーク（SNN）へ移行。実装：イベントストリームをIMUとタイトな時刻同期で融合し，バランスコントローラに対してサブミリ秒状態更新を提供。リスク：SNN アルゴリズムは異なる学習パイプラインを要し，時間符号化の慎重なキャリブレーションが必要。
- ・コンパクト 3D シーン表現とニューラルフィールド。問題：ヒューマノイドは操作および移動のための幾何学的に整合しメモリ効率の良いマップを必要とする。分析：ニューラル放射場（NeRFs）と学習済み符号付き距離関数は幾何を連続関数に圧縮。リアルタイム利用には，これらを疎なボクセルグリッドまたは学習済み基底関数に蒸留。sim-to-real 移行中は微分可能レンダリングを用いてセンサ配置または合成照明パラメータを最適化。実用的公式：融合シーン推定 $m(x)$ を表現し，ベイズ融合により更新

$$[H]x_{\text{fused}} = \left(\sum_i P_i^{-1} \right)^{-1} \left(\sum_i P_i^{-1} x_i \right), \quad (194)$$

ここで x_i はモダリティ推定， P_i はその共分散。ガウス仮定下で平均二乗誤差を最小化。

- ・オンデバイス継続的・自己教師あり学習。問題：事前学習済みネットワークは新奇環境および新物体で劣化する。分析：継続学習手法は正則化，リプレイバッファ，動的アーキテクチャを組み合わせることで破滅的忘却を軽減。自己教師ではロボット自身のカメラからの時間的整合性および多視点整合性を用いてオンラインでデータにラベルを付与。実装上の制約：ストレージと計算は限られており，小さな優先付きリプレイバッファと定期的な蒸留をメインモデルへ行う。
- ・マルチモーダル融合とセマンティックアフォーダンス。問題：視覚セマンティクスのみでは物体引き渡し時の触覚・力キューを見逃す。分析：ビジョン，力・トルク，プロプライオセプションをタスクごとに各モダリティに重みを付ける注意機構で融合。エンジニアリング実践：タスクコンテキストとロボット状態を条件として注意重みを計算し，(1) のような共分散重み付き規則で融合。トレードオフ：センサ追加はヒューマノイドの重量・電力を増加させる。
- ・ハードウェア認識モデル圧縮およびスケジューリング。問題：ヒューマノイドのペイロードおよび熱予算はオンボード GPU を制限する。分析：ターゲットアクセラレータ向けにチューニングされた量子化，剪定，構造化スパース，ニューラルアーキテクチャ探索（NAS）がモデルサイズ

と遅延を削減。多目的最適化を定式化

$$[H] \min_{\theta} \alpha \mathcal{L}_{\text{task}}(\theta) + \beta T(\theta) + \gamma E(\theta), \quad (195)$$

ここで $\mathcal{L}_{\text{task}}$ はタスク損失, T は遅延, E はエネルギー。 α, β, γ は運用要件により選択。

- 不確実性定量化およびランタイム検証。問題：誤分類は人間の近くで安全でない行動をリスクとする。分析：ベイズ深層学習またはモンテカルロドロップアウトを用いて予測不確実性を生成。閾値を用いて保守的コントローラまたは人間介入をトリガ。運動学的制約との視覚推定整合性を検査する形式的ランタイムモニタと組み合わせる。

実装上のパターン

1. タスクごとに知覚目的を定義：遅延予算, 検出範囲, 許容偽陰性率。
2. 目的を (2) を通じてハードウェア制約にマッピングし, 圧縮・スケジューリング戦略を選択。
3. 共分散重み付き融合 (1) と注意ネットワークを用いてマルチモーダル融合を実装し, コンテキストに適応。
4. 境界付きリプレイと定期的な蒸留を用いて継続学習を展開し, 過渡的条件への過学習を防ぐ。

展開例スニペット：量子化セグメンテーションモデルを TensorRT にエクスポートし, オンボード GPU で低遅延推論を実現。リストはヒューマノイドビジョンパイプラインに適したモデルエクスポートと推論ループを示す。

コードサンプル 83 Export and run a quantized segmentation model with TensorRT

```
import os
import time
import logging
from pathlib import Path
from typing import Tuple

import torch
import torch.quantization as quant
import onnx
import tensorrt as trt
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np
from PIL import Image

# ログ設定
logging.basicConfig(level=logging.INFO, format="%(asctime)s_%(levelname)s_%(message)s")
logger = logging.getLogger(__name__)

# 定数
```

```

MODEL_HUB = "pytorch/vision:v0.10.0"
MODEL_NAME = "deeplabv3_resnet50"
ONNX_PATH = Path("seg_model.onnx")
ENGINE_PATH = Path("seg_model.trt")
CALIB_FRAMES_DIR = Path("calib_frames")
INPUT_SHAPE = (1, 3, 480, 640) # humanoid head camera
BATCH_SIZE = 1
MAX_CALIB_BATCH = 10
TRT_MAX_WORKSPACE = 1 << 30 # 1GB
TRT_LOGGER = trt.Logger(trt.Logger.WARNING)

class CalibDataset(torch.utils.data.Dataset):
    """ キャリブレーション用の簡易データセット """
    def __init__(self, root: Path, max_samples: int = MAX_CALIB_BATCH):
        self.imgs = sorted(root.glob("*.jpg"))[:max_samples]

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        img = Image.open(self.imgs[idx]).convert("RGB").resize((640, 480))
        return torch.from_numpy(np.array(img)).permute(2, 0, 1).float() / 255.0

def quantize_model(model: torch.nn.Module, save_path: Path) -> torch.nn.Module:
    """ 与えられたモデルを静的量子化し保存 """
    model.eval()
    model.qconfig = quant.get_default_qconfig("fbgemm")
    quant.prepare(model, inplace=True)

    # 校正
    dataset = CalibDataset(CALIB_FRAMES_DIR)
    loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=False)
    with torch.no_grad():
        for x in loader:
            _ = model(x)

    quant.convert(model, inplace=True)
    torch.save(model.state_dict(), save_path)
    logger.info(f"量子化モデルを保存: {save_path}")
    return model

```

```

def export_onnx(model: torch.nn.Module, path: Path) -> None:
    """PyTorchモデルをONNX形式でエクスポート"""
    dummy = torch.randn(INPUT_SHAPE)
    torch.onnx.export(
        model,
        dummy,
        str(path),
        opset_version=13,
        input_names=["input"],
        output_names=["output"],
        dynamic_axes={"input": {0: "batch"}, "output": {0: "batch"}},
    )
    onnx.checker.check_model(str(path))
    logger.info(f"ONNXモデルを保存: {path}")

def build_engine(onnx_path: Path, engine_path: Path) -> trt.ICudaEngine:
    """ONNXからTensorRTエンジンをビルド"""
    if engine_path.exists():
        with open(engine_path, "rb") as f, trt.Runtime(TRT_LOGGER) as runtime:
            return runtime.deserialize_cuda_engine(f.read())

    builder = trt.Builder(TRT_LOGGER)
    config = builder.create_builder_config()
    config.max_workspace_size = TRT_MAX_WORKSPACE
    config.set_flag(trt.BuilderFlag.FP16)

    network = builder.create_network(
        1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
    )
    parser = trt.OnnxParser(network, TRT_LOGGER)
    with open(onnx_path, "rb") as f:
        parser.parse(f.read())

    engine = builder.build_engine(network, config)
    with open(engine_path, "wb") as f:
        f.write(engine.serialize())
    logger.info(f"TensorRTエンジンを保存: {engine_path}")
    return engine

```

```

class TRTInference:
    """TensorRT推論ラッパー"""
    def __init__(self, engine: trt.ICudaEngine):
        self.engine = engine
        self.context = engine.create_execution_context()
        self.stream = cuda.Stream()

        # バッファ確保
        self.host_in = cuda.pagelocked_empty(
            trt.volume(engine.get_binding_shape(0)), dtype=np.float32
        )
        self.host_out = cuda.pagelocked_empty(
            trt.volume(engine.get_binding_shape(1)), dtype=np.float32
        )
        self.dev_in = cuda.mem_alloc(self.host_in.nbytes)
        self.dev_out = cuda.mem_alloc(self.host_out.nbytes)

    def infer(self, frame: np.ndarray) -> np.ndarray:
        """1フレーム推論"""
        np.copyto(self.host_in, frame.ravel())
        cuda.memcpy_htod_async(self.dev_in, self.host_in, self.stream)
        self.context.execute_async_v2(
            bindings=[int(self.dev_in), int(self.dev_out)], stream_handle=self.stream
        )
        cuda.memcpy_dtoh_async(self.host_out, self.dev_out, self.stream)
        self.stream.synchronize()
        return self.host_out.reshape(self.engine.get_binding_shape(1))

def main():
    # モデル取得・量子化
    model = torch.hub.load(MODEL_HUB, MODEL_NAME, pretrained=True)
    quantize_model(model, Path("seg_model_q.pth"))

    # ONNXエクスポート
    export_onnx(model, ONNX_PATH)

    # TensorRTエンジン構築
    engine = build_engine(ONNX_PATH, ENGINE_PATH)

    # 推論ループ

```

```

trt_inf = TRTInference(engine)
cap = cv2.VideoCapture(0) # カメラ例
while True:
    ret, frame = cap.read()
    if not ret:
        break
    frame = cv2.resize(frame, (640, 480))
    input_blob = np.expand_dims(frame.transpose(2, 0, 1), 0).astype(np.float32) /
    seg = trt_inf.infer(input_blob)
    # 後処理例: segを可視化
    cv2.imshow("seg", seg[0])
    if cv2.waitKey(1) == 27:
        break
cap.release()
cv2.destroyAllWindows()

if __name__ == "__main__":
    main()

```

エンジニアリングトレードオフと運用上のリスク

- 遅延対精度：積極的圧縮は遅延を削減するが安全上重要な知覚を劣化させる。(2) と保守的安全余裕でチューニング。
- 計算・熱予算：センサ追加と高フレームレートは電力・熱を増加。熱経路とスケジューリングを設計し、スロットリングを防ぐ。
- 継続学習リスク：無制御オンライン更新はモデルをドリフトさせる。制約付きリプレイとヒューマン・イン・ザ・ループ検証を用いる。
- センサ故障モード：イベントカメラ，深度センサ，LiDAR は雨，グレア，動的な人間ワークスペースで異なる故障特性を示す。モダリティ認識フォールバックと不確実性ベースゲーティングを実装。
- プライバシーと規制：人間環境の視覚データは法的・倫理的制約を引き起こす。可能な場合はオンデバイス匿名化と federated learning を適用。

これらの動向は知覚アルゴリズム，ハードウェアアクセラレーション，制御ループを横断した統合最適化を要し，ヒューマノイドを複雑な人間環境で応答性・安全性・能力を維持させる。

運動計画と制御

24 モーション制御の基礎

24.1 運動学と動力学の理解

前述のアクチュエータ選定とセンサ配置の検討を踏まえ、これらのハードウェア選択を安定したヒューマノイド動作に必要な形式的な運動学・動力学モデルに結びつける。精密なモデルはリアルタイム全身制御器と安全なバランス回復戦略を可能にする。

運動学は関節配置がエンドエフェクタ姿勢にどう写像されるかを定義する。ヒューマノイドにおいて、 q を一般化関節座標ベクトルとする。順運動学は q をフレーム姿勢 $x = f(q)$ に写像する。逆運動学は所望の作業空間姿勢 x_{des} が与えられたとき q を求める。ヒューマノイドの制御設計では2つの性質が支配的である：冗長性と特異点である。冗長性はロボットの駆動自由度が作業次元より多いときに生じる。特異点はヤコビアン $J(q)$ がランクを失う地点で発生し、有限な作業速度に対して無界の関節速度を引き起こす。

動力学は力がどう運動を生むかを記述する。一般化座標 q を用い、行列形式の剛体運動方程式は

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J_c(q)^\top f_c, \quad (196)$$

である。ここで

- $M(q)$ は対称正定質量行列；
- $C(q, \dot{q})\dot{q}$ はコリオリ・遠心力項を集めたもの；
- $g(q)$ は重力ベクトル；
- τ はアクチュエータトルクベクトル；
- $J_c(q)$ は接触点におけるヤコビアン；
- f_c は接触反力

である。

問題定義。ヒューマノイドの歩行と操縦において、以下を満たす制御トルク τ を計算せよ：

1. 所望の重心 (CoM) と足軌道を追従；
2. 接触制約下でバランスを維持；
3. アクチュエータ限界と関節限界を尊重。

技術解析と制御プリミティブ。

- 逆運動学 (IK)：作業目標と関節制約を満たす q_{des} を非線形最適化で解く。特異点近傍で正則化するため緩和最小二乗を用いる：

$$\delta q = J^\# \delta x = J^\top (JJ^\top + \lambda^2 I)^{-1} \delta x,$$

緩和係数 λ を伴う。

- 逆動力学 (計算トルク)：式 (eq:eom) のモデルを用いてフィードフォワードトルクを合成する。一般的な制御則は

$$[H]\tau = M(q)u + C(q, \dot{q})\dot{q} + g(q) - J_c^\top f_c, \quad (197)$$

ここで $u = \ddot{q}_{\text{des}} + K_d(\dot{q}_{\text{des}} - \dot{q}) + K_p(q_{\text{des}} - q)$ は関節空間で PD フィードバックを実装する.

- 作業空間（タスク空間）制御：エンドエフェクタ加速度を直接調整する. 作業空間慣性行列 $\Lambda = (JM^{-1}J^\top)^{-1}$ を用いてタスク空間力を計算する：

$$F = \Lambda(\ddot{x}_{\text{des}} + K_v(\dot{x}_{\text{des}} - \dot{x}) + K_p(x_{\text{des}} - x)).$$

関節トルクに変換する $\tau = J^\top F + N^\top \tau_0$, ここで N は運動学的ヌル空間へ射影し τ_0 は二次目標（例：姿勢）を実施する.

- ヌル空間射影と冗長性解決：優先順位付きタスクスタックに対して，関節速度

$$\dot{q} = J^\# \dot{x} + N \dot{q}_0, \quad N = I - J^\# J,$$

を計算する，ここで $J^\#$ は緩和疑似逆行列であり \dot{q}_0 は低優先度タスクを実行する.

実装（実用上の検討）. 実機ヒューマノイド実装には以下が必要：

- M , C , g のための頑健なモデル同定.
- 正確な J_c と f_c を供給する接触推定.
- 制御レート（トルクループで 500–1000 Hz）で動作するリアルタイム逆動力学ソルバ（例：Pinocchio, RBDL).
- 特異点近傍，関節限界違反，トルク飽和のための安全チェック.

コードリストは剛体動力学ライブラリを用いた計算トルク制御器を示す. API 呼び出しは選択したライブラリとセンサインタフェースに置き換えること.

コードサンプル 84 Pinocchio 風 API を用いた計算トルク制御ループ

```
import numpy as np
import pinocchio as pin
from typing import Optional, Tuple

class ComputedTorqueController:
    def __init__(self, urdf_path: str,
                  kp_diag: np.ndarray,
                  kd_diag: np.ndarray,
                  torque_limits: np.ndarray,
                  tip_ids: Optional[list] = None) -> None:
        # URDF読み込みとデータ生成
        self.model = pin.buildModelFromUrdf(urdf_path)
        self.data = self.model.createData()
        self.robot = pin.RobotWrapper(self.model)

        # ゲイン・制限値保存
        self.Kp = np.diag(kp_diag)
        self.Kd = np.diag(kd_diag)
```

```

self.torque_limits = torque_limits

# 接触先端フレームID（未指定時は両足）
if tip_ids is None:
    self.tip_ids = [self.model.getFrameId("l_sole"),
                    self.model.getFrameId("r_sole")]
else:
    self.tip_ids = tip_ids

def update(self, q: np.ndarray, v: np.ndarray,
           q_des: np.ndarray, v_des: np.ndarray,
           a_des: np.ndarray,
           f_ext: Optional[np.ndarray] = None) -> np.ndarray:
    # 順運動学・動力学一式
    pin.computeAllTerms(self.model, self.data, q, v)
    M = self.data.M
    n = self.model.nv

    # 加速度レベルPD
    e = q_des - q
    ed = v_des - v
    a_ref = a_des + self.Kd @ ed + self.Kp @ e

    # 接触ヤコビアン・外力補償
    J_stack = np.zeros((0, n))
    f_stack = np.zeros(0)
    for fid in self.tip_ids:
        J6 = pin.getFrameJacobian(self.model, self.data, fid,
                                  pin.ReferenceFrame.LOCAL_WORLD_ALIGNED)
        J_stack = np.vstack((J_stack, J6))
        if f_ext is not None:
            f_stack = np.hstack((f_stack, f_ext))

    # 計算トルク
    tau = M @ a_ref + self.data.nle
    if f_stack.size:
        tau -= J_stack.T @ f_stack

    # 飽和
    tau = np.clip(tau, -self.torque_limits, self.torque_limits)

```

return tau

設計トレードオフとリスク.

- モデル忠実度対計算：高忠実度動力学は制御を改善するが遅延を増やす．高制御レートのためモデル簡略化を用いる．
- コンプライアンスと安全：剛性インピーダンスは精密追従をもたらす．しかし低コンプライアンスは予期せぬ接触下で転倒リスクを増やす．ハイブリッドトルク／位置制御が性能と安全を両立できる．
- 特異点と緩和：特異配置への近傍は制御誤差を増幅する．緩和疑似逆行列は不安定性を緩和するがタスク精度を低下させる．
- センシングと接触推定：不正確な接触検出は J_c と f_c を破損し，誤った補償と潜在的な不安定を生じる．

具体的な工学上の示唆：実機への逆動力学制御器実装は，システム同定で動力学モデルを検証してから行う．低レベルのトルク飽和，遅延監視，特異点や接触近傍での監視移行を最優先とする．これらの対策は転倒リスクを低減し，歩行，全身操縦，回復動作の信頼性を向上させる．

24.2 バランシングアルゴリズム

これまでに導出された運動学的・動力学の関係は，重心（CoM）の運動が関節トルクおよび接触力にどう写像されるかを決定する．バランシングアルゴリズムはこれらの写像を用いて，CoM 射影を実現可能な支持領域内に留めるため，または回復が必要な場合に歩行を行うためにロボットがどこどのように動作すべきかを決定する．

問題の厳密な工学記述：アクチュエータ，接触，運動学的制限を満たしながら，ヒューマノイドを外乱下で安定に維持する．実用的には，CoM を制御し，上半身運動を通じて運動量を再配分し，地面反力を調整し，外乱後にロボットをキャプチャするための足置きを計画する制御アーキテクチャを選択することを意味する．

技術的分析

- 縮約モデルは，制御設計のための扱いやすく予測可能な動特性を提供する．
 - 線形倒立振子モデル（LIPM）は CoM 高さ z_c を一定と仮定し，水平 CoM x の動特性をモデル化する．その簡略化された形は

$$[H]\ddot{x} = \omega^2(x - p), \quad \omega = \sqrt{\frac{g}{z_c}}, \quad (198)$$

である．ここで p はゼロ・モーメント・ポイント（ZMP）または圧力中心（CoP）の位置を表す．式 198 は多くのプレビューおよび MPC コントローラの基礎である．

- 運動の発散成分（DCM），キャプチャポイント ξ と呼ばれるものは，CoM 動特性の不安定部分を分離する：

$$[H]\xi = x + \frac{\dot{x}}{\omega}, \quad \dot{\xi} = \omega(\xi - p). \quad (199)$$

DCM は安定性から離れて推移する； p または足の配置を制御することで ξ をキャプチャ領域へと駆動する．

- ヒューマノイドバランシングで用いられる古典的な制御プリミティブ：

1. アンクル戦略：足首トルクを調整して ZMP を支持多角形内に移動する。小さな外乱および平坦な地形に有効。
 2. ヒップ（胴体）戦略：角運動量を変化させてアンクル作動が飽和した際に大きな外乱を補償する。
 3. ステップ戦略：DCM をキャプチャするために新たな支持多角形を作るよう足を配置して接触を再構成する。
- 最適化ベースの全身制御は、多目的バランシングタスクを統合する。一般的な二次計画（QP）定式化は、剛体動力学

$$[H]M(q)\ddot{q} + h(q, \dot{q}) = S^T \tau + J_c^T \lambda, \quad (200)$$

を満たしながら、関節加速度 \ddot{q} 、関節トルク τ 、または接触力 λ を解く。摩擦円錐および接触一方向性に対する不等式制約を含む。QP は重み付きタスク誤差および制御努力を最小化し、(200) を等式制約として課す、または λ を決定変数として選択する。

実装パターン

- ZMP プレビュー制御器（離散時間 MPC）は計画された CoM 軌道を追従するための所望 ZMP シーケンスを予測する。プレビューゲインは (198) からオフラインで計算されリアルタイムで適用される。
- キャプチャポイントベースステップ： ξ を監視し、足部支持領域と比較し、 ξ が安全集合から外れた際に足踏みをトリガする。足位置は (199) の解析逆解または運学的ステップ到達可能性を課した短い予測幅 MPC から計算する。
- 全身 QP は優先度によってタスクを積み重ねる：
 1. 動力学および接触実現可能性を課す（ハード制約）。
 2. CoM / DCM 設定点を追従（高い重み）。
 3. 姿勢および関節限界を維持（ソフトタスク）。

日常的に使用される実用方程式

- 接触力からの ZMP：

$$[H]p_x = \frac{-\sum_i (y_i f_{z,i} - z_i f_{y,i})}{\sum_i f_{z,i}}, \quad (201)$$

ここで (x_i, y_i, z_i) および $(f_{x,i}, f_{y,i}, f_{z,i})$ は接触 i の位置および力である。

- LIPM 定常近似におけるキャプチャポイントステップ位置：

$$[H]x_{\text{step}} \approx \xi + e^{-\omega T}(\xi - p_{\text{current}}), \quad (202)$$

ここで T はステップ時間幅である。これは遊脚限界に合わせて調整される初期推定値を与える。

コード例：シンプルなキャプチャポイントステップ判定と足踏み目標の発行。

コードサンプル 85 Capture-point based step decision

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```

from __future__ import annotations

import numpy as np
from typing import Tuple, Final

# 物理定数
GRAVITY: Final[float] = 9.81 # [m/s^2]
COM_HEIGHT: Final[float] = 0.9 # [m]

# 歩行パラメータ
STEP_TIME: Final[float] = 0.6 # [s]
MAX_STEP_REACH: Final[float] = 0.4 # [m]

# 自然周波数
OMEGA: Final[float] = np.sqrt(GRAVITY / COM_HEIGHT)

def capture_point(com_pos: np.ndarray, com_vel: np.ndarray) -> np.ndarray:
    """
    キャプチャポイントを計算
    """
    return com_pos + com_vel / OMEGA

def propose_step(
    com_pos: np.ndarray,
    com_vel: np.ndarray,
    current_zmp: np.ndarray,
) -> np.ndarray:
    """
    次のZMP候補を提案（単一ステップ）
    """
    xi = capture_point(com_pos, com_vel)
    # 指数減衰モデルによる目標ZMP
    target_zmp = xi + np.exp(-OMEGA * STEP_TIME) * (xi - current_zmp)
    # 運動学的制約でクリップ
    delta = np.clip(target_zmp - current_zmp, -MAX_STEP_REACH, MAX_STEP_REACH)
    return current_zmp + delta

```

```
def main() -> None:
    # 初期状態
    com_pos = np.array([0.0, 0.0])      # [m]
    com_vel = np.array([0.3, 0.0])      # [m/s]
    current_zmp = np.array([0.0, 0.0])  # [m]

    next_zmp = propose_step(com_pos, com_vel, current_zmp)
    print(f"Proposed next ZMP: {next_zmp}")

if __name__ == "__main__":
    main()
```

工学への影響, トレードオフ, 運用上のリスク

- 縮約モデルは計算を大幅に簡素化するが, 接触近傍, 大きな胴体運動, または可変 CoM 高さでは忠実度を失う。高レート計画にはこれらを用い, 全身 QP で洗練する。
- アンクルおよびヒップ戦略はエネルギー効率が良いが, トルクおよび可動域に制限がある。ステップは頑健性を提供するが, タイミング, 衝突, 知覚要件を導入する。
- MPC および QP コントローラは制約処理を保証するが, 信頼できる状態推定を必要とし, センサドロップまたはモデル不整合下では失敗する可能性がある。最悪ケースのリアルタイム遅延に合う計算を予算化する。
- 設計上のトレードオフ: 小さな押しに対しては低遅延 LIPM ベースの反射を優先し, 計画されたステップには高遅延最適化を用いる。押し外乱および不整地をシミュレーションで検証し, 故障モードを定量化する。

24.3 ケーススタディ: ヒューマノイドの歩行

歩行に関するケーススタディは, 前小節の運動学的・動力学的プリミティブと, これまでに導入したバランス戦略を直接踏襲する。これらのモデルを統合し, 安定した足並び列とヒューマノイドに対して実行可能な関節指令を生成する実用的な ZMP ベース歩行パイプラインを構築する。

問題定義. 60–80 kg のヒューマノイドが平地で安定かつ省エネな歩容を生成する。制約として, アクチュエータトルク限界, 可動域限界, 足部接触摩擦, 小さな押しに対する擾乱抑制が挙げられる。制御アーキテクチャは組込み CPU/GPU でリアルタイムに動作し, Isaac Sim からハードウェアへ移行可能でなければならない。

技術解析. Linear Inverted Pendulum Model (LIPM) を用いて, Zero Moment Point (ZMP) の実行可能性を満たす重心 (CoM) 軌道を生成する。LIPM 近似では, 重心高さ z_c を一定とし, 胴体の動力学を無視する。連続 LIPM の動力学と ZMP の関係は

$$[H]\ddot{x} = \omega^2(x - p), \quad \omega^2 = \frac{g}{z_c}, \quad (203)$$

である。ここで $x(t)$ は水平 CoM, $p(t)$ は ZMP, g は重力加速度, ω は固有角振動数である。これを変

形すると計画に用いる ZMP 表式が得られる：

$$[H]p = x - \frac{1}{\omega^2}\ddot{x}. \quad (204)$$

設計パイプライン. 歩行コントローラは、ソフトウェア・センシングコンポーネントに直接対応するモジュールとして実装される：

1. 足並びプランナ：目標速度と旋回レートから目標足部姿勢列を計算する。
2. CoM 軌道生成：離散 LIPM 上で予見制御またはモデル予測制御（MPC）問題を解き、支持多角形内の参照 ZMP を追従する。
3. 逆運動学（IK）と全身 QP：CoM・足部姿勢を関節位置に写像し、関節限界・接触制約を課す。
4. 関節レベルコントローラ：インナーループ PD またはトルクコントローラで、必要に応じて逆動力学フィードフォワードを付加する。

実用的 MPC 定式化. (203) をサンプリング時間 T_s で離散化する。状態を $X_k = [x_k, \dot{x}_k]^T$ 、制御入力を離散 ZMP p_k とする。短いホライズン MPC は CoM・ZMP 追従を最小化する：

$$[H] \min_{\{p_{k+i}\}} \sum_{i=0}^{N-1} \|x_{k+i} - x_{k+i}^{\text{ref}}\|_{Q_x}^2 + \|p_{k+i} - p_{k+i}^{\text{ref}}\|_{Q_p}^2, \quad (205)$$

ただし (203) の線形動力学、現在の支持多角形内の ZMP 制約、アクチュエータレート限界を満たす。MPC は CoM 軌道と最初の ZMP 制御入力を出力する。

実装詳細. 有用なエンジニアリング選択：

- プレビューホライズン 1.0–1.5 s、 $T_s = 0.02$ s を標準サイズのヒューマノイドに用いる。
- 状態推定誤差に対処するため、支持多角形の保守的マージン内に ZMP を拘束する。
- MPC は小型 QP ライブラリ（OSQP）または高速化リカッチソルバで解く。
- 全身 QP でバランスを優先：足部接触を等式、胴体姿勢を軟追従、関節限界を不等式とする。

制御ループスケルトン例（Python）. 以下はシミュレーションループ内で動作し、CoM 積分、簡易足並び更新、PD 関節指令を示す。ハードウェア利用時はソルバ呼び出しを本番用 MPC・IK/QP ソルバに置き換える。

コードサンプル 86 簡易歩行ループ：LIPM 積分と関節 PD

```
import numpy as np
from typing import List, Tuple, Optional
import logging

# ロギング設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class LIPMController:
    """Linear Inverted Pendulum Model (LIPM) コントローラ"""
```

```

def __init__(self, com_height: float = 0.9, dt: float = 0.02, gravity: float = 9.8):
    self.g = gravity
    self.z_c = com_height
    self.omega2 = self.g / self.z_c
    self.dt = dt

    # 状態: [位置, 速度]
    self.x = np.array([0.0, 0.0], dtype=np.float64)

    # 歩容パラメータ
    self footsteps: List[np.ndarray] = []
    self.current_step_index = 0

def set_footsteps(self, footsteps: List[Tuple[float, float]]) -> None:
    """歩容シーケンスを設定"""
    self.footsteps = [np.array(step, dtype=np.float64) for step in footsteps]
    self.current_step_index = 0

def lipm_step(self, zmp_pos: np.ndarray) -> np.ndarray:
    """LIPMダイナミクスを1ステップ積分"""
    #  $\ddot{x} = \omega^2 * (x - p)$  の離散化
    accel = self.omega2 * (self.x[0] - zmp_pos[0])
    self.x += self.dt * np.array([self.x[1], accel])
    return self.x.copy()

def get_current_support_foot(self) -> Optional[np.ndarray]:
    """現在の支持脚位置を取得"""
    if self.current_step_index < len(self.footsteps):
        return self.footsteps[self.current_step_index].copy()
    return None

def update_support_foot(self) -> bool:
    """支持脚を次のステップに更新"""
    if self.current_step_index < len(self.footsteps) - 1:
        self.current_step_index += 1
        return True
    return False

def main():

```

```

# コントローラ初期化
controller = LIPMController(com_height=0.9, dt=0.02)

# 歩容シーケンス設定
footsteps = [(0.0, 0.0), (0.2, 0.0), (0.4, 0.0), (0.6, 0.0)]
controller.set_footsteps(footsteps)

# シミュレーションパラメータ
sim_time = 2.0
total_steps = int(sim_time / controller.dt)

# データ記録用
com_trajectory = []
zmp_trajectory = []

# メインループ
for step in range(total_steps):
    # 現在の支持脚取得
    support_foot = controller.get_current_support_foot()
    if support_foot is None:
        logger.warning("歩容シーケンスが終了しました")
        break

    # ZMPを支持脚中心に配置
    zmp_pos = support_foot.copy()

    # LIPMダイナミクスを積分
    com_state = controller.lipm_step(zmp_pos)

    # データ記録
    com_trajectory.append(com_state.copy())
    zmp_trajectory.append(zmp_pos.copy())

    # 10cm進んだら次の支持脚に移行（簡易的な遷移条件）
    if step > 0 and step % 50 == 0:
        controller.update_support_foot()

# ログ出力
if step % 100 == 0:
    logger.info(f"Step_{step}: CoM_pos={com_state[0]:.3f}, vel={com_state[1]:.3f}")

```

```

# 結果をnumpy配列に変換
com_trajectory = np.array(com_trajectory)
zmp_trajectory = np.array(zmp_trajectory)

return com_trajectory, zmp_trajectory

if __name__ == "__main__":
    com_traj, zmp_traj = main()

```

検証戦略. Isaac Sim で以下を実施：

- IMU/ビジョンに現実的なノイズを付加し、不完全な状態推定を再現。
- 外部擾乱：体重の 10% までの横方向インパルス。
- 摩擦係数を $\mu = 0.4$ まで低下させる。

成功指標は複数試行で転倒ゼロ、歩行タイミング頑健性、アクチュエータ電流値である。

エンジニアリングへの影響、トレードオフ、リスク：

- LIPM は動力学を簡略化し計算量を削減するが、大きな腕振り時の胴体慣性効果を過小評価する。
- MPC ホライズンを短くすると遅延は減るが長期的安定性を見落とす；長くすると計算量が増大。
- 保守的 ZMP マージンは推定誤差に対する頑健性を高めるが、最大速度を低下させる。
- 全身 QP は制約下で実行可能性を保証するが、信頼できる接触センシングを要し計算負荷が高い。
- 運用上のリスクとして、予期せぬ衝突時のアクチュエータ飽和、低摩擦面での滑り、センサ故障による誤 ZMP 推定が挙げられる。

モデル忠実度、計算予算、フォールバック動作を慎重にバランスし、シミュレーションからハードウェアへの安全な移行を確保する。

25 高度な運動計画

25.1 経路探索アルゴリズム

運動学およびバランス制御に関する前述の議論を踏まえ、経路探索は全身プランナーが安定した足取りおよび重心（CoM）軌道に変換可能な実行可能な幾何学的経路を提供しなければならない。プランナーは障害物の幾何学的形状、足取りの到達可能性、およびバランスコントローラによって課される動的安定性制約を尊重しなければならない。

ヒューマノイドロボットのための経路探索は、以下の要件を持つ工学上の問題である：

- 雑然とした人間サイズ的环境で衝突のない経路を生成する；
- 足取りおよび全身プランナーを支援する構造を明らかにする；
- 通過時間、エネルギー消費、および安定性の余裕をトレードオフする。

問題定義. 地図またはセンシングされた点群が与えられた場合、開始姿勢から目標姿勢への空間経

路を生成し、下流の足取り列および許容 CoM 運動の生成を支援する。形式的に、ロボットを障害物 O を持つワークスペース W 内の計画エージェントとして扱う。隣接サンプル間の到達可能性制約を満たす $P = \{p_0, \dots, p_N\}$ という列を計算し、各 $p_i \in W \setminus O$ とする。

グラフベース探索は、構造化され検証可能な経路を生成するため、ヒューマノイドにとって依然として中心的である。以下の実用的表現のいずれかを使用する：

1. 屋内シーンの高レベルナビゲーションのための 2D/2.5D 占有率グリッド。
2. 稀疏な複雑性シーンのための可視性グラフまたはロードマップ (PRM)。
3. 実行可能な足取りおよび立脚遷移を符号化した運動プリミティブの格子。

コストモデリング. ヒューマノイドにとって、エッジごとのコストは幾何学的長さとタスク固有のペナルティを含まなければならない。ノード $a \rightarrow b$ を結ぶエッジ e に対する実用的コストは

$$[H]c(e) = \ell(a, b) + \alpha E(a, b) + \beta \frac{1}{S(a, b) + \epsilon}, \quad (206)$$

である。ここで

- $\ell(a, b)$ はユークリッド距離、
- $E(a, b)$ は増分エネルギーを近似 (関節トルクプロキシ)、
- $S(a, b)$ は安定性余裕 (例：支持多角形から予測される ZMP 距離)、
- α, β は調整可能な重み、 ϵ はゼロ除算を回避。

ヒューリスティック設計. 許容ヒューリスティックは最適性を保証するのに役立つ。位相的 2D プランナーに対して、ユークリッド距離 $h(n) = \|\mathbf{x}_n - \mathbf{x}_g\|$ は許容である。エッジコストにエネルギーまたは安定性項が含まれる場合、ヒューリスティックを保守的にスケールリングする：

$$[H]h(n) \leq \min_{\text{paths } \pi \text{ from } n} \sum_{e \in \pi} c(e). \quad (207)$$

実際には、ARA* (anytime A*) を通じて有界された非最適性を維持しながら、速度のために膨張ヒューリスティックを使用する。

アルゴリズムアプローチとその工学上の役割：

- 占有率グリッド上の A*：屋内ナビゲーションに頑健、SLAM との統合が容易、足取りプランナーへの粗い誘導に適する。
- 運動プリミティブを持つ格子プランナー：足取り長、ヨー変化、横シフトを符号化し、直接足取り生成。離散足取り変換段階を除去。
- サンプリングベースプランナー (RRT、PRM)：高次元マニピュレーション文脈、または関節空間障害物が重要な全身再構成に有用。
- Anytime および増分プランナー (ARA*、D* Lite)：知覚更新または動的障害物が地図を変更する場所が必要。

ヒューマノイド展開のための実用的パイプライン：

1. グローバルプランナ (グラフベース) が地図上で粗い経路を計算。
2. 立脚プランナが粗い経路を到達可能性に制約された足取り列に変換。
3. 全身オプティマイザが動的バランスおよび衝突回避を保証するように関節軌道を改良。

4. 局所反応層が予期しない障害物または足取り滑りを処理。

計算制約. ヒューマノイドプランナは制御ループによって課されるリアルタイムデッドラインを満たさなければならない。階層的分解を使用して、高コスト計算をオフラインまたは低レートプランナーに移行する。高スループットが必要な場合、サンプリングベースプランナーで GPU 並列化を使用する。

実装例. 以下の Python スニペットは、占有率グリッド上の簡潔な A* を示す。エッジコストに近接ペナルティを追加して、より大きなクリアランスを持つ経路を優先することを示す。この連続経路を候補立脚点に投影することで足取りプランナーと統合する。

コードサンプル 87 占有率グリッド上の A* (近接ペナルティ付き)

```
import heapq
import numpy as np
from typing import Tuple, List, Dict, Optional, Generator, Set

Coord = Tuple[int, int]

def heuristic(a: Coord, b: Coord) -> float:
    """ ユークリッド距離 """
    return np.hypot(a[0] - b[0], a[1] - b[1])

def neighbors(node: Coord, grid: np.ndarray) -> Generator[Coord, None, None]:
    """ 8近傍で占有されていないセルを返す """
    x, y = node
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1),
                   (-1, -1), (1, 1), (1, -1), (-1, 1)]:
        nx, ny = x + dx, y + dy
        if 0 <= nx < grid.shape[0] and 0 <= ny < grid.shape[1] and grid[nx, ny] == 0:
            yield (nx, ny)

def proximity_cost(node: Coord, dist_map: np.ndarray) -> float:
    """ 障害物までの距離に応じたペナルティ """
    d = dist_map[node]
    return max(0.0, 0.5 - d) * 10.0 if d <= 0.5 else 0.0

def astar(start: Coord, goal: Coord, grid: np.ndarray,
          dist_map: np.ndarray) -> List[Coord]:
```

```
"""A*で最短経路を返す（経路なしの場合は空リスト）"""
```

```
open_q: List[Tuple[float, float, Coord, Optional[Coord]]] = []  
heapq.heappush(open_q, (heuristic(start, goal), 0.0, start, None))
```

```
came_from: Dict[Coord, Optional[Coord]] = {}
```

```
gscore: Dict[Coord, float] = {start: 0.0}
```

```
closed_set: Set[Coord] = set()
```

```
while open_q:
```

```
    _, g, current, parent = heapq.heappop(open_q)
```

```
    if current in closed_set:
```

```
        continue
```

```
    closed_set.add(current)
```

```
    came_from[current] = parent
```

```
    if current == goal:
```

```
        break
```

```
    for nbr in neighbors(current, grid):
```

```
        step = heuristic(current, nbr)
```

```
        tentative_g = g + step + proximity_cost(nbr, dist_map)
```

```
        if tentative_g < gscore.get(nbr, np.inf):
```

```
            gscore[nbr] = tentative_g
```

```
            heapq.heappush(open_q,
```

```
                (tentative_g + heuristic(nbr, goal), tentative_g, nbr,
```

```
# 経路再構築
```

```
path: List[Coord] = []
```

```
n: Optional[Coord] = goal
```

```
while n is not None:
```

```
    path.append(n)
```

```
    n = came_from.get(n)
```

```
return path[::-1] if path[-1] == start else []
```

工学上の意味、トレードオフ、およびリスク：

- 次元削減（ワークスペース→足取り→全身）は速度を改善するが、改良中に非実行可能な転送のリスクがある。
- 積極的なヒューリスティック膨張は計画時間を削減するが、エネルギーおよび衝突リスクを増加させる。
- 格子プランナーは検証しやすい足取り列を生成するが、様々な地形に対して運動プリミティブの

慎重なチューニングが必要。

- ・知覚エラーもしくは古い地図は危険な立足を引き起こす可能性がある；常に保守的な安定性余裕およびランタイムリプランニングを含める。
- ・計算負荷—特にサンプリングベースまたは最適化プランナーの—はオフロードまたはハードウェア加速を必要とする場合がある。

設計者は最適性、頑健性、およびリアルタイム要件をバランスさせなければならない。エッジコストに安定性およびエネルギーを明示的にモデル化し、ハードウェア実行前にシミュレーションで全身コントローラを通じて経路を検証する。

25.2 反応的計画 vs 先行的計画

これまでのグラフベースおよびサンプリングベースの経路探索の比較では、グローバルな経路の質と計算コストを重視した。これらの議論は、知覚や動的な不確実性の下で、ヒューマノイドが短い遅延の反応と長い予測計画をどうバランスさせるべきかという問題へと自然に繋がる。

反応的計画（反射制御）と先行的計画（予測計画）は、ヒューマノイドロボットにとって補完的な運用ニーズに対処する。反応的計画とは、現在のセンサ状態から計算される閉ループかつ低遅延の応答を指す。例としては、インピーダンス制御の変調、突然の滑りを回避するための即座の足踏み再計画、局所的ポテンシャル場による障害物回避がある。先行的計画は、ロボットと環境の動的なモデルを明示的に用いて、ホライズンにわたる軌道を最適化する。例としては、歩容安定化のためのモデル予測制御（MPC）や、予測された人間の動きを考慮した到達動作の軌道最適化がある。

問題設定。ヒューマノイドが人間が存在する動的環境で動作する際に、安全性とミッション完了を保証する制御アーキテクチャを設計せよ。主要指標は衝突確率、エネルギー消費、追従誤差、計算遅延である。反応的コントローラは遅延を最小化し予測を最小限に抑えることで安全性を優先する。先行的プランナはホライズン長 N と予測精度にわたる最適化を解くことで効率と意図認識を向上させる。

技術的分析。

- ・反応要素：フィードバック則または高速局所プランナとして実装する。障害物回避の一般的な反応則は、人工ポテンシャル $U(x)$ の負の勾配を用いる：

$$[H]u_{\text{react}}(x) = -k\nabla_x U(x), \quad (208)$$

ここで k は制御ゲイン、 x はロボット状態である。これにより多ステップ最適化を解くことなく即座の修正動作が計算される。

- ・先行要素：離散的な再計画時刻に解く有限ホライズン最適制御問題として定式化する。標準的な MPC 定式化は

$$[H] \min_{u_{0:N-1}} \sum_{k=0}^{N-1} \ell(x_k, u_k) + V_f(x_N) \quad \text{s.t.} \quad x_{k+1} = f(x_k, u_k), \quad x_0 = \hat{x}, \quad (209)$$

ここで ℓ はステージコスト、 V_f は終端コスト、 f は離散ダイナミクス、 \hat{x} は現在の状態推定値である。解は将来のイベントを予測しながら時間的に一貫したエネルギー効率の良い動作を生成する。

切替とブレンディング。純粋に反応的または純粋に先行的なシステムは現実的なシナリオで失敗す

る。実用的なアーキテクチャは、衝突までの時間または予測信頼度に基づくスケジューリング基準でそれらをブレンドする。近似衝突までの時間を定義：

$$[H]t_c \approx \min_{t \geq 0} \frac{\|p_r(t) - p_o(t)\|}{\|v_r(t) - v_o(t)\| + \epsilon}, \quad (210)$$

ここで $\epsilon > 0$ はゼロ除算を回避し、 p, v は位置と速度を表す。これを切替閾値として用いる： $t_c < T_{\text{react}}$ なら u_{react} を優先し、そうでなければ MPC を実行する。あるいは凸結合でブレンド：

$$[H]u = \alpha(t_c) u_{\text{react}} + (1 - \alpha(t_c)) u_{\text{mpc}}, \quad \alpha \in [0, 1], \quad (211)$$

ここで $\alpha(t_c)$ は t_c が減少するにつれ増加する。

実装上の考慮事項。

- 知覚遅延と不確実性は有効な予測ホライズンを短縮する。センサ共分散を用いて先行的計画への依存を減衰させる。
- 計算予算はホライズン長 N とソルバ反復回数を制約する。組込みヒューマノイドコントローラでは、ウォームスタート QP ソルバまたは疎直接法を優先する。
- 安全フォールバック：認証済みの反応的安全コントローラを常に維持し、先行的プランナを先取りできるようにする。

実用的アルゴリズムパターン。

1. カルマンフィルタまたは粒子フィルタを用いて動的障害物の信念を維持する。
2. t_c と障害物予測不確実性 Σ_{pred} を推定する。
3. $t_c < T_{\text{react}}$ または $\text{trace}(\Sigma_{\text{pred}}) > \sigma_{\text{th}}$ なら α を 1 に近づける。
4. そうでなければ利用可能な計算時間に合わせてホライズン長を調整し (290) を解く。

最小限の ROS2 互換疑似コードがこのループと切替ロジックを示す。

コードサンプル 88 衝突までの時間に基づく切替を伴う反応-先行プランナループ

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from rclpy.executors import MultiThreadedExecutor
from rclpy.callback_groups import ReentrantCallbackGroup
from nav_msgs.msg import Odometry
from sensor_msgs.msg import PointCloud2
from geometry_msgs.msg import Twist
import numpy as np
from threading import Lock
from typing import Optional
import asyncio
from concurrent.futures import ThreadPoolExecutor
```

```

class ReactiveMPCNode(Node):
    def __init__(self):
        super().__init__('reactive_mpc_node')

        # QoS設定：センサデータはベストエフォートで最新のみ
        sensor_qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        # トピックの購読・配信
        self.odom_sub = self.create_subscription(
            Odometry, '/odometry/filtered', self.odom_cb, sensor_qos)
        self.obstacle_sub = self.create_subscription(
            PointCloud2, '/obstacle_fusion', self.obstacle_cb, sensor_qos)
        self.cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)

        # パラメータ
        self.declare_parameter('T_proactive', 2.0)
        self.declare_parameter('sigma_th', 0.3)
        self.declare_parameter('N', 20)
        self.T_proactive = self.get_parameter('T_proactive').value
        self.sigma_th = self.get_parameter('sigma_th').value
        self.N = self.get_parameter('N').value

        # 共有データ保護
        self.lock = Lock()
        self.x_hat: Optional[np.ndarray] = None
        self.obs: Optional[np.ndarray] = None

        # 非同期MPC用executor
        self.executor = ThreadPoolExecutor(max_workers=1)

        # 周期タイマー (100 Hz)
        self.create_timer(0.01, self.control_loop, callback_group=ReentrantCallbackGro

    def odom_cb(self, msg: Odometry):
        with self.lock:

```

```

        p = msg.pose.pose
        v = msg.twist.twist
        self.x_hat = np.array([
            p.position.x, p.position.y, p.position.z,
            p.orientation.x, p.orientation.y, p.orientation.z, p.orientation.w,
            v.linear.x, v.linear.y, v.linear.z
        ])

def obstacle_cb(self, msg: PointCloud2):
    # PointCloud2 → numpy (簡易実装)
    pts = []
    for i in range(0, len(msg.data), msg.point_step):
        x = np.frombuffer(msg.data[i:i+4], '<f4')[0]
        y = np.frombuffer(msg.data[i+4:i+8], '<f4')[0]
        pts.append([x, y])
    with self.lock:
        self.obs = np.array(pts)

def get_obstacle_estimates(self) -> Optional[np.ndarray]:
    with self.lock:
        return self.obs

def get_state_estimate(self) -> Optional[np.ndarray]:
    with self.lock:
        return self.x_hat

def estimate_ttc(self, x: np.ndarray, obs: np.ndarray) -> float:
    # 簡易TTC: 最近傍障害物までの距離 / 速度ノルム
    if obs.size == 0:
        return np.inf
    pos = x[:2]
    vel = x[7:9]
    v_norm = np.linalg.norm(vel)
    if v_norm < 1e-3:
        return np.inf
    dists = np.linalg.norm(obs[:, :2] - pos, axis=1)
    return np.min(dists) / v_norm

def prediction_uncertainty(self, obs: np.ndarray) -> float:
    # 分散の最大固有値を不確実性指標とする

```

```

        if obs.size == 0:
            return 0.0
        cov = np.cov(obs[:, :2].T)
        return np.max(np.linalg.eigvals(cov))

def reactive_controller(self, x: np.ndarray, obs: np.ndarray) -> np.ndarray:
    # 単純なポテンシャル法による回避
    u = np.zeros(2)
    if obs.size == 0:
        return u
    pos = x[:2]
    for o in obs:
        diff = pos - o[:2]
        d = np.linalg.norm(diff)
        if d < 1e-3:
            continue
        u += diff / d * (1.0 / d**2)
    return np.clip(u, -1.0, 1.0)

async def solve_mpc_async(self, x: np.ndarray, obs: np.ndarray, horizon: int) -> C
    # 非同期MCP (ダミー実装)
    loop = asyncio.get_event_loop()
    return await loop.run_in_executor(self.executor, self._mpc_solve, x, obs, hori

def _mpc_solve(self, x: np.ndarray, obs: np.ndarray, horizon: int) -> np.ndarray:
    # 実際のMPC求解 (ここでは仮に0ベクトル)
    return np.zeros(2)

def compute_alpha(self, ttc: float, sigma: float) -> float:
    #  $\alpha$  はTTCと不確実性に応じて0~1で変化
    if ttc > self.T_proactive and sigma < self.sigma_th:
        return 0.0
    return 1.0

def control_loop(self):
    obs = self.get_obstacle_estimates()
    x = self.get_state_estimate()
    if x is None or obs is None:
        return

```

```

t_c = self.estimate_ttc(x, obs)
sigma = self.prediction_uncertainty(obs)
u_react = self.reactive_controller(x, obs)

# MPC非同期実行判定
if t_c > self.T_proactive and sigma < self.sigma_th:
    u_mpc_future = asyncio.ensure_future(self.solve_mpc_async(x, obs, self.N))
    # 即座に結果を待たず、次ループで利用（簡易）
    u_mpc = None
else:
    u_mpc = None

alpha = self.compute_alpha(t_c, sigma)
u = alpha * u_react + (1 - alpha) * (u_mpc if u_mpc is not None else u_react)

cmd = Twist()
cmd.linear.x = u[0]
cmd.angular.z = u[1]
self.cmd_pub.publish(cmd)

def main():
    rclpy.init()
    node = ReactiveMPCNode()
    executor = MultiThreadedExecutor()
    executor.add_node(node)
    try:
        executor.spin()
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計上のトレードオフと工学上の含意。

- 遅延 vs 最適性：反応的コントローラは低遅延と予期せぬ事象に対する頑健性を保証する。先行的プランナはエネルギーを削減し協調性を向上させるが、高遅延と正確な予測への依存という代

償を伴う。

- 安全認証：反応的安全層は長期保証を必要としないため認証が簡単になる。しかし過度に保守的となりミッション効率を低下させる。
- 知覚の限界：高センサノイズまたは断続的な遮蔽は α を高くしホライズンを短くする。
- リソース配分：オンボード計算資源は MPC を CPU、GPU、またはエッジサーバにオフロードするかを決定する。ネットワークオフロードは遅延と新たな故障モードを導入する。
- ヒューマンファクタ：人間の意図を予測したヒューマノイドの動作は自然に見える。過度に反応的な動作は人間にとって不安定で予測不能に見える。

運用上のリスク。

- 先行的予測への過度な依存はモデルが誤っている際に安全でない挙動を引き起こす。
- 厳密な不変条件なしの積極的なブレンディングは閉ループダイナミクスの安定性を侵害する可能性がある。
- 切替遷移はアクチュエータチャタやトルクスパイクを回避するために平滑化されなければならない。

具体的ガイドライン。

- 常に検証済みの反応的安全コントローラを含める。
- 計算認識型終端を備えた適応ホライズン MPC を用いる。
- 予測不確実性を定量化しブレンディング重みを変調するために用いる。
- フィールド試験中の故障モードを評価するために切替イベントをログに記録する。

25.3 衝突回避技術

これまでの反応的対計画的計画およびグラフまたはサンプリングベースの経路探索の議論を踏まえ、衝突回避は高レベルの大域軌道と低レベルの安全臨界運動指令との橋渡しとして機能する。全身運動学、動的安定性、センサ不確実性を調整しながら、ヒューマノイドプラットフォームで計算量的に扱い可能でなければならない。

問題定義と運用上の関連

- ヒューマノイドは以下を回避しなければならない：(1) 多数の関節における自己衝突、(2) 静的・動的障害物との衝突、(3) 回避機動によるバランス違反。
- 解は制御レート（一部のコントローラでは 100–1000 Hz）で実行し、視覚と proprioceptive センシングを統合し、アクチュエータトルクおよび関節制限を尊重しなければならない。

技術の分類

1. 反応的速度レベル手法

- Velocity Obstacles (VO) および Dynamic Window Approach (DWA) はロボット中心座標系で安全な速度指令を計算する。動的障害物に高速に反応し、粗い基部移動や腕エンドエフェクタ制御に適する。
- 形式 (VO)：障害物の相対状態と時間ホライズン τ が与えられたとき、禁止速度集合 $VO(\tau)$ は

τ 以内に衝突する速度を含む。コントローラは $VO(\tau)$ に含まれない v を効用 $f(v)$ を最大化するように選択する。

- 実用上の注意：VO/DWA は追加の運動学的チェックなしでは関節化ヒューマノイドの全身安全を保証しない。

2. 最適化ベース全身回避

- 衝突防止を二次計画 (QP) の不等式制約として定式化し、タスク空間目的と組み合わせる。これにより自己衝突、環境距離、安定性を観察しながら協調関節運動を強制する。
- 利点：複数タスク、関節制限、線形化衝突制約の明示的取り扱い。欠点：計算負荷が高い；信頼できる距離ヤコビアンと良好なウォームスタートが必要。

技術解析：線形化衝突制約

- q を現在の関節姿勢、 δq を小さな関節更新とする。障害物点 i に対し、 d_i を符号付き距離（安全なら正）、 $J_{d,i}$ を δq を d_i の変化率に写像するヤコビアンとする。距離を線形化すると

$$[H]d_i(q + \delta q) \approx d_i + J_{d,i} \delta q. \quad (212)$$

- 安全マージン d_{safe} を課し $d_i(q + \delta q) \geq d_{\text{safe}}$ を要求する。整理すると：

$$[H]J_{d,i} \delta q \geq d_{\text{safe}} - d_i. \quad (213)$$

- $A \delta q \leq b$ を用いる QP ソルバーでは -1 を乗じて $-J_{d,i} \delta q \leq d_i - d_{\text{safe}}$ を得る。

全身衝突回避のための QP 定式化

- 目的：所望タスク空間速度 v_{task} を追従するか、運動を正則化しながら公称関節指令からの逸出を最小化：

$$[H] \min_{\delta q} \|J_{\text{task}} \delta q - v_{\text{task}}\|^2 + \lambda \|\delta q\|^2. \quad (214)$$

- 制約：
 - 各障害物に対する線形化衝突制約（式 213）。
 - 関節制限： $q_{\min} \leq q + \delta q \leq q_{\max}$ 。
 - 動力学から導かれるオプションの速度／トルク境界。

実装スケッチ（実用的）

- 衝突ライブラリ（FCL、Bullet、符号付き距離場）を用いて符号付き距離と最近接点を計算。
- 最近接点の幾何ヤコビアンを距離勾配に投影して $J_{\{d,i\}}$ を計算。最近接点での外向き法線を \mathbf{n}_i とすると $J_{d,i} = \mathbf{n}_i^\top J_{p,i}$ （ $J_{p,i}$ はその点の位置ヤコビアン）。
- 制御レートでウォームスタートソルバ（OSQP、qpOASES）を用いて QP を更新し、QP が失敗した場合は保守的姿勢プリミティブにフォールバック。

実用的コード例（簡略）：cvxpy を用いた QP ベース全身回避

コードサンプル 89 QP ベース全身衝突回避（簡略）

```
import cvxpy as cp
```

```

import numpy as np
from typing import List, Optional, Tuple

def compute_safe_joint_update(
    J_task: np.ndarray,
    v_task: np.ndarray,
    Jd_list: List[np.ndarray],
    d_list: List[float],
    q: np.ndarray,
    q_min: np.ndarray,
    q_max: np.ndarray,
    d_safe: float = 0.05,
    lambda_reg: float = 1e-3,
    solver: Optional[str] = None,
    max_iter: int = 4000,
    eps_abs: float = 1e-6,
    eps_rel: float = 1e-6,
    verbose: bool = False,
) -> Tuple[np.ndarray, bool, str]:
    """
    衝突回避と関節限界を考慮した最適関節増分を計算する。
    戻り値: (dq_cmd, success, status)
    """
    m, n = J_task.shape
    assert J_task.ndim == 2 and v_task.ndim == 1 and v_task.size == m
    assert q.size == n and q_min.size == n and q_max.size == n
    assert len(Jd_list) == len(d_list)
    assert np.all(q_min <= q) and np.all(q <= q_max), "初期関節角が限界外"

    dq = cp.Variable(n)

    # タスク追従+正則化
    objective = cp.Minimize(
        cp.sum_squares(J_task @ dq - v_task) + lambda_reg * cp.sum_squares(dq)
    )

    # 衝突回避: Jd @ dq >= d_safe - d
    collision_constraints = [
        -Jd @ dq <= d - d_safe for Jd, d in zip(Jd_list, d_list)
    ]

```

```

# 関節限界
joint_limits = [q + dq >= q_min, q + dq <= q_max]

prob = cp.Problem(objective, collision_constraints + joint_limits)

# ソルバ選択
if solver is None:
    solver = cp.OSQP

try:
    prob.solve(
        solver=solver,
        warm_start=True,
        max_iter=max_iter,
        eps_abs=eps_abs,
        eps_rel=eps_rel,
        verbose=verbose,
    )
except Exception as e:
    return np.zeros(n), False, f"Solver error: {e}"

if dq.value is None:
    return np.zeros(n), False, prob.status

return dq.value, True, prob.status

```

設計・運用上の含意

- トレードオフ：
 - 反応的手法は計算量がスケラブルで動的障害物に対処するが、局所最小に陥り全身一貫性を維持できないリスクがある。
 - QP ベース手法は協調し制約を尊重した運動を生成するが、堅牢な距離ヤコビアン計算とより多くの CPU リソースを要する。
- 安定性・バランス：下位のバランスコントローラとの互換性を確保。同じ QP 内で重心 (CoM) タスクや ZMP (ゼロモーメント点) 境界を明示的に制約し、不安定化回避動作を防ぐ。
- センサ・モデル不確実性：遅延・雑音を考慮して d_{safe} を膨張する。知覚不確実性が定量化されている場合は確率的距離マージンを用いる。
- 故障モードと緩和：
 - QP 非可行性：検出し、凍結または実証済み安全軌道に沿って後退するなどの保守的反射動作に切り替える。

- 計算過負荷：CPU 負荷下では安全制約を優先し、重要性の低いタスクを降格させる。

具体的工学リスク

- ・過小評価された安全マージンは遅延やモデル誤差が存在する場合に衝突を引き起こす。
- ・誤った距離ヤコビアンは制約違反を引き起こす；ヤコビアンを有限差分テストで検証する。
- ・認証・展開にはフォールバック動作および時間境界応答に対する決定的最悪ケース解析が必要。

26 最適化と課題

26.1 動作効率の向上

これまでに導入した運動学的関係とバランス戦略を基に、安定性とタスク精度を保ちながらエネルギーおよびアクチュエータ負荷を削減することに本項では焦点を当てる。目的は実用的であり、バッテリー寿命を延ばし、熱ストレスを軽減し、ヒューマノイドプラットフォームの稼働時間を増やすことである。

問題定義。ヒューマノイドの運動計画は、エネルギー使用量、動的可行性、外乱に対する頑健性のトレードオフを伴う。エンジニアは次のようなコントローラを必要とする：

- ・電気的および機械的消費電力を最小化；
- ・アクチュエータトルクおよび速度制限を尊重；
- ・全身ダイナミクスと接触制約を満たす；
- ・組み込みハードウェアでリアルタイムに動作。

技術解析。エネルギー関連のコスト関数はアクチュエータ機構と電力電子機器に由来する。一般的な工学的目的は次の2つである：

1. 最小二乗トルク、これはピークモータ電流を減らしアクチュエータ寿命を延ばす傾向がある；
2. 積分機械電力最小、これは両方向の仕事に対してバッテリー使用量とより良く相関する。これらをブレンドした連続時間目的は次のように表される：

$$[H]J = \int_{t_0}^{t_f} (\alpha \tau(t)^\top \tau(t) + \beta |\tau(t)^\top \dot{q}(t)|) dt, \quad (215)$$

であり、 τ はアクチュエータトルク、 \dot{q} は関節速度、重み $\alpha, \beta \geq 0$ はミッション優先度に応じて選ばれる。

このコストはロボットダイナミクスと接触制約の下で最小化されなければならない。完全剛体方程式は

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = S^\top \tau + J_c(q)^\top \lambda, \quad (216)$$

であり、 M は慣性、 C はコリオリ／遠心力、 g は重力、 S はアクチュエータ選択、 J_c は接触ヤコビアン、 λ は接触力である。接触可行性は次を課す：

1. 一方向法線力 $\lambda_n \geq 0$ ；
2. 摩擦円錐制約、凸性のために線形ピラミッドで近似。

リアルタイム制御のための最適化定式化。接触が切り替わる非線形計画を組み込みコントローラで直接解くことは計算的に困難である。実用的なパターンは、低次元または線形化モデルを用いたモデル予測制御（MPC）である。短いホライズンと小さな時間刻みに対し、ダイナミクスを離散化し公称軌道まわりで線形化して二次計画（QP）を得る：

$$\begin{aligned}
 & \min_{\{\tau_k\}} \sum_{k=0}^{N-1} (\tau_k^\top R \tau_k + \Delta x_k^\top Q \Delta x_k) \\
 [H] \quad & \text{s.t.} \quad x_{k+1} = A_k x_k + B_k \tau_k + d_k, \\
 & G_k \tau_k \leq h_k \quad (\text{トルク, 接触, 摩擦制約}).
 \end{aligned} \tag{217}$$

ここで x は重心状態または低次元関節状態とできる。凸 QP は OSQP や qpOASES のような高速ソルバを許し、最新の組み込み CPU で 100 Hz 以上の更新レートを実現する。

実装スニペット。リストは CVXPY を用いた単一ステップ凸 QP 構築を示す。重み付きトルク目的、トルク制限、線形化重心ダイナミクスを実装しており、ヒューマノイドスタックの中レベルコントローラを代表する。

コードサンプル 90 低次元 MPC のための単一ステップトルク QP

```

import cvxpy as cp
import numpy as np
from typing import Optional, Tuple

class SingleStepMPC:
    """
    1ステップ最適トルク指令を計算する軽量MPC
    """
    def __init__(
        self,
        A: np.ndarray,
        B: np.ndarray,
        Q: np.ndarray,
        R: np.ndarray,
        tau_min: np.ndarray,
        tau_max: np.ndarray,
        solver: str = cp.OSQP,
    ) -> None:
        assert A.shape == (6, 6)
        assert B.shape == (6, 12)
        assert Q.shape == (6, 6)
        assert R.shape == (12, 12)
        assert tau_min.shape == (12,)
        assert tau_max.shape == (12,)

```

```

self.A = A
self.B = B
self.Q = Q
self.R = R
self.tau_min = tau_min
self.tau_max = tau_max
self.solver = solver

# 決定変数とパラメータを初期化
self.tau = cp.Variable(12)
self.x = cp.Parameter(6)
self.x_ref = cp.Parameter(6)

# コストと制約を構築
x_next = self.A @ self.x + self.B @ self.tau
cost = cp.quad_form(self.tau, self.R) + cp.quad_form(x_next - self.x_ref, self.Q)
constraints = [self.tau_min <= self.tau, self.tau <= self.tau_max]

self.prob = cp.Problem(cp.Minimize(cost), constraints)

def solve(self, x: np.ndarray, x_ref: np.ndarray) -> Tuple[bool, Optional[np.ndarray]]:
    """1ステップ最適化を実行し、成功フラグと最適トルクを返す"""
    self.x.value = x
    self.x_ref.value = x_ref
    try:
        self.prob.solve(solver=self.solver, warm_start=True)
        if self.prob.status != cp.OPTIMAL:
            return False, None
        return True, self.tau.value
    except Exception:
        return False, None

# 使用例（本ファイルが直接実行された場合）
if __name__ == "__main__":
    A = np.eye(6)
    B = np.zeros((6, 12))
    Q = np.eye(6)
    R = 1e-3 * np.eye(12)

```

```

tau_min = -50.0 * np.ones(12)
tau_max = 50.0 * np.ones(12)

mpc = SingleStepMPC(A, B, Q, R, tau_min, tau_max)

x = np.zeros(6)
x_ref = np.zeros(6)

ok, tau_opt = mpc.solve(x, x_ref)
if ok:
    print("最適トルク指令:", tau_opt)
else:
    print("最適化失敗")

```

モデリングおよびアルゴリズム選択. 忠実度と計算量をバランスする次の階層を用いる:

- 低次元モデル (重心または線形倒立振子) を長いホライズン計画に.
- 全身 QP を接触制約を伴う短いホライズン追従に.
- ローレベルインピーダンスまたは受動性ベースコントローラを実行安全のため.

評価指標. 次で改善を定量化する:

- メートルまたはタスクあたりのエネルギー (Wh m^{-1});
- ピークモータ電流と温度上昇;
- 外乱下でのタスク完了時間と失敗率;
- 頑健性マージン (例: 許容最大押し力).

設計上のトレードオフと運用上のリスク. エネルギー消費を削減すると大きな外乱に対する頑健性が低下することが多い. 重要なトレードオフは:

- 最適性対計算: 完全非線形最適化は最良のエネルギーを与えるが期限逸脱のリスクがある;
- モデル忠実度対凸性: 正確な摩擦・接触モデルを組み込むと非凸性が増す;
- 安全性対効率: 積極的なトルク最小化は減衰を減らし転倒リスクを増やす.

エンジニアリング推奨:

- 前回の解で QP ソルバをウォームスタートしリアルタイム期限を満たす;
- 直列弾性アクチュエータとコンプライアント制御を用いてエネルギー回収とピーク電流削減を可能にする;
- ハードウェア試験前に Isaac Sim のような高忠実度シミュレーションで検証;
- 接触遷移近傍では保守的なトルク制限を課す.

具体的な影響. 動作効率を改善すると稼働時間が増えメンテナンスが減る. しかし, 攻めの効率チューニングは予期せぬ相互作用中の故障を増幅することがある. エネルギー目標と安全マージンをバランスし, 性能向上を壊滅的故障から切り離す層状制御を実装せよ.

26.2 予期せぬ障害物への対処

前節では、参照軌道およびアクチュエータ指令を整形することで、運動効率の向上がエネルギー使用量と実行時間を削減する方法を確立した。予期せぬ障害物への対処は、これらの効率向上を維持しながら、突発的な環境変化下での安全性と安定性を保証することを目的とする。

予期せぬ障害物はヒューマノイドにとって二重の課題を提示する：高速な知覚から行動へのループと、バランスを維持するための拘束された全身ダイナミクスである。形式的には、効率的な参照からの逸出を最小化しながら、ダイナミクスと障害物回避の制約を厳しい時間制約内で満たす新しい制御系列 $u_{0:N-1}$ を生成する問題である。主要なシステム構成要素は以下の通り：

- ・不確定性推定を伴うリアルタイム知覚および物体追跡；
- ・短いホライズンを持つ反動的プランナ（しばしばモデル予測制御、MPC）；
- ・バランス回復のためのフォールバック安定化コントローラ。

問題定義。現在の状態 x_0 と参照軌道 $x_{0:N}^{\text{ref}}$ が与えられたとき、以下を満たす制御 $u_{0:N-1}$ を計算する：

1. 線形化または完全非線形ダイナミクスを尊重し、
2. ロボットの衝突幾何を占有領域の外側に保ち、
3. バランスのための重心（CoM）およびゼロモーメント点（ZMP）制約を維持し、
4. 利用可能な計算遅延内で実行する。

技術的分析。ダイナミクスと障害物制約を明示的に組み込むために MPC を使用する。MPC は有限ホライズン最適化を解き、最初の制御を繰り返し適用する。計算効率のため、現在の動作点周辺でダイナミクスを線形化し、最新のセンサ融合出力から符号付き距離勾配を用いて障害物制約を線形化する。典型的な二次計画（QP）定式化は以下の通り：

$$\begin{aligned}
 \min_{u_{0:N-1}} \quad & \sum_{k=0}^{N-1} (x_k - x_k^{\text{ref}})^{\top} Q (x_k - x_k^{\text{ref}}) + u_k^{\top} R u_k \\
 \text{s.t.} \quad & x_{k+1} = A_k x_k + B_k u_k \quad \forall k, \\
 [H] \quad & a_{i,k}^{\top} x_k + b_{i,k} \geq d_{\text{safe}} \quad \forall i, k, \\
 & C_{\text{bal}}(x_k, u_k) \in \mathcal{B} \quad \forall k, \\
 & u_{\min} \leq u_k \leq u_{\max}.
 \end{aligned} \tag{218}$$

ここで $a_{i,k}^{\top} x_k + b_{i,k} \geq d_{\text{safe}}$ は、時刻ステップ k における障害物 i への符号付き距離関数から導出される線形化された障害物制約である。 C_{bal} は ZMP 境界または CoM 支持多角形包含を実施するバランス関連の線形または凸制約を示す。

実用上の考慮事項：

- ・センシング遅延：線形化前に推定速度を用いて障害物状態を前方に伝播する。
- ・非凸障害物：凸プリミティブに分解するか、保守的な凸包を使用する。
- ・移動障害物：各ホライズンステップに対して予測障害物位置を $b_{i,k}$ に含める。
- ・フェイルセーフ切替：QP が非実行可能になった場合、低レベルの反射動作をトリガーする（例：

停止、スタンス拡大、後退)。

実装概要。知覚、予測、および MPC を制御ループに統合する。以下の表は軽量な処理チェーンを要約する：

- ・知覚：深度カメラ+ LIDAR を IMU と融合。出力：位置、速度、共分散を持つ障害物リスト。
- ・予測：MPC ホライズンにわたる各障害物に対する等速または学習済み運動モデル。
- ・制約生成：符号付き距離勾配を計算し、各障害物および時刻ステップに対して線形化制約を生成。
- ・ソルバ：ウォームスタート QP ソルバ (OSQP, qpOASES) を厳密なランタイム予算で実行。
- ・反射：MPC 非実行可能または遅延が閾値を超えた場合の PD ベースバランスフォールバック。

リスク軽減とトレードオフ：

- ・ホライズン長のトレードオフ：長いホライズンは先見性を改善するが、計算量と予測誤差への感度を増加させる。
- ・保守的制約は衝突を減らす、不必要な遠回り、エネルギーコストを引き起こす可能性がある。
- ・センサ故障モードには堅牢なフォールバックが必要：速度低下、安全マージン拡大、またはオペレータ引継ぎ。
- ・バランス対回避：積極的な障害物回避は転倒リスクのある姿勢変更を強制する可能性がある；常に ZMP/CoM 制約を優先的に確認する。

技術的影響：最悪ケースの知覚誤差下で実行可能性を保証するように制御スタックを設計する。計算予算を高速反射と中規模 MPC の両方に割り当て、応答性と最適性をバランスさせる。運用上のリスクには、誤検出が不要な回復ステップをトリガーし、敏捷な移動障害物に対する予測ミスマッチが含まれる。これらのトレードオフをテスト中に非実行可能イベントとフォールバック起動頻度をログに記録して定量化し、センサ融合、予測モデル、および制約保守性を繰り返し改善する。

26.3 複雑なタスクへのモーションプランニングのスケールアップ

これまでの障害物処理と動作効率に関する議論は、長いホライズン、多接触ヒューマノイドタスクにプランニングをスケールさせるための層横断的戦略を動機付ける。ここでは、30-50 自由度のヒューマノイドが「歩行-到達-操作」といった一連の動作を実行する際にプランナーを扱いやすく保つための分解、最適化構造、実用的な実装を分析する。

複雑なタスクでは、記号的シーケンシングと連続軌道最適化を組み合わせる必要がある。形式的には、ホライズン T にわたる離散化された軌道最適化を考える。決定変数は状態列 $X = \{x_0, \dots, x_T\}$ と制御列 $U = \{u_0, \dots, u_{T-1}\}$ である。標準的な直接記述定式化は以下の通り：

$$\begin{aligned} \min_{X, U} \quad & \sum_{t=0}^{T-1} \ell(x_t, u_t) + \ell_T(x_T) \\ [H] \quad & \text{s.t. } x_{t+1} = f(x_t, u_t), \quad t = 0, \dots, T-1, \\ & c(x_t, u_t) \leq 0, \quad \forall t, \\ & x_0 = x_{\text{init}}, x_T \in \mathcal{X}_{\text{goal}}. \end{aligned} \tag{219}$$

ここで f は離散時間ダイナミクスを符号化し、 c は接触幾何、関節限界、トルク境界、衝突回避をまとめる。接触相補性は非凸性を導入し、相補条件は通常 $0 \leq \lambda \perp \phi(x) \geq 0$ と書かれ、実際には緩和ま

たは罰則化される。

この問題をスケールさせるエンジニアリングアプローチは3つの原則を強調する。

1. 階層的分解およびタスク分解

- ・記号的プランナがサブゴールと接触モードの列を生成する（例：左足ステップ、右手グラスプ）。
- ・各モードごとに低次元連続最適化器が短いホライズンで局所問題を解く。
- ・これは組合せモード爆発を削減し、モーションプリミティブの再利用を可能にする。

2. 凸化と逐次手法

- ・逐次凸計画（SCP）は公称軌道周りでダイナミクスを線形化し、衝突制約を凸化する。凸部分問題を解き、公称を更新し、繰り返す。
- ・iLQR および微分動的計画法（DDP）は局所2次モデルを高速フィードバックポリシーに活用する。
- ・サンプリングベースプランナ（PRM や RRT）またはモーションライブラリからのウォームスタートが収束を加速する。

3. 再利用、並列化、および学習ベース提案

- ・タスク文脈でインデックス化されたパラメータ化プリミティブのモーションライブラリを構築する。プリミティブを再利用し、それらの間を補間する。
- ・学習済み提案またはニューラルサンブラを用いて、サンプルベースプランナを実行可能領域にバイアスする。
- ・衝突チェック、ダイナミクスロールアウト、勾配評価をマルチコア CPU または GPU で並列化する。Isaac Sim や GPU ベース衝突ライブラリが現実的なバッチ評価を提供する。

実用的な制約処理戦略

- ・接触処理：相補性をペナルティ法で緩和するか、勾配ベース最適化のための滑らかな接触モデルをシミュレーションで使用する。ペナルティ化接触力 λ は外力インパルスとしてダイナミクスに現れ、不等式罰則で近似実行可能性を強制する。
- ・衝突回避：障害物を符号付き距離場（SDF）で表現する。SCP 反復中に SDF 制約を線形化して凸部分問題を作る。
- ・トルクとバランス：遷移中の安定性維持のためゼロモーメントポイント（ZMP）または重心運動量制約を含める。

複雑度解析とスケーリングヒューリスティクス

- ・状態次元 n とホライズン T が変数数 $\mathcal{O}(nT)$ を決定する。計算量は少なくとも T に対して線形、密ソルバーでは n に対して超線形にスケールする。
- ・接触モードの組合せは潜在接触位置数に対して指数的に増大する。物理的ヒューリスティクスでモードを刈り込む：到達可能足配置、手到達マップ、安定性マージン。
- ・可変時間離散化を用いる：高精度操作時は細かい解像度、移動輸送時は粗い解像度。

記号的シーケンシングと軌道最適化を組み合わせた実装スケッチ：

コードサンプル 91 タスクシーケンシング、サンプリングベース初期化、軌道最適化を組み合わせた階層プラン

ナ

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from __future__ import annotations
import logging
from dataclasses import dataclass
from typing import List, Optional, Tuple

import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.executors import MultiThreadedExecutor
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
from geometry_msgs.msg import PoseArray
from std_msgs.msg import Header
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint

from your_robotics_lib.contact import ContactMode, ContactSampler
from your_robotics_lib.planning import PRM, RRT, SamplingPlanner
from your_robotics_lib.dynamics import RobotDynamics
from your_robotics_lib.optim import TrajectoryOptimizer, SCPConfig
from your_robotics_lib.utils import rate_limit, nan_guard

# ----- #
# ロギング設定
# ----- #
_logger = logging.getLogger("task_planner")
logging.basicConfig(level=logging.INFO)

# ----- #
# データ構造
# ----- #
@dataclass(frozen=True)
class SubGoal:
    """1サブゴールの記述"""
    name: str
    target_pose: np.ndarray # 4x4 SE(3)
```

```

        contacts: List[ContactMode]

    @dataclass(frozen=True)
    class LocalPlan:
        """最適化された局所軌道"""
        states: np.ndarray # (T+1, n)
        controls: np.ndarray # (T, m)
        contact_sequence: List[ContactMode]
        cost: float

# ----- #
# ノード本体
# ----- #
class TaskPlannerNode(Node):
    def __init__(self) -> None:
        super().__init__("task_planner")

        # ROS 2 パラメータ
        self.declare_parameter(
            "robot_description",
            "",
            ParameterDescriptor(
                type=ParameterType.PARAMETER_STRING,
                description="URDFファイルパス",
            ),
        )
        self.declare_parameter(
            "max_scp_iters",
            30,
            ParameterDescriptor(
                type=ParameterType.PARAMETER_INTEGER,
                description="SCP最大反復数",
            ),
        )
        self.declare_parameter(
            "control_freq",
            100.0,
            ParameterDescriptor(

```

```

        type=ParameterType.PARAMETER_DOUBLE,
        description="制御周波数[Hz]",
    ),
)

# 内部モジュール初期化
self._dynamics = RobotDynamics(self.get_parameter("robot_description").value)
self._contact_sampler = ContactSampler(self._dynamics)
self._sampling_planner: SamplingPlanner = PRM(self._dynamics)
self._optimizer = TrajectoryOptimizer(
    SCPConfig(max_iters=self.get_parameter("max_scp_iters").value)
)

# Publisher / Subscriber
self._traj_pub = self.create_publisher(
    JointTrajectory, "/local_plan/joint_trajectory", 1
)
self.create_subscription(
    PoseArray, "/goal_poses", self._goal_callback, 1
)

# 非同期実行用タイマー
self._timer = self.create_timer(0.1, self._spin_once)

# 内部状態
self._goal: Optional[List[SubGoal]] = None
self._executing: bool = False

# ----- #
# コールバック
# ----- #
def _goal_callback(self, msg: PoseArray) -> None:
    """目標姿勢を受信→タスク列生成"""
    self._goal = [
        SubGoal(
            name=f"subgoal_{i}",
            target_pose=self._pose_to_se3(pose),
            contacts=self._contact_sampler.sample(pose),
        )
        for i, pose in enumerate(msg.poses)
    ]

```

```

    ]
    self._executing = True
    _logger.info("新規ゴールを受信、計画開始")

# ----- #
# メインループ
# ----- #
def _spin_once(self) -> None:
    if not self._executing or self._goal is None:
        return

    for subgoal in self._goal:
        local_plan = self._plan_subgoal(subgoal)
        if local_plan is None:
            _logger.warning(f"{subgoal.name}の計画失敗、次を試行")
            continue

        self._publish(local_plan)
        self._wait_execution(local_plan)

    self._executing = False

# ----- #
# 計画関数
# ----- #
def _plan_subgoal(self, subgoal: SubGoal) -> Optional[LocalPlan]:
    """1サブゴールに対して局所最適軌道を生成"""
    # 接触モードを複数サンプリング
    for mode in self._contact_sampler.enumerate(subgoal.contacts):
        # 初期幾何経路生成
        init_path = self._sampling_planner.plan(
            start=self._dynamics.q0,
            goal=subgoal.target_pose,
            mode=mode,
        )
        if init_path is None:
            continue

        # 初期推定値生成
        X0, U0 = self._generate_initial_guess(init_path)

```

```

# SCP 最適化
Xopt, Uopt = self._optimizer.solve(
    X0, U0,
    dynamics=self._dynamics,
    constraints=self._build_constraints(mode),
    warm_start=True,
)
if Xopt is None:
    continue # 失敗 → 次のモードへ

nan_guard(Xopt) # 数値エラー監視
return LocalPlan(
    states=Xopt,
    controls=Uopt,
    contact_sequence=mode,
    cost=self._optimizer.last_cost,
)

return None # 全モード失敗

# ----- #
# ユーティリティ
# ----- #
@staticmethod
def _pose_to_se3(pose) -> np.ndarray:
    """geometry_msgs/Pose → SE(3) 4x4"""
    from scipy.spatial.transform import Rotation as R
    T = np.eye(4)
    T[:3, 3] = [pose.position.x, pose.position.y, pose.position.z]
    T[:3, :3] = R.from_quat(
        [pose.orientation.x, pose.orientation.y,
         pose.orientation.z, pose.orientation.w]
    ).as_matrix()
    return T

def _generate_initial_guess(self, path):
    """幾何経路から状態・制御初期推定値を生成"""
    # 簡略化：直線補間 + 逆運動学
    qs = np.array([self._dynamics.ik(p) for p in path])

```

```

        xs = np.array([self._dynamics.q2x(q) for q in qs])
        us = np.diff(xs, axis=0) / self._dynamics.dt
        return xs, us

def _build_constraints(self, mode):
    """接触・衝突回避制約を構築"""
    return {
        "contact": mode,
        "collision": self._dynamics.collision_checker,
    }

def _publish(self, plan: LocalPlan) -> None:
    """最適軌道をJointTrajectoryとして送信"""
    traj = JointTrajectory()
    traj.header = Header(stamp=self.get_clock().now().to_msg())
    traj.joint_names = self._dynamics.joint_names
    for k, x in enumerate(plan.states):
        pt = JointTrajectoryPoint()
        pt.positions = x[: self._dynamics.nq].tolist()
        pt.velocities = x[self._dynamics.nq:].tolist()
        pt.time_from_start.sec = int(k * self._dynamics.dt)
        traj.points.append(pt)
    self._traj_pub.publish(traj)

@rate_limit
def _wait_execution(self, plan: LocalPlan) -> None:
    """実行者が完了するまでスピン（簡略化）"""
    # 実機ではフィードバックトリックを監視
    pass

# ----- #
# メイン
# ----- #
def main(args=None):
    rclpy.init(args=args)
    node = TaskPlannerNode()
    executor = MultiThreadedExecutor()
    executor.add_node(node)
    try:

```

```

        executor.spin()
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == "__main__":
    main()

```

設計選択は性能と安全性に影響する。

- 最適化ホライズンと離散化は計算コストとタスク忠実度のトレードオフ。
- 積極的凸化は実行を高速化するが、非実行可能または危険な制約ぎりぎりの解を招くリスク。
- 学習ベースサンブラは成功率を向上させるが、環境が学習時と異なると分布外リスクを誘発。

運用エンジニアリングの影響、トレードオフ、リスク：

- 分解プランニングは計算負荷を削減するが、堅牢なシーケンシングを要し、サブタスクの失敗は連鎖する。
- ウォームスタートとモーションライブラリはランタイムを下げるがバイアスを蓄積する；エッジケースタスクのカバレッジを確保する。
- 緩和接触モデルを用いると最適化が簡単になるが、真の接触不安定性を隠蔽する可能性がある。ハードウェア実行前に高精度シミュレーションで最適化軌道を検証する。
- 並列化プランニングと GPU 加速によりヒューマノイドのニアリアルタイムリプランニングが可能になるが、システム複雑度と故障モードが増加する。ウォッチドッグと保守的フォールバックコントローラを実装してリスクを軽減する。

26.4 ヒューマノイド制御におけるよくある落とし穴

前節では、プランナーが複雑なタスクや予期しない障害物への適応にスケールする際に、計算量とホライズン長のトレードオフをどう扱うかを示した。これらの制約は、ヒューマノイドシステムの安定性、安全性、あるいは展開可能性を損なう繰り返し起こる制御上の落とし穴を直接露呈させる。

ヒューマノイド制御の落とし穴は、通常、モデル・センシング・計算の仮定と物理的な駆動が交差する箇所が発生する。以下に最も深刻なエラーのクラス、それぞれについての簡潔な技術的分析、実用的な緩和パターン、産業用あるいは実地ロボットに適した実装ノートを示す。

- モデルミスマッチと未モデル化ダイナミクス。
 - 問題の記述：コントローラはしばしば公称の質量・慣性・接触モデルを仮定する。実際のロボットにはアクチュエータ摩擦、伝達コンプライアンス、ペイロード変動がある。
 - 技術的分析：逆ダイナミクスあるいはモデル予測コントローラは $M(q)\ddot{q} + h(q, \dot{q}) = S^T \tau + J_c^T f_c$ を用いる。 M あるいは h の誤差は計算トルクにバイアスを生じ、遅延下で定常

偏差あるいは不安定性を生む。

- 緩和：高精度モデルとオンライン推定を組み合わせる。適応項あるいは外乱オブザーバを用いて Δh を推定・除去する。正確なモデルへの依存を、モデルベース指令と頑強なフィードバックを混在させることで制限する。

- トルクと帯域制限。

- 問題の記述：計画されたトルクがアクチュエータ能力を超えるか、コントローラが非現実的な帯域を仮定する。
- 技術的分析：飽和はインテグレータwindアップと非線形性を誘発する。指令トルクが制限を超えれば、接触反力と ZMP 予測は無効になる。
- 緩和：最適化器内でトルク制限を課し、制約にアクチュエータダイナミクスを含める。QP をウォームスタートして応答性を維持。最終安全レイヤで必ず指令をクリップ： $\tau_{\text{cmd}} = \text{clip}(\tau_{\text{cmd}}, -\tau_{\text{max}}, \tau_{\text{max}})$ 。

- 不適切に定式化された最適化問題。

- 問題の記述：コスト重み、制約、離散接触スケジュールが非実現あるいは悪条件 QP を生む。
- 技術的分析：一般的凸 QP は

$$\min_x \frac{1}{2} x^T H x + g^T x \quad \text{subject to} \quad A x = b, C x \leq d. \quad (220)$$

H が近似的に特異であれば数値誤差が増大する。矛盾する等式制約で過剰に拘束すると非実現性が生じる。

- 緩和： H を正則化（小さな対角追加）、等式制約を安全ならソフトコストに緩和し、非実現性を早期に検出する健全性チェックを適用。実現可能なフォールバック挙動を用意する。

- 遅延と離散化誤差。

- 問題の記述：知覚・ソルバ遅延が指令を陳腐化させる。離散時間積分が状態遷移を誤推定する。
- 技術的分析：遅延 τ を持つ制御ループは事実上位相遅れでループを閉じる。高ゲインコントローラでは数ミリ秒の遅延でもバランスコントローラを不安定化できる。
- 緩和：予測遅延補償を組み込む。ダイナミクスに明示的遅延を含む MPC を用いるか、フィードバック帯域を下げ予測ホライズンを増やす。状態測定をタイムスタンプ付きで最適化前に補間する。

- 接触・摩擦モデリングの失敗。

- 問題の記述：接触を固定あるいは急激に切り替えると衝撃力あるいはチャタリングを生じる。
- 技術的分析：接触制約の切り替えは解多様体とヤコビアン J_c に不連続性を生む。QP ではバイナリ接触変数を近似する必要がある、さもなければ混合整数問題になる。
- 緩和：接触相をスケジュールしたハイブリッド計画、あるいは相補近似による緩和を用いる。時間ベースのブレンディングとコンプライアンスモデリングで接触遷移を滑らかにする。

- センサノイズと状態推定障害。

- 問題の記述：ノイズの多い IMU、遅延した関節エンコーダ、あるいはドロップしたパケットが状態推定を劣化させる。
- 技術的分析：誤ったベース姿勢推定は ZMP と運動量計算をずらし、誤った補正動作を生む。推定器の発散は壊滅的な制御出力を引き起こす。

- 緩和：外乱除去を備えた頑強なセンサフュージョン (EKF/UKF)、共分散チューニング、障害検出。推定品質が劣化してもコントローラが優雅に劣化するように設計する。
- シミュレーションへの過適合。
 - 問題の記述：理想的シミュレータでチューニングされたコントローラが、未モデル化散逸あるいは遅延のために実機で失敗する。
 - 技術的分析：シミュレータの人工物を悪用したポリシーあるいはゲインは、正確な運動学あるいは摩擦係数に依存する場合がある。
 - 緩和：訓練・チューニング中に現実的なノイズ、遅延、アクチュエータモデルを注入する。実地配備前にハードウェアインザループで検証する。

実装チェックリスト (短縮版)：

1. QP ヘシアンを正則化しソルバをウォームスタートする。
2. アクチュエータ制限をハード制約として課し、その後コマンドクリップを適用する。
3. 未モデル化ダイナミクス用の外乱オブザーバを追加する。
4. センサデータにタイムスタンプを付け、遅延を補償するよう状態を予測する。
5. 安全上重要だがしばしば矛盾する目的にはソフト制約を用いる。

これらの緩和を示すコンパクトなコントローラループを以下に示す。QP セットアップ、正則化、遅延補償、ソルバ状態チェック、トルククリップを例示する。

コードサンプル 92 実用的保護を備えたシンプルな QP ベース逆ダイナミクスコントローラループ

```
import numpy as np
import osqp
from typing import Optional, Tuple

class QPController:
    """MPCレベルで関節トルクをQP最適化するプロダクションコントローラ"""
    def __init__(self,
                 n_j: int,
                 tau_max: np.ndarray,
                 solver_eps_abs: float = 1e-4,
                 solver_eps_rel: float = 1e-4,
                 max_iter: int = 4000):
        self.n_j = n_j
        self.tau_max = tau_max
        self.solver = osqp.OSQP()
        self.solver.settings.update(eps_abs=solver_eps_abs,
                                    eps_rel=solver_eps_rel,
                                    max_iter=max_iter,
                                    warm_start=True,
                                    verbose=False)
```

```

self._last_valid_tau = np.zeros(n_j)

def update(self,
            cost_hess: np.ndarray,
            cost_grad: np.ndarray,
            A_eq: Optional[np.ndarray],
            b_eq: Optional[np.ndarray],
            C_ineq: Optional[np.ndarray],
            d_ineq: Optional[np.ndarray]) -> np.ndarray:
    """QPを組み立て・解き、安全なトルクを返す"""
    # 正定化
    H = cost_hess + 1e-6 * np.eye(self.n_j)

    # 等式・不等式を結合
    if A_eq is not None and b_eq is not None:
        A = np.vstack([A_eq, C_ineq]) if C_ineq is not None else A_eq
        l = np.hstack([b_eq, -np.inf * np.ones(d_ineq.shape)]) if C_ineq is not None else b_eq
        u = np.hstack([b_eq, d_ineq]) if C_ineq is not None else b_eq
    else:
        A = C_ineq
        l = -np.inf * np.ones(d_ineq.shape)
        u = d_ineq

    # OSQPセットアップ（行列疎化済みと仮定）
    self.solver.setup(P=H, q=cost_grad, A=A, l=l, u=u, warm_start=True)
    res = self.solver.solve()

    if res.info.status_val != self.solver.constant('OSQP_SOLVED'):
        tau_cmd = self._last_valid_tau.copy()
    else:
        tau_cmd = res.x[:self.n_j]
        self._last_valid_tau = tau_cmd.copy()

    # 遅延補償（簡易版）
    tau_cmd = self._latency_compensate(tau_cmd)

    # 安全クリップ
    return np.clip(tau_cmd, -self.tau_max, self.tau_max)

def _latency_compensate(self, tau: np.ndarray) -> np.ndarray:

```

```

# 実機では測定遅延分の状態予測を行う
return tau

# 使用例
controller = QPController(n_j=7, tau_max=np.array([100.0]*7))

# 各時刻で呼ぶ
tau_safe = controller.update(build_Hessian(cost_terms),
                             build_gradient(task_errors),
                             *build_equality_constraints(),
                             *build_inequality_constraints())

send_to_actuators(tau_safe)

```

運用上の含意とトレードオフ：

- より厳密な性能はより正確なモデルを要求し、再較正が必要な脆いシステムを生む。
- 保守的設計は頑強性を高めるが、敏捷性と効率を低下させる。
- ソルバ選択と正則化はリアルタイム確実性とび制御周波数に影響する。
- これらの落とし穴に対処しなければ、転倒、ハードウェア損傷、安全でない相互作用のリスクが増大する。

設計チームは、層状の安全、現実的シミュレーション、オンライン適応、厳格な推定器検証を優先し、これらの一般的なヒューマノイド制御障害を削減しなければならない。

高度な振る舞い設計

27 ビヘイビアツリーの拡張

27.1 複雑なツリー構造

前述のビヘイビアツリーの基礎とブラックボードパターンに従い、ここではヒューマノイドロボットにツリーをスケールさせるための構造的技法を検討する。これらの技法は、並行モーター制御、知覚駆動型タスク切り替え、競合アクチュエータ間の安全な仲裁に対処する。

複雑なツリー設計は、明確な問題提起から始まる：複数の高帯域サブシステム（移動、操縦、視覚、バランス）を調整しながら、応答性と安全性を保持する。エンジニアは各サブシステムをモジュール型サブツリーとしてモデル化し、タイミング、リソースロック、グレースフルデグラデーションを強制する制御ノードを用いてこれらのサブツリーを結合しなければならない。

技術的分析：

- ノード構成とモジュール性：
 1. 各サブシステムを小さなブラックボード API を持つ自己完結型サブツリーとして設計する。センサ前処理、状態推定、アクチュエータコマンドをサブツリー内にカプセル化する。

2. サブツリーをパラメータ化することで再利用を促進する。ロジックを複製するのではなく、引数化されたライブラリノードを使用する。
- 並列性と同期：
 1. 胴体安定化や物体追跡などの同時タスクには並列ノードを使用する。低レベルコントローラが適切なレートで動作することを確保する。
 2. タスクがアクチュエータを共有する場合は同期バリアを挿入する。ブラックボード上に軽量リソースアービタを実装し、競合アクションを直列化する。
 - リアクティブ対デリバティブ構成：
 1. 転倒検出や関節限界モニタなどのリアクティブ安全チェックを、デコレータノード経由で高優先度に配置し、低優先度サブツリーをプリエンブトする。
 2. 長いホライズンの計画を、リアクティブデコレータに中断可能な効用またはシーケンスサブツリー内にカプセル化する。

スケーリングと信頼性を評価するのに数学式が役立つ。レベル i の分岐係数を b_i 、ツリー深さを d とする。ティックごとのノード訪問数の保守的上界は

$$[H]N_{\max} \leq \sum_{i=0}^d \prod_{j=0}^i b_j, \quad (221)$$

であり、分岐係数が 1 を超えると指数的に増大する。実用的なヒューマノイドシステムでは、最悪ケースのリアルタイムオーバランを防ぐため分岐を制限する。

独立した葉の成功確率 p_k を持つフォールバックノード（セレクトア）の場合、セレクトア全体の成功確率は

$$[H]P_{\text{success}} = 1 - \prod_{k=1}^n (1 - p_k). \quad (222)$$

となり、これは冗長戦略（例：複数アプローチによる把持）の利点を定量化する。

実装指針（実用的でテスト済みパターン）：

- リソース仲裁：ブラックボード管理ロックテーブルを実装する。アクチュエータコマンドの前にロックを取得し、完了後に解放する。
- 動的サブツリー交換：ペイロードや地形が変化した際に、実行時に代替安定化戦略をロードする。
- ヘルスデコレータ：長時間タスクにハートビートチェックとウォッチドッグタイマを添付する。

例：Python による動的サブツリーローダ。以下のスニペットは、実行時に新しい操縦戦略をロードして接続する例である。ビヘイビアツリーライブラリを想定し、ノードは Python オブジェクトとして表現され、中央のツルートを持つ。コメントは簡潔で実用的。

コードサンプル 93 操縦代替戦略の実行時サブツリー交換

```
import json
import logging
import time
from pathlib import Path
from typing import Any, Dict, Optional, Tuple
```

```

import rclpy
from rclpy.node import Node
from rclpy.executors import MultiThreadedExecutor
from std_msgs.msg import Float32
from ament_index_python import get_package_share_directory

# ロギング設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class BehaviorTreeNode:
    """ROS2対応のBehaviorTreeノードラッパー"""
    def __init__(self, node_name: str):
        self.ros_node = Node(node_name)
        self.blackboard = Blackboard()
        self.root = None

    def initialize(self):
        """ROS2ノード初期化"""
        self.mass_sub = self.ros_node.create_subscription(
            Float32, '/detected_payload_mass', self.mass_callback, 10)
        self.strategy_pub = self.ros_node.create_publisher(
            Float32, '/current_strategy', 10)

    def mass_callback(self, msg: Float32):
        """ペイロード質量変更時のコールバック"""
        self.adapt_manipulation_strategy(self.root, msg.data)

    def spin(self):
        """ROS2スピンループ"""
        executor = MultiThreadedExecutor()
        executor.add_node(self.ros_node)
        executor.spin()

class Blackboard:
    """スレッドセーフなブラックボード実装"""
    def __init__(self):

```

```

        self.data: Dict[str, Any] = {}
        self.arbiter = ActuatorArbiter()

    def get(self, key: str, default=None):
        return self.data.get(key, default)

    def set(self, key: str, value: Any):
        self.data[key] = value

class ActuatorArbiter:
    """アクチュエータアービタ"""
    def __init__(self):
        self._locks: Dict[str, bool] = {}

    def lock_for(self, actuators: list):
        """アクチュエータロックコンテキストマネージャー"""
        return ActuatorLock(self, actuators)

class ActuatorLock:
    """アクチュエータロック実装"""
    def __init__(self, arbiter: ActuatorArbiter, actuators: list):
        self.arbiter = arbiter
        self.actuators = actuators

    def __enter__(self):
        for actuator in self.actuators:
            if self.arbiter._locks.get(actuator, False):
                raise RuntimeError(f"アクチュエータ_{actuator}は既にロックされています")
            self.arbiter._locks[actuator] = True

    def __exit__(self, exc_type, exc_val, exc_tb):
        for actuator in self.actuators:
            self.arbiter._locks[actuator] = False

def load_tree(path: str) -> Any:
    """JSONファイルからサブツリーをロード"""
    try:

```

```

        full_path = Path(get_package_share_directory('manipulation_strategy')) / path
        with open(full_path, 'r', encoding='utf-8') as f:
            data = json.load(f)
            return parse_bt_json(data)
    except FileNotFoundError:
        logger.error(f"ファイルが見つかりません: {path}")
        raise
    except json.JSONDecodeError as e:
        logger.error(f"JSON解析エラー: {e}")
        raise

def find_parent_and_index(root: Any, target_name: str) -> Tuple[Optional[Any], int]:
    """ターゲットノードの親とインデックスを検索"""
    def _search(node: Any, parent: Any, index: int) -> Tuple[Optional[Any], int]:
        if hasattr(node, 'name') and node.name == target_name:
            return parent, index
        if hasattr(node, 'children'):
            for i, child in enumerate(node.children):
                result = _search(child, node, i)
                if result[0] is not None:
                    return result
        return None, -1

    return _search(root, None, -1)

def swap_subtree(root: Any, target_name: str, new_subtree: Any) -> None:
    """ターゲットノードを新しいサブツリーと交換"""
    parent, index = find_parent_and_index(root, target_name)
    if parent is None:
        raise RuntimeError(f"ターゲットノード '{target_name}' が見つかりません")

    # アクチュエータロック取得
    with root.blackboard.arbiter.lock_for(new_subtree.required_actuators):
        parent.children[index] = new_subtree
        new_subtree.initialize()
        logger.info(f"サブツリー '{target_name}' を更新しました")

```

```

def adapt_manipulation_strategy(root: Any, mass_kg: float) -> None:
    """ペイロード質量に基づいて戦略を適応"""
    if mass_kg > 3.0:
        subtree = load_tree('strategies/heavy_grasp.json')
        logger.info("重量物戦略をロード")
    else:
        subtree = load_tree('strategies/light_grasp.json')
        logger.info("軽量物戦略をロード")

    swap_subtree(root, 'manipulation_subtree', subtree)

def sensor_update(blackboard: Blackboard) -> None:
    """センサデータ更新（実装に応じてカスタマイズ）"""
    # 実際のセンサ読み取り処理を実装
    pass

def main():
    """メイン関数"""
    rclpy.init()

    bt_node = BehaviorTreeNode('manipulation_strategy_node')
    bt_node.initialize()

    # 初期ツリー構築
    bt_node.root = load_tree('trees/main_tree.json')

    # 別スレッドでROS 2スピン
    import threading
    ros_thread = threading.Thread(target=bt_node.spin)
    ros_thread.start()

    # メインループ
    try:
        rate = bt_node.ros_node.create_rate(100) # 100 Hz
        while rclpy.ok():
            sensor_update(bt_node.blackboard)
            bt_node.root.tick()
            rate.sleep()
    
```

```

except KeyboardInterrupt:
    pass
finally:
    bt_node.ros_node.destroy_node()
    rclpy.shutdown()
    ros_thread.join()

if __name__ == '__main__':
    main()

```

設計パターンとトレードオフ：

- ・階層ネスティングはグローバル複雑さを削減するが、深さ d を増加させ、式 (221) 経由で最悪ケースティックコストに影響する。
- ・並列ノードは応答性を改善する。アクチュエータ競合のリスクがある；明示的ロックプロトコルで軽減する。
- ・動的サブツローディングは適応ビヘイビアをサポートするが、一時的な不整合を引き起こす可能性がある。初期化と状態転送ルーチンを使用してコントローラ連続性を維持する。

運用上の考慮事項とリスク：

- ・リアルタイム保証：ティックごとのノード評価数を境界付けることでデッドラインミスを防止する。最大サブツリー実行時間を強制し、重いタスクには増分計算を優先する。
- ・安全性と中断：リアクティブデコレータは安全にプリエンプトし部分的コマンドを元に戻せる必要がある。モーターコマンドには常にアボートハンドラを設計する。
- ・検証：統合前に各サブツリーをシミュレーションで単体テストする。滑りや極端なペイロードなどエッジケースを実行するために物理インザループシミュレーションを使用する。
- ・保守性：過度の動的構成はシステムビヘイビアを不明瞭にする可能性がある。明確な命名、簡潔なブラックボードスキーマ、文書化されたリソース所有権を優先する。

具体的なエンジニアリングへの影響：

- ・サブツリーごとに CPU と通信予算を割り当てる。最悪ケースティックコストを経験的に測定する。
- ・サブツリー間で厳格なアクチュエータ所有権ルールを強制し、危険な同時トルクを回避する。
- ・劣化センサや部分的ハードウェア故障に対するリカバリサブツリーを計画し、グレースフルデグラデーションを維持する。

27.2 意思決定へのハイブリッドアプローチ

前の小節では、ヒューマノイド動作のための複雑なツリートポロジとモジュラ構成を検討した。それらの構造化パターンを基に、本小節ではビヘイビアツリーを補完的な意思決定層と組み合わせ、現実世界のヒューマノイド要求を満たす方法を示す。

ハイブリッドアプローチはビヘイビアツリー（BT）を計画、学習、形式的安全性コンポーネントと統合し、それらの補完的な強みを活用する。BT は明確な階層的順序と反応性を提供する。モデル予測制御（MPC）と全身コントローラは動的で制約を意識した動作を供給する。強化学習（RL）または学習済みポリシーは知覚が豊富な文脈での適応を提供する。形式モニタまたはオフライン POMDP 解は不確実性下での確率的推論を提供する。ヒューマノイドがバランスを保ち、安全性制約を尊重し、効率的にタスクを達成するために、これらのモジュールを仲裁することが工学上の課題である。

問題定義

- ・センサ状態 s とタスク目標 g が与えられたとき、BT リーフ、RL ポリシ、または動作プランナが生成した候補から制御行動 a を選択する。
- ・目的：タスク効用を最大化し、安全性制約を満たし、発振を防ぐために切替周波数を制限する。

技術解析

- ・各決定ソースを効用推定器 $u_i(a, s)$ と信頼度または有効性マスク $v_i(s) \in \{0, 1\}$ として表現する。
- ・切替ペナルティを伴う重み付き仲裁を用いて選択を安定化する。選択された行動は

$$[H]a^* = \arg \max_{a \in \mathcal{A}} \sum_i w_i v_i(s) u_i(a, s) - \lambda c(a, a_{\text{prev}}), \quad (223)$$

ここで w_i はモジュール重み、 λ はトレードオフパラメータ、 $c(\cdot, \cdot)$ は切替をペナルティする。典型的な c は二値： $a \neq a_{\text{prev}}$ なら $c(a, a_{\text{prev}}) = 1$ 。

- ・安全性は監視述語 $S(s, a) \in \{0, 1\}$ によって施行される。行動は $S(s, a) = 1$ の場合のみ実行可能。ハード安全性では、仲裁器は

$$[H]a^* = \arg \max_{a \in \mathcal{A}_S} (\text{式 223}), \quad \mathcal{A}_S \triangleq \{a \mid S(s, a) = 1\}. \quad (224)$$

を解く。

ヒューマノイドのための実装パターン

1. モジユラ出力：

- ・BT は離散的なタスクレベル行動を提供： $\{\text{approach}, \text{object}, \text{grasp}, \text{step}\}$ 。
- ・RL ポリシは視覚的曖昧さが高いときに操作作用の連続的な全身コマンドを生成する。
- ・MPC/WBC は零力矩点（ZMP）と関節限界を尊重したトルクレベル軌道を計算する。
- ・安全性モニタは運動学的衝突、関節トルク境界、転倒リスク指標をチェックする。

2. 仲裁ランタイムループ：

- ・センサをサンプリングし、制御レートで状態 s を推定する。
- ・各モジュールに候補行動と効用スコアを問い合わせる。
- ・式 223 を安全性フィルタ式 224 と共に適用する。
- ・選択された行動をコントローラに送信し、 a_{prev} を記録する。

3. 実用上の考慮事項：

- ・共通の決定周波数（例：50 Hz）で同期する。重いプランナは非同期で実行し、更新されるまで最後の有効な計画を使用する。
- ・効用にヒステリシスまたはリーキング平均を用いてチャタリングを回避する。
- ・行動を統一的に表現（例：ホライズン上の所望関節軌道）して安全性チェックを簡素化する。

例：到達・回復シナリオ

- BT リーフ「reach」はエンドエフェクタ軌道を要求する。知覚信頼度が低い場合、RL は学習済みビジュアルサーボポリシーを提供する。MPC は全身バランス補正を供給する。
- 押し擾乱中は仲裁重みが MPC にシフトする。切替ペナルティは過渡的知覚更新中にコントローラ連続性を保持する。

設計選択とパラメータ選択

- 重み選択 w_i は権限配分を制御する。安定性が重要な場合は $w_{\text{MPC}} > w_{\text{RL}}$ を選ぶ。
- ペナルティ λ は応答性と安定性をトレードする。大きい λ は切替を減らすが適応を遅延させる可能性がある。
- 有効性マスク v_i は学習可能、例：センサ共変量からモジュール信頼性を予測する小分類器。

統合例：簡潔な仲裁器実装。このリストは 3 つのモジュールを問い合わせ、選択された行動識別子を返す GROOT 互換カスタムノードである。

コードサンプル 94 ハイブリッド仲裁器ノード：モジュールを問い合わせ、効用を計算、安全性を施行。

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Int8, Float32MultiArray
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from typing import Dict, Tuple, List, Optional, Any
import numpy as np
from dataclasses import dataclass
import threading
import time

@dataclass
class ActionCandidate:
    action: np.ndarray
    source: str
    utility: float
    valid: bool

class HybridArbiterNode(Node):
    def __init__(self, params: Dict[str, Any]) -> None:
        super().__init__('hybrid_arbiter', namespace=params.get('namespace', ''))
        self.declare_parameters(
            namespace='',
```

```

        parameters=[
            ('weights.bt', 1.0),
            ('weights.rl', 0.9),
            ('weights.mpc', 1.2),
            ('lambda_switch', 0.5),
            ('timeout_sec', 0.1),
            ('rate_hz', 50),
        ]
    )
    self.weights: Dict[str, float] = {
        'bt': self.get_parameter('weights.bt').value,
        'rl': self.get_parameter('weights.rl').value,
        'mpc': self.get_parameter('weights.mpc').value,
    }
    self.lambda_switch: float = self.get_parameter('lambda_switch').value
    self.timeout_sec: float = self.get_parameter('timeout_sec').value
    self.rate_hz: int = self.get_parameter('rate_hz').value

    self.prev_action: Optional[np.ndarray] = None
    self.state_lock = threading.Lock()
    self.latest_state: Optional[Dict[str, np.ndarray]] = None

    qos = QoSProfile(
        reliability=ReliabilityPolicy.BEST_EFFORT,
        history=HistoryPolicy.KEEP_LAST,
        depth=1
    )

    self.cmd_pub = self.create_publisher(Twist, 'cmd_vel_out', qos)
    self.state_sub = self.create_subscription(
        Odometry, 'odom_in', self._state_cb, qos)
    self.timer = self.create_timer(1.0 / self.rate_hz, self.tick)

    def _state_cb(self, msg: Odometry) -> None:
        with self.state_lock:
            self.latest_state = {
                'pose': np.array([
                    msg.pose.pose.position.x,
                    msg.pose.pose.position.y,
                    msg.pose.pose.position.z,

```

```

        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w,
    ]),
    'twist': np.array([
        msg.twist.twist.linear.x,
        msg.twist.twist.linear.y,
        msg.twist.twist.linear.z,
        msg.twist.twist.angular.x,
        msg.twist.twist.angular.y,
        msg.twist.twist.angular.z,
    ]),
}

```

```

def query_bt_leaf(self, state: Dict[str, np.ndarray]) -> Tuple[np.ndarray, float,
# 実装は実際のBTノードに置き換える
    return np.array([0.0, 0.0]), 0.8, True

```

```

def query_rl_policy(self, state: Dict[str, np.ndarray]) -> Tuple[np.ndarray, float,
# 実装は実際のRLポリシーに置き換える
    return np.array([0.1, 0.0]), 0.7, True

```

```

def query_mpc(self, state: Dict[str, np.ndarray]) -> Tuple[np.ndarray, float, bool,
# 実装は実際のMPCに置き換える
    return np.array([0.05, 0.0]), 0.9, True

```

```

def safety_check(self, state: Dict[str, np.ndarray], action: np.ndarray) -> bool:
# 簡易衝突判定
    return np.linalg.norm(action) < 2.0

```

```

def tick(self) -> None:
    with self.state_lock:
        if self.latest_state is None:
            return
        state = self.latest_state.copy()

    candidates: List[ActionCandidate] = []
    a_bt, u_bt, v_bt = self.query_bt_leaf(state)
    if v_bt:

```

```

        candidates.append(ActionCandidate(a_bt, 'bt', u_bt, v_bt))
a_rl, u_rl, v_rl = self.query_rl_policy(state)
if v_rl:
    candidates.append(ActionCandidate(a_rl, 'rl', u_rl, v_rl))
a_mpc, u_mpc, v_mpc = self.query_mpc(state)
if v_mpc:
    candidates.append(ActionCandidate(a_mpc, 'mpc', u_mpc, v_mpc))

safe = [c for c in candidates if self.safety_check(state, c.action)]
if not safe:
    self.get_logger().warn('No safe action')
    return

def score(c: ActionCandidate) -> float:
    w = self.weights[c.source]
    switch = 0.0 if np.array_equal(self.prev_action, c.action) else 1.0
    return w * c.utility - self.lambda_switch * switch

chosen = max(safe, key=score)
self.prev_action = chosen.action

twist = Twist()
twist.linear.x = float(chosen.action[0])
twist.angular.z = float(chosen.action[1])
self.cmd_pub.publish(twist)

def main(args=None):
    rclpy.init(args=args)
    node = HybridArbiterNode({})
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

運用上の影響、トレードオフ、リスク

- 利点：ハイブリッド仲裁は BT の可読性を保持しながら学習と最適制御の強みを活用する。
- トレードオフ：
 - 複雑さが増加する；モジュール間相互作用のデバッグが困難になる。
 - 複数の並列モジュールにより計算負荷が増加する可能性がある。
 - 重みチューニングと切替ペナルティはハードウェアでの実証的検証を要する。
- リスク：
 - 不適切に較正された仲裁は安定性コントローラを上書きし、転倒を引き起こす可能性がある。
 - 非同期モジュール故障は古い行動を生じる；タイムアウトと有効性チェックを使用する。
 - 安全性モニタは独立し、論理的に検証可能で、ハードオーバーライド能力を持つ必要がある。

エンジニアは安全性最優先アーキテクチャを優先すべき：形式的安全性モニタが仲裁コマンドを拒否または置換できることを確認する。ハードウェア試験前に、段階的により現実的なシミュレーションで仲裁ロジックを検証する。

27.3 ケーススタディ：マルチタスクロボット

前述のハイブリッド協調手法と複雑なツリーテンプレートは、実用的なアーキテクチャを促す。ここでは、スケジューラ駆動のアロケータと並列およびデコレータノードを組み合わせることでタスクを安全に実行、プリエンプト、シーケンスするヒューマノイドマルチタスクシステムを実装し、分析する。

問題定義。病院支援ヒューマノイドは、配達、患者モニタリング、緊急対応を同時に実行しなければならない。タスクはアクチュエーション、知覚パイプライン、バッテリー電力を競合する。システムは、安全制約、タイミング境界、障害からの回復可能性を保証しながらミッション効用を最大化しなければならない。

技術分析。各候補タスク i をタプル (u_i, c_i, t_i, r_i) で表現する。ここで、 u_i は効用、 c_i はエネルギーコスト、 t_i は予想実行時間、 r_i は必要なリソースセットである。リソースセットは相互に排他的なアクチュエータまたは高帯域幅の知覚チャネルをモデル化する。安全制約は、タスク実行中に保持されなければならない不変条件 S_k にマップされる。各決定エポックで制約付き割当問題を定式化する：

$$\begin{aligned} [H] \quad & \max_{x_i \in \{0,1\}} \sum_i u_i x_i \\ & \text{s.t.} \quad \sum_i c_i x_i \leq B \quad (\text{battery budget}) \\ & \sum_{i: a \in r_i} x_i \leq 1 \quad \forall \text{ actuator } a \\ & x_i = 0 \quad \text{if } \neg S_k \text{ for any required safety } S_k. \end{aligned} \tag{225}$$

式 (1) は追加のパッキング制約を持つ 0/1 ナップサックである。CPU の可用性に応じて貪欲ヒューリスティックまたは整数計画で近似解を求める。実行不確実性と知覚センサノイズを考慮するためにモデル予測更新を使用する。

ビヘイビアツリー統合。BT は依然として主要な実行基盤である。ツリーを以下で拡張する：

- トップレベルの Parallel ノードが実行する：

1. アクティブタスクセット X^* を計算する Task Allocator リーフ。
 2. 安全不変条件を強制する Supervisor サブツリー。
 3. アクティベーションデコレータで守られた Task Subtrees のプール。
- アクティベーションデコレータは Allocation 出力を購読し、対応する $x_i = 1$ のときに子サブツリーを有効にする。デコレータはグレースフルプリエンプトを実装：タスク固有の完了ポリシーに基づいて子に終了または中止を通知する。

実装スケッチ。TaskAllocator は高速リアクティブサービスとして機能する。バッテリー状態、キュー優先度、センサから派生した緊急フラグを取り込む。高優先度の緊急事象が現れると、アロケータは他の非クリティカルタスクを強制中止し、緊急事象のためにリソースを予約する。

以下の例コードは、`py_trees` ライクな API を使用した最小 Python 実装を示す。アロケータは速度のための単純な貪欲

コードサンプル 95 Minimal Task Allocator and BT wiring for a humanoid multi-task system.

```
import py_trees as pt
import rclpy
from rclpy.node import Node
from rcl_interfaces.msg import SetParametersResult
from typing import Dict, Any, Callable, Set, Optional
import numpy as np

class TaskAllocator(pt.behaviour.Behaviour):
    """
    タスクの優先度とバッテリー残量を考慮し、実行タスク集合を決定する。
    ROS2 パラメータサーバ経由で割当結果を公開。
    """
    def __init__(
        self,
        node: Node,
        tasks: Dict[str, Dict[str, Any]],
        battery_provider: Callable[[], float],
        param_ns: str = "task_allocator"
    ):
        super().__init__(name="TaskAllocator")
        self.node = node
        self.tasks = tasks
        self.battery = battery_provider
        self.active: Set[str] = set()
        self._param_ns = param_ns
        self._declare_params()

    # パラメータ宣言
```

```

def _declare_params(self):
    self.node.declare_parameter(f"{self._param_ns}.emergency_preempt", True)
    self.node.declare_parameter(f"{self._param_ns}.safety_check", True)

# パラメータ取得
def _get_params(self):
    return {
        "emergency_preempt": self.node.get_parameter(f"{self._param_ns}.emergency_preempt"),
        "safety_check": self.node.get_parameter(f"{self._param_ns}.safety_check").value
    }

def update(self) -> pt.common.Status:
    B = self.battery()
    params = self._get_params()

    # 緊急タスクが存在すれば即座に選択
    if params["emergency_preempt"]:
        for tid, meta in self.tasks.items():
            if meta.get("emergency", False):
                self.active = {tid}
                self._publish_allocation()
                return pt.common.Status.SUCCESS

    # 効用/コスト比で降順ソート
    ordered = sorted(
        self.tasks.items(),
        key=lambda kv: kv[1]["u"] / max(kv[1]["c"], 1e-6),
        reverse=True
    )

    chosen: Set[str] = set()
    used_cost = 0.0
    used_res: Set[str] = set()

    for tid, meta in ordered:
        if used_cost + meta["c"] > B:
            continue
        if used_res & set(meta.get("r", [])):
            continue
        if params["safety_check"] and not meta.get("safety_ok", True):

```

```

        continue
    chosen.add(tid)
    used_cost += meta["c"]
    used_res |= set(meta.get("r", []))

self.active = chosen
self._publish_allocation()
return pt.common.Status.SUCCESS

# 割当結果をROS 2 パラメータとして公開
def _publish_allocation(self):
    self.node.set_parameters([
        rclpy.parameter.Parameter(
            f"{self._param_ns}.active_tasks",
            rclpy.Parameter.Type.STRING_ARRAY,
            list(self.active)
        )
    ])

class ActivationDecorator(pt.decorators.Decorator):
    """
    allocator.active_に含まれる場合のみ子を実行。
    含まれなくなった瞬間に子を中断 (cancel) する。
    """
    def __init__(
        self,
        task_id: str,
        allocator: TaskAllocator,
        child: pt.behaviour.Behaviour,
        name: Optional[str] = None
    ):
        name = name or f"Active_{task_id}"
        super().__init__(name=name, child=child)
        self.task_id = task_id
        self.allocator = allocator
        self._was_active = False

    def update(self) -> pt.common.Status:
        active = self.task_id in self.allocator.active

```

```

    if active:
        self._was_active = True
        return self.decorated.update()
    else:
        if self._was_active:
            # 一度実行していたら中断
            self.decorated.stop(pt.common.Status.INVALID)
            self._was_active = False
        return pt.common.Status.FAILURE

```

ノード初期化 & ツリー構築例

```

def build_tree(node: Node, tasks_dict: Dict[str, Any], battery_provider: Callable[[],
    allocator = TaskAllocator(node, tasks_dict, battery_provider)

    supervisor = build_supervisor_subtree(node)

    task_pool = pt.composites.Parallel(
        "TaskPool",
        policy=pt.common.ParallelPolicy.SuccessOnAll(False, False)
    )

    for tid, meta in tasks_dict.items():
        leaf = build_task_subtree(node, tid, meta)
        deco = ActivationDecorator(tid, allocator, leaf)
        task_pool.add_child(deco)

    root = pt.composites.Parallel(
        "Root",
        policy=pt.common.ParallelPolicy.SuccessOnAll(False, False)
    )
    root.add_children([allocator, supervisor, task_pool])
    return root

```

運用上の考慮事項と安全性。実装しなければならない重要なランタイム動作：

- プリエンプトポリシー：ハードウェアに損傷を与える可能性があるアクチュエータの中止レイテンシ境界を指定する。
- リソースロック：高電力モーターのための原子獲得を強制；順序付きリソース獲得でデッドロックを回避する。
- 不確実性処理：安全マージン σ でコスト見積もりを膨張して予算を保守的に保つ、例：

$c'_i = c_i(1 + \sigma)$ を使用する。

- ・検証：代表性のある BT 設定でオフラインモデル検査を実行し不変条件を検証する。

設計のトレードオフとリスク。貪欲割当は低レイテンシ決定を提供するが、大域的最適性を犠牲にする。整数計画はより良い計画を生むが、より高い計算コストでレイテンシと電力消費を増加させる。積極的なプリエンプトは緊急対応時間を短縮するが、頻繁なアクチュエータ起動により摩耗を増加させる可能性がある。効用またはコストの誤推定は、タスク選択を系統的に歪め、運用上の盲点を作り出すことができる。

具体的な工学上の影響。このアーキテクチャの実装には、アロケータの決定的タイミング保証、割当を上書きできる認証済み安全モニタ、タスク完了セマンティクスの慎重な選択が必要である。安全違反とリソース飢餓を回避するために、現実的なセンサノイズとタスク期間分散を持つシミュレーションで割当ポリシーを検証してからハードウェアに展開する。

28 協力とチームワーク

28.1 複数ロボットの協調

この小節では、これまでに導入された拡張決定構造とハイブリッド制御モチーフを基に、共有環境における複数ヒューマノイドの協調にそれらのアイデアを適用する。焦点は単一エージェントのビヘイビアビッツリーから、ヒューマノイドチーム間でフォーメーション、タスク割当、安全な相互作用を実現する分散プロトコルへと移る。

問題定義. 複数のヒューマノイドロボットは、バランス、運動学的到達可能性、動的相互作用力を尊重しながら全身動作を協調させなければならない。典型的なタスクには、大型物体の協調操縦、構造物の協調点検、人間支援による運搬が含まれる。協調は通信遅延、断続的センシング、異種作動能力に耐えなければならない。

技術解析. 効果的な協調は、知覚、状態推定、通信、決定仲裁、低次制御へ関心事を分離する。設計選択は集中、分散、ハイブリッドアーキテクチャに分類される：

- ・集中制御はグローバル計画合成を単純化するが、単一障害点と高帯域要求を生む。
- ・分散制御は頑健性とスケーラビリティを高めるが、衝突を回避するために合意または交渉を必要とする。
- ・ハイブリッド方式は粗い割当てにコーディネータを用い、微調整には局所コントローラを用いる。

主要プリミティブと制約：

- ・状態ブロードキャスト：各ヒューマノイドはベースポーズ、重心 (CoM)、接触状態、重要タスク変数を含むコンパクトな状態ベクトルを共有する。ブロードキャストには `/robot_state` トピックを使用。
- ・フォーメーション制御：ZMP (ゼロモーメントポイント) エンベロープから計算された安定性マージンに従い相対ポーズを維持する。
- ・タスク割当て：市場ベースオークションまたは契約ネットプロトコルを用いてサブタスクを割当

て、異種スキルを活用する。

- 安全エンベロープ：足とマニピュレータの周りに動的な安全ゾーンを定義し、衝突と相互失穩を防ぐ。

連続変数に対する合意は実用的な分散手法である。スカラー協調変数 x_i に対する離散時間合意はエージェント i で

$$[H]x_i^{k+1} = x_i^k + \alpha \sum_{j \in \mathcal{N}_i} (x_j^k - x_i^k), \quad (226)$$

に従い、ここで \mathcal{N}_i は近傍集合、 $\alpha \in (0, 1/\Delta)$ 、 Δ は最大ノード次数である。平均への収束には $\alpha < 1/\lambda_{\max}(L)$ が必要で、 L はグラフラプラシアンである。

フォーメーション維持のため、スカラー合意を SE(3) 平均または姿勢のためのデュアル四元数合意へ拡張する。人間が存在する場合、動的制約に違反せずにフォーメーションにバイアスをかけるため、人間の意図を合意項への高優先度入力として組み込む。

オークションによるタスク割当ては異種ヒューマノイドにとって実用的である。オークションループはロボット i とタスク t に対して効用 $u_{i,t}$ を計算する。効用には到達可能性、予想エネルギー消費、バランスリスク、推定完了時間が含まれる。グローバル割当ては最大重みマッチング問題を解き、リアルタイム制約を満たすために貪欲または分散アルゴリズムで近似する。

実装スケッチ. 以下の Python ベース ROS2 スタイルノードは、所望フォーメーションオフセットのための分散合意ループを示す。/desired_offset をパブリッシュし、/robot_state を購読する。コメントは簡潔で実用的である。

コードサンプル 96 フォーメーションオフセットのための分散合意ループ (ROS2 風擬似コード)

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
from std_msgs.msg import Float32
from geometry_msgs.msg import PoseStamped
import argparse
import sys

# QoS設定：信頼性重視でネットワーク負荷を抑制
QOS = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    history=HistoryPolicy.KEEP_LAST,
    depth=10,
    durability=DurabilityPolicy.VOLATILE
)

ALPHA = 0.1          # 合意ゲイン
PUBLISH_RATE = 10.0 # Hz
TIMEOUT_SEC = 1.0    # 隣接情報の有効期限[s]
```

```

class FormationConsensus(Node):
    def __init__(self, robot_id: int):
        super().__init__(f'formation_consensus_{robot_id}')
        self.robot_id = robot_id
        self.offset = 0.0
        self.neighbor_offsets = {} # neighbor_id -> (offset, stamp)

        self.pub = self.create_publisher(Float32, '/desired_offset', QOS)
        self.create_subscription(
            PoseStamped, '/robot_state', self.state_cb, QOS)
        self.timer = self.create_timer(1.0 / PUBLISH_RATE, self.loop_cb)

    def state_cb(self, msg: PoseStamped):
        # frame_idに隣接ID、pose.xにオフセットを格納
        try:
            nid = int(msg.header.frame_id)
        except ValueError:
            self.get_logger().warn('frame_id is not integer')
            return
        self.neighbor_offsets[nid] = (msg.pose.position.x,
                                       self.get_clock().now())

    def loop_cb(self):
        now = self.get_clock().now()
        # タイムアウトした隣接を除去
        self.neighbor_offsets = {
            k: (v, t) for k, (v, t) in self.neighbor_offsets.items()
            if (now - t).nanoseconds * 1e-9 < TIMEOUT_SEC
        }

        if not self.neighbor_offsets:
            self.get_logger().warn('no valid neighbors')
            self.pub.publish(Float32(data=float(self.offset)))
            return

        # 合意更新
        err = sum(v - self.offset for v, _ in self.neighbor_offsets.values())
        self.offset += ALPHA * err / len(self.neighbor_offsets)

```

```

        self.pub.publish(Float32(data=float(self.offset)))

def main(argv=sys.argv):
    rclpy.init(args=argv)
    parser = argparse.ArgumentParser()
    parser.add_argument('--robot-id', type=int, required=True)
    args, _ = parser.parse_known_args()

    node = FormationConsensus(args.robot_id)
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

ビヘイアビアツリーとの統合. 合意またはオークションノードを GROOT ビヘイアビアツリーのアクションまたはサービスノードとして組み込む。ブラックボードキーを用いて合意導出目標を歩行および操縦サブツリーに格納する。この分離により、高次プランナが協調目標を要求しながら低次コントローラがバランスとトルクリミットを強制することを可能にする。

エンジニアリングにおける意味とトレードオフ：

1. 通信：帯域幅を高くすると収束速度が向上するが、サービス拒否リスクとエネルギーコストが増大する。
2. 遅延とパケット損失：合意ゲインは保守的に設計する。スパースネットワークにはイベントトリガ更新を用いる。
3. 安全性：協調状態に関係なくバランスを強制する局所フォールバックを常に実行する。共有ペイロードには接触力センシングとインピーダンス制御が必要。
4. 複雑性対頑健性：集中最適化は高い最適性をもたらすが、回復力は低い。分散方式はよりスケールするが、慎重な調整が必要。

運用上のリスクには、複数ヒューマノイドが歩行位相を同期させる際の失穩化フィードバックループや、個体ロボットへの過負荷を引き起こすタスクの誤割当てが含まれる。Isaac Sim でグループダイナミクスを検証し、ゲインを調整し、ハードウェア展開前に通信障害をシミュレートするための設計テスト手順を作成する。

28.2 群知能の実装

前小節の分散状態共有やリーダー選出といった協調プリミティブを基に，ここでは数十体のヒューマノイドエージェントにスケールする分散アルゴリズムに焦点を当てる．目的は，ヒューマノイドの制約—バランス，全身運動学，限られた通信帯域，人に安全な動作—を尊重した群知能を実装することである．

問題定義．ヒューマノイドチームを展開し，協調マニピュレーションとエリアカバレッジを実行させる．各エージェントは以下を満たす必要がある：

- 移動中に動的安定性を維持；
- 他エージェントや人との衝突を回避；
- 局所通信のみでタスク割当とフォーメーションに合意；
- 故障やセンサノイズに頑健に反応．

技術分析．ヒューマノイドの群知能は，相補的な2層を組み合わせる必要がある：

1. 低レベル安定性・反射層．関節限界，零モーメント点（ZMP）制約，コンプライアンスを強制する．各ロボットで高周波で動作させる必要がある．
2. 高レベル分散協調層．コンセンサス，タスク割当，フォーメーション制御を近隣間メッセージングで処理する．

合意のためのコンセンサスは基本的なプリミティブである． x_i をエージェント i の共有変数（例：タスク優先度スコアやフォーメーション中心）の局所推定値とする．離散時間線形コンセンサス則は：

$$[H]x_i[k+1] = x_i[k] + \alpha \sum_{j \in \mathcal{N}_i} w_{ij} (x_j[k] - x_i[k]), \quad (227)$$

ここで \mathcal{N}_i は近隣集合， $w_{ij} \geq 0$ は重み， $0 < \alpha < 1/\lambda_{\max}(L)$ は収束を保証する．ここで L はグラフラプラシアンである．通信グラフが連結でステップサイズがスペクトル制約を満たせば収束が保証される．

フォーメーションと衝突回避は，ヒューマノイド安全エンベロップに適合した人工ポテンシャル場で処理する．エージェント位置を地面座標で $p_i \in \mathbb{R}^2$ と定義する．ペアワイズ分離とフォーメーション吸引ポテンシャルを用いる：

$$[H]U = \sum_i \sum_{j>i} \frac{k_r}{\|p_i - p_j\|^2} + \sum_i \frac{k_a}{2} \|p_i - p_i^{\text{ref}}\|^2, \quad (228)$$

反発係数 k_r ，吸引係数 k_a を含む．分散制御入力 $u_i = -\nabla_{p_i} U$ で，バランスを維持する実行可能な地面平面運動に射影する．

タスク割当は分散オークションやコンセンサスベース負荷分散を用いる．安全クリティカルタスクでは冗長性を持つ保守的割当を好む．役割遷移中の単一障害点を防ぐため，タイムドソフトハンドオフを実装する．

実装概要．主要なエンジニアリング要素：

- 局所状態ベクトル： $s_i = [p_i, v_i, x_i, \text{role}_i]$ ， p_i は姿勢， v_i 速度， x_i コンセンサス変数， role_i 現タスク．

- 近隣メッセージングは低遅延ミドルウェア（例：ROS2 DDS）を使用．メッセージは状態とタイムスタンプを運ぶ．
- オンボードフィルタ：姿勢と近隣相対推定に拡張カルマンフィルタを使い，コンセンサス入力前のノイズを削減．
- 安全監督：ZMP とトルクリミットを監視し，高レベルコマンドを拒否できる．

最小限の Python 風分散制御ループを以下に示す．局所コンセンサス，ポテンシャルベースフォーメーション制御，安全チェックを含む．転送は ros2 またはカスタム UDP で低遅延に置き換える．

コードサンプル 97 ヒューマノイドエージェント用分散コンセンサス・フォーメーション制御ループ

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
import numpy as np
from copy import deepcopy
from typing import List, Dict
from geometry_msgs.msg import Twist, Vector3, PoseStamped
from sensor_msgs.msg import JointState, Imu
from std_msgs.msg import Header
import tf2_ros
import tf2_geometry_msgs
from ament_index_python.packages import get_package_share_directory
import yaml
import os
import time
from threading import Lock

class ConsensusFormationNode(Node):
    def __init__(self):
        super().__init__('consensus_formation_node')

        # パラメータ読み込み
        self.declare_parameters(
            namespace='',
            parameters=[
                ('robot_id', 0),
                ('alpha', 0.1),
                ('k_r', 1.0),
                ('k_a', 0.5),
                ('control_dt', 0.01),
```

```

        ('max_velocity', 0.5),
        ('safety_threshold', 0.1),
        ('formation_radius', 2.0),
        ('neighbor_timeout', 0.5),
    ]
)

self.robot_id = self.get_parameter('robot_id').value
self.alpha = self.get_parameter('alpha').value
self.k_r = self.get_parameter('k_r').value
self.k_a = self.get_parameter('k_a').value
self.control_dt = self.get_parameter('control_dt').value
self.max_velocity = self.get_parameter('max_velocity').value
self.safety_threshold = self.get_parameter('safety_threshold').value
self.formation_radius = self.get_parameter('formation_radius').value
self.neighbor_timeout = self.get_parameter('neighbor_timeout').value

# QoS設定
qos_profile = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    history=HistoryPolicy.KEEP_LAST,
    depth=10
)

# パブリッシャー
self.cmd_vel_pub = self.create_publisher(Twist, 'cmd_vel', qos_profile)
self.state_pub = self.create_publisher(PoseStamped, 'robot_state', qos_profile)

# サブスクライバー
self.imu_sub = self.create_subscription(Imu, 'imu/data', self.imu_callback, qos_profile)
self.joint_sub = self.create_subscription(JointState, 'joint_states', self.joint_callback, qos_profile)
self.neighbor_sub = self.create_subscription(PoseStamped, 'neighbor_states', self.neighbor_callback, qos_profile)

# TF2バッファ
self.tf_buffer = tf2_ros.Buffer()
self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

# 状態変数
self.state_lock = Lock()
self.state = {

```

```

        'x': np.array([0.0, 0.0]),
        'p': np.array([0.0, 0.0]),
        'p_ref': np.array([0.0, 0.0]),
        'imu': None,
        'joints': None,
        'timestamp': time.time()
    }

    self.neighbors: Dict[int, Dict] = {}

    # タイマー
    self.control_timer = self.create_timer(self.control_dt, self.control_loop)
    self.state_publish_timer = self.create_timer(0.1, self.publish_state)

    self.get_logger().info(f'Robot_{self.robot_id}_consensus_formation_node_starte

def imu_callback(self, msg: Imu):
    with self.state_lock:
        self.state['imu'] = msg

def joint_callback(self, msg: JointState):
    with self.state_lock:
        self.state['joints'] = msg

def neighbor_callback(self, msg: PoseStamped):
    try:
        # ロボットIDをヘッダーから取得 (frame_idに埋め込む想定)
        neighbor_id = int(msg.header.frame_id.split('_')[-1])

        # 地面平面座標に変換
        p = np.array([msg.pose.position.x, msg.pose.position.y])

        with self.state_lock:
            self.neighbors[neighbor_id] = {
                'x': np.array([msg.pose.position.x, msg.pose.position.y]),
                'p': p,
                'timestamp': time.time()
            }

    except Exception as e:

```

```

        self.get_logger().warn(f'Failed to process neighbor state: {e}')

def sense_state(self) -> Dict:
    """現在の状態を取得"""
    with self.state_lock:
        return deepcopy(self.state)

def get_neighbors(self) -> List[Dict]:
    """有効な近隣ロボットを取得"""
    current_time = time.time()
    valid_neighbors = []

    with self.state_lock:
        for neighbor_id, neighbor_data in list(self.neighbors.items()):
            if current_time - neighbor_data['timestamp'] < self.neighbor_timeout:
                valid_neighbors.append(neighbor_data)
            else:
                # タイムアウトした近隣を削除
                del self.neighbors[neighbor_id]

    return valid_neighbors

def w(self, id1: int, id2: int) -> float:
    """重み関数：距離に基づく重み付け"""
    return 1.0 / (1.0 + abs(id1 - id2))

def consensus_update(self, state: Dict, neighbors: List[Dict]) -> np.ndarray:
    """コンセンサス更新：式(1)"""
    x = state['x'].copy()

    for nb in neighbors:
        # 重み付き平均で共有変数を更新
        x += self.alpha * self.w(self.robot_id, 0) * (nb['x'] - x)

    return x

def formation_force(self, state: Dict, neighbors: List[Dict]) -> np.ndarray:
    """フォーメーション力計算：式(2)"""
    force = np.zeros(2)
    p = state['p']

```

```

# 反発力（近づきすぎ防止）
for nb in neighbors:
    r = p - nb['p']
    d = np.linalg.norm(r) + 1e-6

    if d < self.formation_radius:
        # 近距離では強い反発力
        force += self.k_r * r / (d**3)

# 吸引力（目標位置への復帰）
force += -self.k_a * (p - state['p_ref'])

# 力の大きさを制限
force_norm = np.linalg.norm(force)
if force_norm > self.max_velocity * 10:
    force = force / force_norm * self.max_velocity * 10

return force

def project_to_feasible_motion(self, force: np.ndarray, state: Dict) -> np.ndarray
    """実行可能な運動に射影（バランス考慮）"""
    # IMUから傾きを取得
    if state['imu'] is not None:
        orientation = state['imu'].orientation
        # 簡易的な傾き計算（実際は適切な変換が必要）
        tilt = np.array([orientation.x, orientation.y])

        # 傾きが大きい場合は速度を制限
        tilt_magnitude = np.linalg.norm(tilt)
        if tilt_magnitude > 0.1:
            force *= (1.0 - tilt_magnitude)

    # 速度制限
    velocity = force * 0.1 # 力を速度に変換（簡易的）
    velocity_norm = np.linalg.norm(velocity)

    if velocity_norm > self.max_velocity:
        velocity = velocity / velocity_norm * self.max_velocity

```

```

        return velocity

def safety_check(self, velocity: np.ndarray, state: Dict) -> bool:
    """安全性チェック：ZMP・関節限界考慮"""
    # 簡易的な実装：速度が閾値を超えたら拒否
    if np.linalg.norm(velocity) > self.max_velocity * 1.5:
        return False

    # IMUによる転倒検知
    if state['imu'] is not None:
        orientation = state['imu'].orientation
        # 簡易的な転倒判定
        if abs(orientation.x) > 0.3 or abs(orientation.y) > 0.3:
            return False

    return True

def safe_stop_velocity(self) -> np.ndarray:
    """安全停止速度"""
    return np.zeros(2)

def control_loop(self):
    """メイン制御ループ"""
    try:
        # 状態取得
        state = self.sense_state()
        neighbors = self.get_neighbors()

        # コンセンサス更新
        state['x'] = self.consensus_update(state, neighbors)

        # フォーメーション力計算
        force = self.formation_force(state, neighbors)

        # 実行可能運動に射影
        desired_velocity = self.project_to_feasible_motion(force, state)

        # 安全性チェック
        if not self.safety_check(desired_velocity, state):
            desired_velocity = self.safe_stop_velocity()

```

```

        self.get_logger().warn('Safety_veto_activated')

    # 速度指令発行
    cmd_vel = Twist()
    cmd_vel.linear.x = desired_velocity[0]
    cmd_vel.linear.y = desired_velocity[1]
    self.cmd_vel_pub.publish(cmd_vel)

except Exception as e:
    self.get_logger().error(f'Control_loop_error:{e}')
    # エラー時は安全停止
    cmd_vel = Twist()
    self.cmd_vel_pub.publish(cmd_vel)

def publish_state(self):
    """状態を公開（近隣更新用）"""
    try:
        state = self.sense_state()

        state_msg = PoseStamped()
        state_msg.header = Header()
        state_msg.header.stamp = self.get_clock().now().to_msg()
        state_msg.header.frame_id = f'robot_{self.robot_id}'

        state_msg.pose.position.x = state['x'][0]
        state_msg.pose.position.y = state['x'][1]
        state_msg.pose.position.z = 0.0

        self.state_pub.publish(state_msg)

    except Exception as e:
        self.get_logger().error(f'State_publish_error:{e}')

def main(args=None):
    rclpy.init(args=args)
    node = ConsensusFormationNode()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:

```

```

        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

実装上の考慮とパラメータチューニング：

- メッセージレートを制限するため近隣半径を選ぶ．典型的なヒューマノイド群は雑踏環境で 2-5 近隣を用いる．
- 重み設計 w_{ij} は次数正規化し，異種接続でもコンセンサス速度を維持．
- k_r は安全半径未満の距離で反発力が吸引力を上回るように設定．
- 非同期コンセンサスでパケットロスに対応．タイムスタンプ付き近隣状態を保存し，古いデータを無視．

ヒューマノイド群の現実的制約：

- 全身協調：フォーメーション制御は運動学的到達可能性を逸脱する関節配置を要求してはならない．
- 遅延と順不同メッセージは一貫性の一時的ずれを生じる．状態経過閾値を実装する．
- エネルギー・計算限界：重いオンボード計算はバッテリー寿命を短縮．安全時には非クリティカル集約をエッジサーバにオフロード．

運用上のリスクとトレードオフ：

- 分散化は単一故障に対する頑健性を高めるが，大域最適化の難易度を増す．
- 積極的なポテンシャル利得は収束時間を短縮するが，振動とエネルギー使用を増大．
- 通信スパース化は帯域を下げるがコンセンサスを遅延させる；ミッションクリティカル期限に合わせたレートを選ぶ．
- 安全監督の上書きは群目標と局所的衝突を生じる．人の安全を優先する仲裁方針を設計する．

エンジニアリングへの影響：安定性クリティカル反射を高周波で，コンセンサスを低周波で動作するマルチレートスタックを実装する．ハードウェア展開前に Isaac Sim で物理ベースヒューマノイドモデルで検証する．最悪遅延・パケットロスシナリオを用いて故障モードを定量化する．

28.3 ケーススタディ：協働ヒューマノイド

群知能の分散協調モチーフと多ロボット協調における明示的タスク割り当てパターンを基に，本ケーススタディは制約のある産業組立における協働ヒューマノイドの実用的設計を示す．シナリオは現実的な通信遅延下での同期操作，共有ペイロード処理，安全を意識した歩容適応を重視する．

問題設定．2 台のヒューマノイドロボットが限られた通路内で 25 kg のパネルを共同で持ち上げ位置決めしなければならない．目的は：

- ペイロード姿勢を 2 cm・2 度の許容範囲内に維持；
- バランスを保ちながら総アクチュエータエネルギーを最小化；
- ロボットが故障を検出または人が安全ゾーンに入った場合にフェイルセーフな解放を保証.

技術解析. 各ロボットを一般化状態 $x_i(t)$ とエンドエフェクタレンチ $f_i(t)$ で表現する. ペイロード平衡は接触力に制約を課す：

$$[H] \sum_{i=1}^N J_i^\top(q_i) f_i + m_p g = 0, \quad (229)$$

ここで $J_i(q_i)$ はロボット i のエンドエフェクタヤコビアン, m_p はペイロード質量, g は重力である. タスク割り当てはエネルギーコストと追従誤差をトレードオフする制約付き二次計画問題 (QP) として定式化される：

$$[H] \min_{f_1, \dots, f_N} \sum_{i=1}^N \left(\frac{1}{2} f_i^\top W_i f_i + \lambda \|J_i(q_i)^\top f_i - \tau_i^*\|^2 \right) \quad \text{s.t.} \quad (229), f_i \in \mathcal{F}_i, \quad (230)$$

ここで W_i はアクチュエータ努力を重み付け, τ_i^* はプランナからの公称レンチ目標, \mathcal{F}_i は摩擦・接触力制限を課す.

制御・同期設計選択：

- エンドエフェクタでインピーダンス制御 (コンプライアンス制御手法) を用いて負荷を共有しミスマイメントを吸収する. 各ロボットごとに所望インピーダンス M_d , B_d , K_d を定義しコンプライアンスを調節する.
- 役割割り当てに市場ベースオークションを採用する. ロボットは到達可能把持品質, 残存バッテリー, 熱状態に基づいて入札する.
- ペイロード姿勢を中央サーバなしで推定する分散オブザーバを用いる. 各ロボットは局所推定 \hat{p}_p を保持し, コンセンサスフィルタで隣接測定を融合しパケットロスに頑強とする.
- 安全：接触ベース故障検出を実装する. 測定レンチが閾値を超えて逸脱した場合, 事前計画された安全解放と二次支援シーケンスが作動する.

実装概要. シミュレーションからハードウェアへのパイプラインはシナリオ検証に Isaac Sim, 通信に ROS2, 高レベル協調に GROOT Behavior Trees を用いる. 主要実装モジュール：

1. 知覚：ペイロード姿勢のための深度カメラ・力トルク融合；
2. アロケータ：入札を実行し一次／二次役割を割り当てるオークショナーノード；
3. コントローラ：インピーダンスコントローラと分散 QP ソルバを 250 Hz で実行；
4. 安全スーパーバイザ：近接を監視しグレースフルハンドオフを自動化.

コンパクトな ROS2 スタイル Python スニペットはビヘイビアツリーに統合されたオークショナービヘイビアを示す. コメントは意図的に簡潔にしてある.

コードサンプル 98 役割割り当てのための簡略化オークショナーノード

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
```

```

from geometry_msgs.msg import PoseStamped
from std_msgs.msg import Float32
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
import math

class Auctioneer(Node):
    def __init__(self):
        super().__init__('auctioneer')

        # パラメータ宣言
        self.declare_parameter('bid_rate', 10.0,
                                ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE,
                                                        description='Bid_publish_frequency'))
        self.declare_parameter('battery_topic', '/battery/percentage',
                                ParameterDescriptor(type=ParameterType.PARAMETER_STRING))
        self.declare_parameter('reachability_radius', 2.0,
                                ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE))

        # QoS設定
        qos = QoSProfile(reliability=ReliabilityPolicy.RELIABLE,
                          history=HistoryPolicy.KEEP_LAST,
                          depth=10)

        # 購読・配信
        self.create_subscription(PoseStamped, '/payload/pose',
                                  self.pose_cb, qos)
        self.create_subscription(Float32, self.get_parameter('battery_topic').value,
                                  self.battery_cb, qos)
        self.bid_pub = self.create_publisher(Float32, '/allocator/bid', qos)

        # タイマー
        period = 1.0 / self.get_parameter('bid_rate').value
        self.timer = self.create_timer(period, self.publish_bid)

        # 状態変数
        self.latest_pose = None
        self.battery_pct = 1.0
        self.local_quality = 0.0

    def pose_cb(self, msg: PoseStamped):

```

```

        self.latest_pose = msg
        self.local_quality = self.estimate_quality(msg)

def battery_cb(self, msg: Float32):
    self.battery_pct = max(0.0, min(1.0, msg.data))

def estimate_quality(self, pose: PoseStamped) -> float:
    # 到達可能性とバッテリー残量を掛け合わせた品質指標
    reach = self._reachability(pose)
    return reach * self.battery_pct

def _reachability(self, pose: PoseStamped) -> float:
    # 原点からの距離に応じて線形減衰
    r = self.get_parameter('reachability_radius').value
    d = math.hypot(pose.pose.position.x, pose.pose.position.y)
    return max(0.0, 1.0 - d / r) if r > 0.0 else 0.0

def publish_bid(self):
    if self.latest_pose is None:
        return # 初期データ未受信
    bid = Float32()
    bid.data = float(self.local_quality)
    self.bid_pub.publish(bid)

def main(args=None):
    rclpy.init(args=args)
    node = Auctioneer()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

テスト・検証戦略. 段階的計画に従う:

- コンセンサスフィルタと QP ソルバをランダム遅延を伴うシミュレーションで検証;
- ダミーペイロードとトルクセンサを用いたハードウェアインザループ試験を実施;
- 緊急停止制御を持つ試験操作者とともにヒューマンインザループ安全試験を実施.

技術的影響, トレードオフ, 運用上のリスク:

- ・集中型対分散割り当て：集中型オークションは大域最適性を簡素化するが単一障害点と遅延ボトルネックを生じる。分散型オークションは頑健性を高めるが非最適性のコストが生じる。
- ・コンプライアンスチューニング：高コンプライアンスはアライメント許容を改善するが精度を低下させ、パートナー故障時に危険なペイロードドリフトを許す可能性がある。
- ・通信制約：コンセンサスと同期はパケットロスで劣化する。設計は指定遅延・パケットドロップ率を許容しなければならず、さもないとタイムアウトとフォールバックビヘイビアを実装する。
- ・センサ依存：カトルクセンサ故障は安全でない力分布を生じる可能性がある。冗長センシングと保守的安全閾値が必須である。
- ・エネルギー・熱トレードオフ：積極的な力共有はピーク関節トルクを低減するが両ロボット間でエネルギー消費を増加させる可能性がある。

設計者は検証可能な安全ビヘイビア、エンドツーエンド遅延予算の計測、縮退モード計画を優先すべきである。実地展開では代表倉庫通路での繰返しストレステストと敵対条件下での安全スーパーバイザの正式検証が必要である。

29 ロボティクスにおける倫理的な行動

29.1 意思決定の制約

堅牢な倫理的挙動には、計画時および実行時に働く明示的な制約が必要である。問題定義：行動を提案する公称決定モジュールが与えられたとき、それらの行動が実行前に安全性、法的・物理的・社会的制約を満たすことを保証する。ヒューマノイド文脈では、次のドメインから制約が生じる：

- ・安全上重要な物理限界：関節トルク、重心（CoM）位置、バランス余裕。
- ・人間中心の制約：パーソナルスペース、同意、プライバシーに配慮したセンシング。
- ・法的・規制上の制約：立入禁止区域、データ保持ルール。
- ・知覚不確実性：センサがノイジーな場合の確率的制約。
- ・リソース制約：バッテリー残量、アクチュエータ発熱、通信帯域。
- ・リアルタイムデッドライン：制御ループは遅延上限を満たす必要がある。

技術解析は3層で進める：仕様、検証、ランタイム強制。

仕様

- ・ハード制約：決して違反してはならない不変条件（例：関節限界、衝突回避）。
- ・ソフト制約：トレードオフ可能な推奨挙動（例：不快感最小化）。
- ・確率的制約：不確実性下での違反許容（チャンス制約）。

制約付き制御により強制を形式化する。ロボット状態を x 、制御／決定出力を u とする。実用的なランタイムフィルタは、公称行動 u_{nom} を最小限修正して制約を満たす二次計画（QP）を用いる。標

準的な定式化は：

$$\begin{aligned}
 u^* &= \arg \min_{u \in \mathbb{R}^m} \|u - u_{\text{nom}}\|_W^2 \\
 [H] \quad &\text{s.t. } A(x)u \leq b(x) \quad (\text{線形化運動学的／回避}) \\
 &\quad L_f h(x) + L_g h(x) u + \alpha(h(x)) \geq 0 \quad (\text{CBF 安全制約})
 \end{aligned} \tag{231}$$

ここで $\|v\|_W^2 = v^\top W v$ 、2 行目は制御バリア関数 (CBF) 制約を符号化する。CBF は安全集合 $\{x \mid h(x) \geq 0\}$ を定義する。不等式 $\dot{h}(x) + \alpha(h(x)) \geq 0$ は適切なクラス K 関数 α に対して前方不変性を強制する。

知覚不確実性を扱うにはチャンス制約が必要。物体姿勢 p に依存する衝突余裕 $g(x, p)$ について、分布 $\mathcal{N}(\hat{p}, \Sigma)$ を持つとき、

$$\Pr(g(x, p) \leq 0) \leq \epsilon,$$

を強制する。これは線形化とガウス境界を用いて保守的に近似するか、サンプリングベース検証で強制できる。

実装戦略

1. 階層アーキテクチャ

- ・高レベルプランナがタスクと軌道を提案。
- ・ミドルレベル安全監督者が QP / CBF フィルタを実装。
- ・ローレベルコントローラがフィルタ済み指令を追従。

2. ビヘイビアツリー統合

- ・リスクの高いサブツリーを、ティック伝播前に監督者に問い合わせる安全デコレータで包む。
- ・ガードノードを用いて制約がアクティブになったときに計画を切り替える。
- ・オフライン監査のため失敗をログに記録。

3. ランタイム検証

- ・信号時限論理 (STL) などの時限論理仕様をチェックするモニタを実装。
- ・モニタが差し迫った違反を通知したら、安全挙動 (スタンバイ、コンプライアントモード) に先取りする。

4. 形式的保証とシミュレーション

- ・到達可能性解析を用いて有界擾乱下で到達可能な状態に上限を設ける。
- ・センサノイズと人間モデルを含む高忠実度シミュレーション (Isaac Sim) で方策を検証。

実用的なコード例：公称速度指令に QP 安全フィルタを適用する ROS2 ノード。当該ノードは OSQP を用いて QP を解く；コメントはビヘイビアツリーとの統合点を示す。

コードサンプル 99 ヒューマノイド速度指令向け QP ベース安全フィルタ

```
import numpy as np
import osqp
from scipy import sparse
from typing import List, Tuple, Optional
```

```

# 障害物型定義
Obstacle = Tuple[np.ndarray, np.ndarray] # (mean, cov)

def solve_safety_qp(
    u_nom: np.ndarray,
    x: np.ndarray,
    obst: List[Obstacle],
    v_max: np.ndarray = np.array([0.8, 1.0, 0.5]),
    v_min: np.ndarray = np.array([-0.8, -1.0, -0.5])
) -> np.ndarray:
    """
    速度指令  $u_{nom}$  を安全な速度  $u$  に修正する QP フィルタ
    """
    nu = len(u_nom)
    P = sparse.csc_matrix(2.0 * np.eye(nu))
    q = -2.0 * u_nom

    # 速度上下限
    A_vel = np.vstack([np.eye(nu), -np.eye(nu)])
    b_vel = np.hstack([v_max, -v_min])

    A_list, b_list = [A_vel], [b_vel]

    # 衝突回避制約の追加
    for mean, cov in obst:
        n, beta = linearize_collision_constraint(x, mean, cov)
        A_list.append(n.reshape(1, -1))
        b_list.append(np.array([beta]))

    A = sparse.csc_matrix(np.vstack(A_list))
    b = np.hstack(b_list)

    # OSQP ソルバ設定
    prob = osqp.OSQP()
    prob.setup(P, q, A, b, verbose=False, polish=True, eps_abs=1e-4, eps_rel=1e-4)
    res = prob.solve()

    # 求解失敗時は公称指令を返す
    return res.x if res.info.status == 'solved' else u_nom

```

```

def linearize_collision_constraint(
    x: np.ndarray,
    mean: np.ndarray,
    cov: np.ndarray,
    d_min: float = 0.3
) -> Tuple[np.ndarray, float]:
    """
    障害物に対する線形化された衝突回避制約を返す
    """
    # 相対位置と距離
    dx = mean[:2] - x[:2]
    dist = np.linalg.norm(dx)
    if dist < 1e-6:
        dx = np.array([1.0, 0.0])
        dist = 1e-6

    # 法線ベクトル
    n = dx / dist

    # 速度空間での制約係数（回転成分はゼロ）
    J = np.array([n[0], n[1], 0.0])

    # 安全距離を考慮したバイアス
    beta = (d_min - dist) / 0.1 # 0.1[s] 予見時間

    return J, beta

```

設計上のトレードオフと運用上のリスク

- 保守性対性能：強い安全余裕はタスク効率を低下させる。ミッションクリティカリティに基づき余裕を選定。
- 計算コスト：QP や CBF 評価はループデッドラインを満たす必要がある。線形化を事前計算しソルバをウォームスタート。
- センサ遅延と誤認識：遅延／破損入力は無効化する。保守的融合とハートビートモニタを用いる。
- 構成可能性：複数のソフト制約を組み合わせると対立目的が生じる。優先度重み付けまたは辞書式 QP を用いる。

エンジニアリングへの影響

- プランナの下に、検証可能で最小修正フィルタを配置してハード安全を強制する。
- 知覚不確実性が重要な場所では確率モデルを用い、シミュレーション+最悪ケース解析で検証する。
- 法的コンプライアンスと事後分析のため、監査ログと人間によるオーバーライドを維持する。

運用上は、明示的かつ強制された意思決定制約を持つヒューマノイドシステムを設計する。この階層的アプローチは意図しない挙動を削減しながら、安全性、敏捷性、計算リソースの間のトレードオフを明確にする。

29.2 意図しない挙動の回避

これまでに議論してきた意思決定の制約を踏まえ、ここではヒューマノイドロボットにおける仕様ゲーミングやその他の意図しない挙動を防ぐための具体的な手法に焦点を当てる。以下のアプローチは、形式的制約、ランタイムモニタリング、堅牢なテストを、転倒防止、人間との近接安全性、タスクコンプライアンスといった運用要件に結びつける。

意図しない挙動は、最適化目的、モデル、または仕様が現実の安全目標と乖離したときに生じる。典型的な失敗モードは以下の通りである：

- 仕様ゲーミング：方策が報酬設計の癖を悪用し、意図に反しながら高報酬を達成する。
- 分布シフト：知覚モデルが訓練分布外の新規入力を誤分類する。
- 潜在的な競合：複数の目的（速度、エネルギー、タスク精度）が競合し、安全でない妥協を生む。
- 創発ループ：ビヘイビアツリーやプランナーが稀な条件下で安全性を低下させるサイクルに入る。

問題定義. 状態 x と公称コントローラ u_{des} を持つヒューマノイドに対し、安全不変量 $S = \{x \mid h_i(x) \geq 0\}$ を強制し、可能な限り公称性能を許容する。安全臨界制御（ランタイムシールド）は、動力学 $\dot{x} = f(x) + g(x)u$ の下ですべての h_i 不変量を維持するために、 u_{des} を最小限変更しなければならない。

技術解析. コントロールバリア関数（CBF）は、安全集合の順方向不変性を強制する数学的に根拠のある手法を提供する。スカラー関数 $h(x)$ は安全集合 $S = \{x \mid h(x) \geq 0\}$ を定義する。有効な CBF は、 S の近傍内のすべての x に対し、

$$[H]\dot{h}(x, u) = \nabla h(x)^\top (f(x) + g(x)u) \geq -\alpha(h(x)), \quad (232)$$

を満たす u が存在することを保証する。ここで、 α は拡張クラス K 関数であり、しばしば線形 $\alpha(s) = \gamma s$ ($\gamma > 0$) が選ばれる。不等式 (232) を強制すると、多くのロボットモデルに対して凸制約が得られる。最小侵襲安全フィルタは次の二次計画（QP）となる：

$$[H]u^* = \arg \min_u \|u - u_{\text{des}}\|^2 \quad \text{s.t.} \quad \nabla h(x)^\top g(x)u \geq -\nabla h(x)^\top f(x) - \alpha(h(x)). \quad (233)$$

ヒューマノイドにとって有用な $h(x)$ 関数は以下を含む：

- 転倒マージン： $h_f(x) = \text{ZMP}_{\text{margin}}(x) - \epsilon$ 、圧力中心が支持多角形内に留まることを保証。
- 人間との近接： $h_p(x) = \|p_{\text{hand}} - p_{\text{human}}\| - d_{\text{safe}}$
- 関節限界スラック：すべての関節に対し $h_j(x) = q_{\text{max}} - q_i$ および $h'_j(x) = q_i - q_{\text{min}}$

実装上の注意：

1. モデル精度：ロボットの動的モデルから ∇h および f, g を計算する。同定とオンラインパラメータ適応を用いてモデル誤差を削減する。
2. リアルタイム QP：専用制御ハードウェア上で特化ソルバ（OSQP、qpOASES）を用いてミリ秒単位のデッドラインを満たす。
3. センサフュージョン：フィルタリングされた状態推定値を用いる。 h 閾値を厳しくすることで不確実性を組み込む（ロバスト安全マージン）。
4. ビヘイビアツリーとの統合：安全フィルタをミドルウェアレイヤーとして配置し、ビヘイビアリーフにはしない。フィルタは高レベルプランナーに対して透過的でなければならない。

形式的検証とテスト：ビヘイビアツリーやハイブリッド制御方針は、生存性および安全性特性についてモデル検査を受けるべきである。知覚依存の挙動には確率的モデルチェッカを用い、可能な限り高レベル挙動を有限状態抽象化に変換する。形式的解析を、シミュレーションでの体系的なシナリオ生成で補完する：

- 環境および人間の姿勢のドメインランダムマイゼーション。
- 稀な失敗モードを露呈する敵対的シナリオ検索。
- Isaac Sim におけるランダムイズされた雑然とした環境およびセンサ障害を伴う閉ループテスト。

ランタイムモニタリングと異常検知。重要な述語を観測し、違反時に介入するモニタを実装する。
推奨レイヤー：

- ハード安全シールド：CBF/QP 制約を物理的安全性のために強制。
- ビヘイビアルスーパーバイザ：繰り返しスケジューラプリエンプションやツリーサイクルを監視し、エスカレーションフラグを上げる。
- 知覚信頼度ゲーティング：信頼度の低い検出に依存するアクションを抑制。

実用的アルゴリズムテンプレート。公称制御、安全フィルタ、監督フォールバックを組み合わせる：

1. モーションプランナまたは学習方針から u_{des} を計算。
2. (233) の安全 QP を実行し、 u^* を得る。
3. QP が非実現または大きな逸脱を要求する場合、フォールバック挙動（例：停止、後退）をトリガ。
4. イベントをログし、教師あり学習または仕様改良のためにオペレータに警告。

コード例：コントロールバリア不等式を用いた簡単なランタイム安全チェック。このスニペットは制御レートで評価される安全フィルタループを示す。

コードサンプル 100 Runtime safety filter using a linearized CBF constraint

```
import numpy as np
from typing import Callable, Optional
```

```
State = np.ndarray
```

```
Control = np.ndarray
```

```
def safety_filter(
    x: State,
    u_des: Control,
    h: Callable[[State], float],
    dh_dx: Callable[[State], np.ndarray],
    f: Callable[[State], State],
    g: Callable[[State], np.ndarray],
    gamma: float = 5.0,
    tol: float = 1e-6,
    max_corr: Optional[float] = None,
) -> Control:
    """
    1次元安全制約に対する解析的QP投影.
    制約:  $dh_{dx}(f + gu) + \gamma h \geq 0$ 
    """
    hx = h(x)
    dhx = dh_dx(x)
    fx = f(x)
    gx = g(x)

    a = float(dhx @ fx)
    b = dhx @ gx  # (1, m)

    alpha = gamma * hx
    rhs = -a - alpha
    lhs = float(b @ u_des)

    # すでに安全
    if lhs >= rhs - tol:
        return u_des

    # 半空間への射影
    b_norm_sq = float(b @ b.T) + tol
    scale = (rhs - lhs) / b_norm_sq
    u_corr = u_des + scale * b.T

    # 補正量が許容範囲を超える場合は零指令にフォールバック
    if max_corr is not None:
```

```

    du = np.linalg.norm(u_corr - u_des)
    if du > max_corr:
        return np.zeros_like(u_des)

return u_corr

```

設計上のトレードオフと運用上のリスク：

- ・保守性対能力：厳しい安全マージンは事故を減らす、敏捷性とタスク性能を制限する。
- ・計算負荷：リアルタイム QP とモニタリングは CPU 需要とレイテンシを増加させ、ハードウェア共設計を要する。
- ・モデルミスマッチ：不正確な動力学は誤検知または見逃し違反を生む。定期的な較正が不可欠。
- ・フィルタへの過度な依存：シールドは多くの失敗を防ぐが、改良された仕様設計と徹底したテストに取って代わるものではない。

エンジニアリングへの影響．配備されたヒューマノイドに対して、形式的制約、ランタイム安全フィルタ、包括的シミュレーションテスト、ヒューマンインザループエスカレーションを組み合わせる。安全ソルバのための計算ヘッドルームを割り当てる。安全介入を追跡・ログし、行動仕様を改良する。これらの実践は、安全性保守性とミッション性能の間の明示的なトレードオフを行いながら、意図しない挙動を削減する。

29.3 規制および倫理的考慮事項

先行する小節で扱った制約付き意思決定パターンとランタイムチェックを結びつけ、ここでは法律的要件と倫理的フレームワークがヒューマノイドの挙動に対する工学上の制約となる仕組みを論じる。本小節では、法律や規範を検証可能なランタイムチェック、モニタリング、および認証対応の監査証跡に変換し、それらをビヘイビアツリーやポリシーモジュールと統合する方法を示す。

問題定義と設計への関連性：製造者およびインテグレータは、ヒューマノイドロボットが複雑なタスク遂行中に安全法、プライバシー法、業種別規則を満たすことを保証しなければならない。規制は、（介護タスク中の人間の顔への接触禁止といった）厳格な禁止事項と、（インシデントレビュー用のデータログといった）報告義務の両方を課す。倫理的には、システムは危害を回避し、プライバシーを尊重し、人間によるオーバーライドを提供しなければならない。工学上の課題は、これらの要件を、ビヘイビアツリーのような挙動設計フレームワークと互換性を持ち、実行可能かつ監査可能な運用上の制約として符号化することである。

技術分析：規制および倫理的要件は、実装可能な 3 つのカテゴリに分類される。

1. ハード安全制約。これらは常に低レベルコントローラまたはランタイムガードによって満たされなければならない。制約を関数 $C_i(x, u)$ として表現し、これを非正に保つ：

$$[H]C_i(x, u) \leq 0 \quad \forall i, \quad (234)$$

ここで x はシステム状態、 u は指令値である。例：最小分離距離、最大付加力、禁止区域。

2. 確率的リスク閾値。センシングや人間の意図が不確実な場合、期待危害に対する上限を課す。リスク関数 $R(x, u)$ を定義し、高レベルコマンド実行前に $R(x, u) \leq \rho_{\max}$ を要求する。

3. 監査可能性および透明性要件。挙動決定を再構築するのに十分なテレメトリをログに記録する。
ログに暗号の署名を行い、保管責任ルールを満たす。

ヒューマノイドの挙動仲裁でよく使われる制約付き行動選択の定式化は次の通り：

$$[H] \min_{u \in \mathcal{U}} J(x, u) \quad \text{s.t.} \quad C_i(x, u) \leq 0, \quad i = 1, \dots, m, \quad (235)$$

ここで J は目的（効率、快適性）、 \mathcal{U} は実行可能指令空間である。この定式化は、制約がアクティブな際にリーフコマンドを拒否または修正するビヘイビアツリーデコレータに自然にマッピングされる。

具体的なリスク指標。人間の周辺での操縦について、近接度、人間姿勢の不確実性、アクチュエータ信頼性を組み合わせたコンパクトなリスクスコアを定義する：

$$[H] R = w_p \frac{1}{d + d_{\min}} + w_u \sigma_h + w_a (1 - \eta), \quad (236)$$

ここで d は最短距離、 d_{\min} は小さな正則化項、 σ_h は人間姿勢共分散ノルム、 η はアクチュエータ健康指標、 w_p, w_u, w_a は校正済み重みである。

実装パターンとコード：ランタイム倫理モニタを軽量な ROS2 ノードとして実装し、プランナまたはビヘイビアツリーから提案される行動を購読する。モニタは制約を計算し、 R を評価し、実行を許可するか、行動を修正するか、またはブロックする。ノードは決定と監査の理由をログに記録する。ノードはモータレベルコントローラの前にコマンドパイプラインに配置し、最終仲裁者として機能するようにする。

コードサンプル 101 Runtime ethics monitor that approves or rejects proposed actions

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import String, Float32
from geometry_msgs.msg import PoseStamped
from example_interfaces.msg import Float64MultiArray # 仮のアクション型
from builtin_interfaces.msg import Time
import time
import json

R_MAX = 10.0 # リスク閾値（適宜調整）

class EthicsMonitor(Node):
    def __init__(self):
        super().__init__('ethics_monitor')

        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
```

```

        history=HistoryPolicy.KEEP_LAST,
        depth=1
    )

    self.human_pose = None
    self.actuator_health = 1.0

    self.sub_action = self.create_subscription(
        Float64MultiArray, 'proposed_action', self.on_proposed_action, qos)
    self.sub_pose = self.create_subscription(
        PoseStamped, 'human_pose', self.on_human_pose, qos)
    self.sub_health = self.create_subscription(
        Float32, 'actuator_health', self.on_actuator_health, qos)

    self.pub_approved = self.create_publisher(
        Float64MultiArray, 'approved_action', qos)
    self.pub_audit = self.create_publisher(String, 'audit_log', 10)

def on_human_pose(self, msg):
    self.human_pose = msg

def on_actuator_health(self, msg):
    self.actuator_health = msg.data

def compute_risk(self, action):
    if self.human_pose is None:
        return float('inf') # 人間位置不明は最大リスク
    d = self._min_distance_to_human(action)
    sigma = self._pose_uncertainty_norm()
    eta = self.actuator_health
    return 1.0/(d+0.01) + sigma*0.5 + (1-eta)*2.0

def _min_distance_to_human(self, action):
    # アクションと人間の距離簡易推定 (Euclidean)
    return 1.0 # 実装依存で置換

def _pose_uncertainty_norm(self):
    # 位置推定の不確実性ノルム (簡易固定値)
    return 0.1

```

```

def safe_fallback(self):
    # 安全な停止指令（ゼロ速度）
    fallback = Float64MultiArray()
    fallback.data = [0.0]*6
    return fallback

def on_proposed_action(self, msg):
    risk = self.compute_risk(msg.data)
    if risk > R_MAX:
        self.publish_audit(msg, risk, "rejected")
        approved = self.safe_fallback()
    else:
        self.publish_audit(msg, risk, "approved")
        approved = msg
    self.pub_approved.publish(approved)

def publish_audit(self, msg, risk, status):
    log = {
        "timestamp": self.get_clock().now().to_msg().sec,
        "proposed": list(msg.data),
        "risk": risk,
        "status": status
    }
    audit_msg = String()
    audit_msg.data = json.dumps(log)
    self.pub_audit.publish(audit_msg)

def main(args=None):
    rclpy.init(args=args)
    node = EthicsMonitor()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

運用工学上の注意事項：

- モニタをアクチュエータに可能な限り近づけて攻撃表面を減らす。
- 可能な限り形式手法を用いる。低レベルコントローラに対して Control Barrier Functions (CBF)

を合成し、式 (1) の制約に対する数学的保証を提供する。

- ログが法的な保存期間とアクセス制御ルールを満たすことを確実にする。証拠保全のため、追記専用ストレージと暗号署名を適用する。
- 式 (3) の重みを Isaac Sim でのシナリオテストを用いて校正し、センサドロップアウトや動的な人間の動作といったエッジケースを含める。

検証と認証：制約境界、敵対的入力、センサ故障を行使するテストスイートを提供する。重要な制約に対する不変条件を証明するために、モデルチェックまたはランタイム検証を用いる。監査人のためにテストベクタとシミュレーショントレースを記録する。

トレードオフとリスク：

- ハード制約は自律性を低下させ、タスク完了時間を増加させることがある。
- 保守的な閾値は偽陰性を減らすが偽陽性を増加させ、不必要な介入を引き起こす。
- ランタイムチェックによる計算オーバーヘッドは遅延を増加させることがある。リアルタイム境界とフォールバック挙動を持つモニタを設計する。
- 説明責任のためのログへの過度の依存はプライバシー懸念を生む。保存される個人データを最小化し、可能な限り匿名化を適用する。

具体的な工学上の影響：

- ビヘイビアツリーを、行動実行前に `ethicsMonitor` を呼び出す明示的な倫理デコレータとともに設計する。
- リアルタイム検証とセキュアロギングのための計算予算を割り当てる。
- システム能力を認証要件にマッピングするために、規制当局との関与を早期に計画する。
- 制約またはモデルが変更された際の再認証を含む更新手順を維持する。

これらの実践は、運用能力を保ちながら法的および倫理的要求事項を施行する。それらは安全性、透明性、システム有用性の間のトレードオフを明らかにし、フィールド展開前に緩和しなければならないリスクを特定する。

人間-ロボット相互作用

30 HRI の基礎

30.1 人間要因の理解

これまでの章で扱った知覚と行動設計の原則を踏まえ、本小節ではヒューマノイドロボットの行動を制約する人間要因を分析する。焦点は、エンジニアが速度、精度、社会的受容性を安全性と使いやすさと交換できる、測定可能で設計可能な量にある。

人間要因の問題定義。実用上の展開では、ヒューマノイドロボットは以下を同時に満たさなければならない：

- 接触時の負傷を防ぐ物理的安全性制約；
- オペレータの作業負荷を軽減し混乱を防ぐ認知的制約；

- ・ パーソナルスペースを保持し意図を示す社会的受容性制約。

技術的分析。人間要因を多目的工学問題として扱う。設計と制御の判断に繰り返し登場する最小限の変数集合を定義する：

- ・ t_r : 人間の反応時間 (s)、対象集団について $t_r \sim \mathcal{N}(\mu_r, \sigma_r^2)$ とモデル化。
- ・ d : 瞬時のロボットから人間までの距離 (m)。
- ・ v : ロボット速度の大きさ (m/s)。
- ・ a_{\max} : ロボットの減速能力 (m/s²)。
- ・ F : 接触力 (N)。
- ・ r_{ps} : 好まれるパーソナルスペース半径 (m)、文脈によって変化し得る。
- ・ T : 信頼スコア、相互作用履歴から得られる無次元推定値。

動的相互作用のための安全余裕。衝突回避のために、停止距離が期待される人間の動作と反応の不確実性を上回ることを要求する。要求される停止距離を以下のように設計する：

$$[H]d_{\text{stop}}(v) = v(\mu_r + k\sigma_r) + \frac{v^2}{2a_{\max}}, \quad (237)$$

ここで k は信頼水準を設定する (例： $k = 2$ で正規性の下で $\approx 97.5\%$ の安全性)。安全な運用には $d - d_{\text{human_motion}} \geq d_{\text{stop}}(v)$ を要求する。

人間工学的指差しと到達。ハンドオーバーやタッチ相互作用のために、Fitts の法則を用いてコントロールの大きさを決め接近速度を計画する：

$$[H]\text{MT} = a + b \log_2 \left(1 + \frac{D}{W} \right), \quad (238)$$

ここで MT は動作時間、 D は目標までの距離、 W は目標幅である。MT が低いほどハンドオーバーは速くなるが、速すぎると知覚され得る。Fitts のパラメータ a, b は対象人口について経験的に測定すべきである。

人間の不快感と統合コスト。物理的、認知的、社会的コストを工学可能なスカラーに統合し最適化する：

$$[H]C_{\text{total}} = \alpha C_{\text{phys}}(F, d) + \beta C_{\text{cog}}(\text{MT}, NL) + \gamma C_{\text{soc}}(r_{\text{ps}}, \Delta\theta), \quad (239)$$

重み α, β, γ はミッションごとに選択する。ここで NL は同時メッセージ数 (ノイズレベル)、 $\Delta\theta$ は期待される正面方向からの角偏差である。実用的な選択：

- ・ $C_{\text{phys}} \propto \max(0, F - F_{\text{safe}})^2$ は過剰な力をペナルティとする。
- ・ C_{cog} は複数の同時モダリティと MT の予測可能性の低下とともに増加する。
- ・ C_{soc} は $d < r_{\text{ps}}$ または後方からの接近といった侵入をペナルティとする。

実装指針。分析をセンシングと制御コンポーネントに翻訳する：

1. センシング

- ・ マルチモーダル知覚を用いる：深度カメラ、レーダー、サーモグラフィ、近接センサで頑健な d と人間姿勢推定を行う。
- ・ 監督付き相互作用中の応答時間ログからオンラインで μ_r, σ_r を推定する。

2. 制御とハードウェア

- 混雑環境では保守的な k を用いて式 (1) から計算される速度制限を施行する。
- 直列弾性アクチュエータまたはトルク制御を用いて接触力 F を F_{safe} に制限する。
- 意図シグナリングを実装する：頭部向きと LED で C_{cog} を減らし T を増加させる。

3. パーソナライゼーション

- Fitts パラメータ (式 (2)) と r_{ps} をユーザクラス (子供、高齢者、産業労働者) ごとに推定する。
- タスクの緊急性 (例：医療 vs. 物流) に応じて α, β, γ を適応させる。

アルゴリズムスニペット。以下の Python 関数は、感知された間隔を与えられた場合の保守的な最大許容速度を計算する。反応時間の不確実性とブレーキ能力を統合している。コメントは簡潔で実用的である。

コードサンプル 102 Compute safe maximum velocity for a given separation.

```
import math
from typing import Final

# 物理定数
_G: Final[float] = 9.80665 # 重力加速度 [m/s^2]

def safe_velocity(
    distance: float,
    mu_r: float,
    sigma_r: float,
    a_max: float,
    d_human_motion: float = 0.0,
    k: float = 2.0,
) -> float:
    """
    人間との衝突回避を保証する最大速度を返す。
    単位は全てSI系 (m, s, m/s, m/s^2)。
    """
    # 負値ガード
    if distance < 0.0 or mu_r < 0.0 or sigma_r < 0.0 or a_max <= 0.0:
        return 0.0

    margin: float = max(0.0, distance - d_human_motion)
    if margin <= 0.0:
        return 0.0

    tau: float = mu_r + k * sigma_r
    A: float = 1.0 / (2.0 * a_max)
```

```

B: float = tau
C: float = -margin

discriminant: float = B * B - 4.0 * A * C
if discriminant < 0.0:
    return 0.0

v_max: float = (-B + math.sqrt(discriminant)) / (2.0 * A)
return max(0.0, v_max)

```

実用的評価指標。以下で成功を測定する：

- 安全性制約下での完了までの時間；
- 稼働時間あたりのインシデント率；
- 反復セッションにわたる主観的作業負荷（NASA-TLX）と知覚された信頼；
- ターゲットタスクのハンドオーバーの失敗率と MT 統計。

工学への影響、トレードオフ、リスク。

- トレードオフ：速度を上げるとスループットは向上するがリスクと認知負荷が増加する。式 (3) の重みは政策決定を符号化する。
- センサ精度： σ_r の過小評価やノイズの多い d は安全でない速度制限を引き起こす。保守的なチューニングはリスクを軽減するが効率を低下させる。
- パーソナライゼーション vs. スケーラビリティ：ユーザごとのキャリブレーションは快適性を向上させるが、フリート全体への展開を複雑にする。
- 運用上のリスク：社会的手がかりの誤解やシグナリングの失敗は、驚き反応、過信、または負傷を引き起こし得る。冗長性と明確なフォールバック行動を設計する。

これらの公式と実用的モジュールをヒューマノイド設計サイクルの初期に採用する。そうすることで、安全性、人間工学、社会的受容性を、機械、ソフトウェア、相互作用設計の間で定量化可能な目的とする。

30.2 直感的なインタラクションの設計

前節の人間の知覚、注意、メンタルモデルの分析を踏まえ、これらの要因を工学原理として操作し、ヒューマノイドロボットと人間の間に直感的なインタラクションを生み出す。焦点は認知負荷の削減、ロボットの目標の可読性、産業・サービスシナリオにおける予測可能で安全な応答の確保にある。

問題定義. ヒューマノイドロボットは動作、視線、音声、ディスプレイを通じて意図を伝達しなければならない。目的は、人間パートナーが迅速に解釈・行動できる行動を選択することである。主な制約は人間の注意の帯域幅の限界、利用者の熟練度のばらつき、協調タスクにおける安全臨界タイミングである。

技術分析. インタラクション設計を導く 2 つの中心的で相補的な概念がある。

1. 予測可能性対可読性. 予測可能性はロボットの動作を目標に対する予想軌道と一致させる。可読

性は動作の初期段階で目標について情報を与える。形式的に、可読性は目標 g に対する部分軌道 $\xi_{0:t}$ の事後確率を最大化として定式化できる：

$$[H]P(g \mid \xi_{0:t}) \propto P(\xi_{0:t} \mid g) P(g). \quad (240)$$

設計は動的・安全制約下で $P(g \mid \xi_{0:t})$ を早期に高める軌道を選択する。

2. 認知コストとタイミング. 人間の情報処理能力は有限である。タスク時間と認知負荷を組み合わせたスカラー・コストを用いる：

$$[H]J = \alpha T_{\text{task}} + \beta C_{\text{cog}}(I), \quad (241)$$

ここで T_{task} は実行時間、 $C_{\text{cog}}(I)$ は情報チャネル I が引き起こす認知負荷、 α, β は速度と明確さをトレードオフする。

実用モデル. 上記を以下で実装する：

- 目標条件付き軌道の生成モデル $P(\xi \mid g)$ によりベイズ可読性スコアリング。
- モダリティ（音声、身振り、光）に認知コストを割り当てるチャネルモデル、ユーザ研究から推定。
- 候補コミュニケーション行動に対するソフトマックス決定方針：

$$[H]\pi(a \mid s) = \frac{\exp(-\gamma U(a, s))}{\sum_{a'} \exp(-\gamma U(a', s))}, \quad (242)$$

効用 $U(a, s) = w_\ell L(a, s) + w_s S(a, s) + w_c C(a)$ は可読性 L 、安全余裕 S 、認知コスト C を結合。

実装パターン. 候補生成＋スコアリングパイプラインを用いる：

1. 現在のタスクに整合した少数の動作・マルチモーダル信号を生成。
2. 初期軌道セグメントに対して式 (1) で可読性スコアを計算。
3. モダリティモデルとユーザプロファイルから認知コストを計算。
4. 検証モジュール（制御バリア関数または衝突チェック）で安全制約を施行。
5. 式 (3) で最効用行動を選択・実行し、オンラインで再計画。

コード例. 以下の Python リストは、可読性とコストで身振りテンプレートをスコアリングする簡易セレクタを示す。ROS2 ビヘイビアツリーノードまたは GROOT アクションノードへの統合に適している。

コードサンプル 103 可読性と認知コストに基づく身振り選択

```
import numpy as np
from typing import List, Dict, Callable, Any, Tuple

GestureTemplate = Dict[str, Any]
GoalPrior       = Dict[str, float]
GoalLikelihood  = Callable[[np.ndarray], float]
```

```

def score_gestures(
    gesture_templates: List[GestureTemplate],
    partial_obs: np.ndarray,
    user_prior: GoalPrior,
    goal_models: Dict[str, GoalLikelihood],
    w_leg: float = 1.0,
    w_safe: float = 1.2,
    w_cost: float = 0.8,
    temperature: float = 2.0,
    rng: np.random.Generator = np.random.default_rng(),
) -> GestureTemplate:
    """
    部分観測に対して最も効用の高い身振りテンプレートを返す
    """
    scores: List[Tuple[float, GestureTemplate]] = []

    for g in gesture_templates:
        traj = g["traj"]
        # 各ゴールに対する事後確率に比例した可読性を集計
        legibility = sum(
            goal_models[goal](partial_obs) * prior
            for goal, prior in user_prior.items()
        )

        # モダリティコストルックアップ
        modality_cost = {"speech": 1.5, "gesture": 1.0, "display": 0.8}.get(
            g["modality"], 1.0
        )

        safety_score = float(np.clip(g.get("safety", 0.0), 0.0, 1.0))

        # 効用関数
        utility = w_leg * legibility + w_safe * safety_score - w_cost * modality_cost
        scores.append((utility, g))

    # ソフトマックスで選択
    utilities = np.array([u for u, _ in scores])
    logits = temperature * (utilities - utilities.max())
    probs = np.exp(logits)
    probs /= probs.sum()

```

```
idx = rng.choice(len(scores), p=probs)
return scores[idx][1]
```

設計原理と工学制約.

- マルチモダリティ: 騒がしい環境で意図を明確にするため音声・視線・動作を組み合わせる。
- 早期ディスambiguation: 意図を早期に明らかにし人間の反応遅延を削減する身振りを優先。
- ユーザモデリング: 軽量のユーザモデルを維持し、詳細さとモダリティを熟練度に適応。
- 安全最優先計画: 常に最小安全余裕を施行; 可読性は衝突回避を損なってはならない。
- レイテンシ予算: 通信レイテンシを境界付き; 知覚・推論・駆動遅延をタイミング予算に含める。

評価指標. 以下でインタラクション品質を測定:

- タスク完了時間。
- 早期タイムスタンプでのロボット目標に対する人間の予測精度。
- NASA-TLX または他の認知負荷スコア。
- ニアミスおよび衝突率による安全検証。

工学への影響、トレードオフ、運用上のリスク.

- トレードオフ: 高速・最小手がかりはタスク時間を短縮するが誤解リスクを増大。式 (2) の α, β を調整。
- 適応コスト: オンラインパーソナライズは使いやすさを向上させるがシステム複雑性と故障モードを増やす。
- 文化間差異: 身振りの意味は集団間で異なる; データセットとテンプレートを広く検証。
- レイテンシと知覚誤差は誤通信を増幅; 安全臨界タスクでは堅牢なフォールバックと明示的確認を実装。

設計者はこれらの原理を制御ループ、知覚スタック、ビヘイビアツリーに統合し、代表タスクシナリオでトレードオフを実証的に定量化してから展開すべきである。

30.3 ケーススタディ：音声コマンドロボット

これまでの人間要因と直感的インタラクションに関する議論では、ユーザの認知限界、期待の整合性、マルチモーダルアフォーダンスを確立した。本ケーススタディでは、これらの原則を実用的な音声コマンドヒューマノイドに適用し、頑健性、安全性、運用可能性に焦点を当てる。

問題定義: ヒューマノイドが現実的な環境で音声コマンドを確実に実行できるようにすること。主な制約として、騒音の多い産業環境、安全上重要な動作に対する遅延制限、オンボード計算容量、クラウド処理に関するプライバシー規則、多様な話者特性が挙げられる。エンジニアリング目標は、正しい意図認識を最大化しながら、誤動作および安全でない動作を制限するモジュラーなパイプラインを構築することである。

技術分析 — センシングおよびフロントエンド。

- マイクロホンアレイ設計: 空間フィルタリングおよび音源定位を可能にするため、4 個以上のマイクロホンを持つ円形または線形アレイを使用する。ビームフォーミングにより、話者への指向

性利得が得られ、拡散ノイズが低減される。

- 遅延和ビームフォーミングは計算が簡単である。ステアリングベクトル $d(\theta)$ および標本共分散 R に対して、最小分散無歪応答 (MVDR) 複素重みベクトル w は

$$[H]w_{\text{MVDR}} = \frac{R^{-1}d}{d^H R^{-1}d}, \quad (243)$$

ここで d^H はエルミート転置である。MVDR は視線方向で利得 1 という制約の下、出力分散を最小化し、遠方話者の信号対雑音比を改善する。

- 音声活動検出 (VAD) およびウェイクワード検出は、継続的なクラウドストリーミングからの誤トリガを制限するため、オンデバイスで実行すべきである。

音声認識および意図マッピング。

- ハイブリッドアーキテクチャを使用：オンデバイス VAD およびキーワードスポッティング；信頼度が低い場合に限り、完全な自動音声認識 (ASR) をエッジサーバにオフロードする。
- ASR 出力 u を離散的な動作仮説 a にマッピングする。確率的意図分類器を使用し、

$$[H]a^* = \arg \max_a P(a | u). \quad (244)$$

を実行する。

- 誤分類リスクを損失認識決定規則で考慮する。損失行列 $L(a, \hat{a})$ および事後確率 $P(\hat{a} | u)$ に対して、期待損失を最小化する動作を選択する：

$$[H]a_{\text{opt}} = \arg \min_{\hat{a}} \sum_a L(a, \hat{a}) P(\hat{a} | u). \quad (245)$$

実装ブループリント。

- モジュラーパイプライン：

1. マイクロホンアレイビームフォーミングおよび DOA (到来方向) 推定。
2. オンデバイス VAD およびウェイクワード検出。
3. 小語彙コマンドおよび緊急フレーズ用のオンデバイス ASR。
4. 長い発話のためのエッジ／クラウド ASR フォールバック (暗号化トランスポート)。
5. 信頼度スコアおよび安全チェックサブシステムを備えた意図分類器。
6. ビヘイビアツリーセーフガードによる動作仲裁。

- プロセス間メッセージングには ROS2 を、ビヘイビアオーケストレーションには GROOT を使用する。ロボットは、信頼度が閾値を超えるか、リスクの高い動作に対して人間の確認ステップが満たされた場合にのみコマンドを実行する。

実践例：倉庫ヒューマノイドが「pick up box」を受け付け、複数の話者および機械が近くで稼働している状況。ビームフォーミングがオペレータを定位し、ASR 精度を改善する。事後確率 $P(\text{pick_up} | u) = 0.62$ でも、動いているコンベア上での誤ピックの損失が高い場合、(3) のリスク認識規則により確認を要求できる。

計算ノート。

- 遅延予算：各ステージに予算を割り当てる。例：
 - ビームフォーミングおよび VAD： ≤ 20 ms。

– オンデバイス ASR：モデルに応じて 50–150 ms。

– エッジ ASR 往復：100–300 ms。

より厳格な安全臨界制御は、オンデバイス推論に依存すべきである。

- モデル選択：オンデバイス ASR には、小型の再帰または Transformer 量子化モデルを使用する。エッジでは精度向上のため、より大きなモデルを使用する。

制御ノードコードスニペット例：簡略化された ROS2 スタイルパイプラインおよびセーフティゲーティングを示す：

コードサンプル 104 簡略化された音声コマンドパイプラインノード

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from audio_msgs.msg import AudioMsg
from geometry_msgs.msg import Twist
from std_msgs.msg import String
import torch
import numpy as np
from threading import Lock
import time

class VoiceCmdNode(Node):
    def __init__(self):
        super().__init__('voice_cmd')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            durability=DurabilityPolicy.VOLATILE,
            depth=1
        )
        self.sub_audio = self.create_subscription(
            AudioMsg, '/mics/beamed', self.on_audio, qos)
        self.pub_cmd = self.create_publisher(Twist, '/robot/cmd', qos)
        self.pub_status = self.create_publisher(String, '/voice/status', qos)

        self.lock = Lock()
        self.intent_model = torch.jit.load('/opt/models/intent.pt').eval()
        self.vad = torch.jit.load('/opt/models/vad.pt').eval()
        self.safety = SafetyChecker(self.get_logger())
        self.pending = None
        self.timer = None
```

```

def on_audio(self, msg):
    with self.lock:
        audio = np.frombuffer(msg.data, dtype=np.float32)
        if not self.vad(torch.from_numpy(audio)).item():
            return
        text, conf = self.asr(audio)
        if conf < 0.6:
            text, conf = self.edge_asr(audio)
        action, prob = self.classify(text)
        if prob < 0.7 or not self.safety.check(action):
            self.ask_confirm(action)
        return
        self.publish_cmd(action)

def asr(self, audio):
    # オンデバイスASR
    with torch.no_grad():
        logits = self.intent_model.encode(audio)
        return logits.argmax(-1).item(), logits.softmax(-1).max().item()

def edge_asr(self, audio):
    # 暗号化エッジ呼び出し（簡略化）
    return "unknown", 0.0

def classify(self, text):
    # 意図分類
    with torch.no_grad():
        out = self.intent_model(text)
        prob, idx = out.softmax(-1).max(-1)
        return idx.item(), prob.item()

def ask_confirm(self, action):
    self.pending = action
    self.timer = self.create_timer(3.0, self.timeout)
    self.pub_status.publish(String(data=f"confirm:{action}"))

def timeout(self):
    self.pending = None
    self.destroy_timer(self.timer)

```

```

def publish_cmd(self, action):
    twist = Twist()
    twist.linear.x = action.get('vx', 0.0)
    twist.angular.z = action.get('wz', 0.0)
    self.pub_cmd.publish(twist)

class SafetyChecker:
    def __init__(self, logger):
        self.logger = logger

    def check(self, action):
        # 速度制限チェック
        if abs(action.get('vx', 0)) > 1.0 or abs(action.get('wz', 0)) > 1.5:
            self.logger.warn("速度制限超過")
            return False
        return True

def main(args=None):
    rclpy.init(args=args)
    node = VoiceCmdNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

```

評価指標およびテスト。

- 環境を横断した精度、リコール、誤動作率を報告する。
- 負荷下でのタイムトゥアクションおよび最悪遅延を測定する。
- リプレイ攻撃およびボイスインパーソネーションを含むアドバーサルコマンドテストを実施する。

設計トレードオフおよび運用リスク。

- 精度 vs 遅延：オフロードは精度を向上させるが、遅延とネットワーク依存を追加する。
- プライバシー vs 機能：ローカル処理はプライバシーを保持する；クラウドは性能を向上させる。
- 誤動作：ウェイクワード閾値は、応答性と迷惑トリガのバランスを取る必要がある。
- 安全リスク：意図しない物理動作には、層状チェック、明示的確認、キルスイッチ機構が必要である。

- ・環境頑健性：マイクロホン配置、振動分離、定期的なキャリブレーションが不可欠である。

具体的なエンジニアリングインプリケーション：不可逆効果を持つすべての動作に対して、オンデバイスキーワードスポッティングおよび安全チェックを優先する。測定可能なリスク予算を維持し、通信またはセンシング劣化下での有害動作を防ぐフォールバックを設計する。ターゲット環境での継続的フィールドテストは、運用ノイズおよび人間のばらつき下での ASR 性能、ビームフォーミング利得、意図仲裁を検証するために必須である。

31 高度な相互作用手法

31.1 ロボティクスにおける自然言語処理

HRI の基礎と音声コマンドのケーススタディを踏まえ、本小節ではヒューマノイドロボットに特化した実用的な自然言語処理（NLP）手法を検討する。頑健な意図推定、グラウンディングされた言語理解、物理的インタラクションに適した低遅延対話制御に焦点を当てる。

問題定義. ヒューマノイドロボットは騒音が多く動的な環境で動作する。音声コマンドを解釈し、物体や動作への参照を曖昧さなく理解し、不確実性の下で安全なアクチュエータ応答を決定しなければならない。主要な工学要件は以下の通りである：

- ・リアルタイム推論（自然なターンタキングのための遅延予算はしばしば < 200 ms）、
- ・音声認識エラーや曖昧な表現への頑健性、
- ・知覚とアフォーダンスへの言語のグラウンディング、
- ・曖昧またはリスクの高い要求に対する安全なフォールバック。

技術分析. 実用的なパイプラインは、自動音声認識（ASR）、自然言語理解（NLU）、対話管理、言語グラウンディングの責任を分割する。NLU をトークン列から意図、スロット、または対話行為への写像と定義する。対話管理は、不確実性と時間依存性を扱うために部分観測マルコフ決定過程（POMDP）として定式化される。

1. 確率的意図推定. ASR の不確実性を考慮するため、認識仮説を周辺化して意図事後確率を計算する：

$$[H]P(\text{intent} \mid \text{audio}, C) = \sum_h P(\text{intent} \mid h, C) P(h \mid \text{audio}), \quad (246)$$

ここで h は ASR 仮説、 C は文脈状態（以前の対話、センサ読み取り値）を示す。この式により、信頼度の低い ASR 単語は全体の意図確実性を低下させる。

2. 対話状態を POMDP として. 潜在対話状態 s 、ロボット行動 a 、観測 o （NLU 出力を含む）を表現する。信念更新は

$$[H]b_t(s') = \eta P(o_t \mid s') \sum_s P(s' \mid s, a_{t-1}) b_{t-1}(s), \quad (247)$$

正規化 η を伴う。この更新を用いて、ロボットは対話状態の明示的な確率分布を維持する。その後、方針はその分布の下で期待効用を最大化できる。

3. ニューラルエンコーダと分類. Transformer エンコーダは、発話と文脈の密な表現 x を生成する。

ソフトマックスを持つ線形分類器は意図確率を生成する：

$$[H]P(y = k | x) = \frac{\exp(w_k^\top x + b_k)}{\sum_j \exp(w_j^\top x + b_j)}. \quad (248)$$

実用的なシステムは、遅延と電力予算に合わせるために微調整された軽量 Transformer または蒸留モデルを使用する。

4. グラウンディングと記号写像. グラウンディングは、言語トークンから知覚エンティティへの確率的写像 g である。トークン w と知覚状態 P に対して：

$$[H]P(\text{obj} | w, P) = \text{softmax}(f(w), \phi(P)), \quad (249)$$

ここで $f(w)$ は語彙埋め込み、 $\phi(P)$ は知覚特徴（バウンディングボックス、アフォーダンス、アフォーダンススコア）を示す。これにより、「赤いカップを拾って」といったコマンドが特定の物体仮説に写像される。

実装スケッチ. リストは、ASR 出力、Transformer NLU、POMDP 信念更新を統合した最小限の ROS2 スタイルノードを示す。展開の詳細は省略するが、運用フローを強調する。

コードサンプル 105 信念更新を備えた最小限の音声→意図パイプライン

```
#!/usr/bin/env python3
import os
import threading
from typing import List, Tuple

import numpy as np
import rclpy
from rclpy.node import Node
from std_msgs.msg import String, Float32MultiArray

from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch

# 意図数とラベル
NUM_INTENTS = 5
INTENT_LABELS = ["goto", "pick", "stop", "status", "fallback"]

# GPU/CPU 自動選択
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class IntentRecognizer(Node):
    def __init__(self) -> None:
        super().__init__("intent_recognizer")
```

```

# モデル読み込み（一度だけ）
model_name = "distilbert-base-uncased"
self.tokenizer = AutoTokenizer.from_pretrained(model_name)
self.model = AutoModelForSequenceClassification.from_pretrained(
    model_name, num_labels=NUM_INTENTS
).to(DEVICE).eval()

# 信念と遷移行列の初期化
self.belief: np.ndarray = np.ones(NUM_INTENTS) / NUM_INTENTS
self.trans_matrix: np.ndarray = np.eye(NUM_INTENTS)
# 単位行列で簡略化

# ロック付き共有変数
self._lock = threading.Lock()

# ROS 2 購読・配信
self.sub = self.create_subscription(
    String, "/asr_hypotheses", self._asr_cb, 10
)
self.pub = self.create_publisher(Float32MultiArray, "/intent_belief", 10)

def _intent_probs_from_text(self, text: str) -> np.ndarray:
    """1文から意図確率を返す"""
    with torch.no_grad():
        inputs = self.tokenizer(
            text, return_tensors="pt", truncation=True, max_length=128
        ).to(DEVICE)
        logits = self.model(**inputs).logits[0].cpu().numpy()
    # ソフトマックス
    exp = np.exp(logits - np.max(logits))
    return exp / exp.sum()

def _aggregate_intent(self, hyps: List[Tuple[str, float]]) -> np.ndarray:
    """ASR仮説リストから統合意図分布を計算"""
    agg = np.zeros(NUM_INTENTS)
    for h_text, p_h in hyps:
        agg += self._intent_probs_from_text(h_text) * p_h
    return agg / (agg.sum() or 1.0)

```

```

def _belief_update(self, obs_probs: np.ndarray) -> None:
    """POMDP信念更新（予測→観測更新→正規化）"""
    with self._lock:
        pred = self.trans_matrix.T @ self.belief
        new_b = obs_probs * pred
        self.belief = new_b / (new_b.sum() or 1.0)

def _asr_cb(self, msg: String) -> None:
    """/asr_hypotheses"␣コールバック"""
    # 簡易フォーマット: "text1,score1;text2,score2;..."
    try:
        hyps = [
            (part.split(",")[0], float(part.split(",")[1]))
            for part in msg.data.split(";")
            if part
        ]
    except (IndexError, ValueError):
        self.get_logger().warn("Invalid␣ASR␣format")
        return

    obs = self._aggregate_intent(hyps)
    self._belief_update(obs)

    # 配信
    out = Float32MultiArray()
    out.data = self.belief.tolist()
    self.pub.publish(out)

def main(args=None):
    rclpy.init(args=args)
    node = IntentRecognizer()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()

```

設計上の考慮事項とトレードオフ.

- 遅延 vs 精度: 大きなモデルは精度を向上させるが、推論時間と消費電力を増加させる。モデル蒸留、プルーニング、またはオンデバイス量子化を使用する。
- データ効率: ドメイン固有の発話とスクリプト化された対話を収集し、産業タスクの失敗率を削減する。
- 安全性とフォールバック方針: 確認クエリまたは非執行的確認のような、高い不確実性に対する安全なデフォルトを指定する。
- グラウンディングエラー: 誤陽性グラウンディングは有害な行動を引き起こす可能性がある。行動リスクが閾値を超える場合は多モーダル確認を使用する。

運用上のリスクと影響.

- ASR エラー下での過信 NLU は危険なアクチュエータコマンドを引き起こす可能性がある。常に信頼度閾値を組み込む。
- 展開文脈が変化するとモデルドリフトが発生する。継続的なデータ収集と増分再学習を計画する。
- プライバシー規制は音声データの保持を制約する。オンデバイス処理または安全で同意済みのクラウドパイプラインを設計する。

具体的な工学上の影響.

- 2 層モデルを展開する: 迅速な意図/確認決定のための小さなオンデバイスエンコーダ、複雑な計画または冗長な対話のための大きなクラウドモデル。
- 各モジュールの遅延予算を定量化し、リアルタイム制約を満たすように知覚パイプラインを設計する。
- グラウンディングと意図の曖昧さを軽減するため、高リスク行動に対して明示的な検証プロトコルを実装する。

31.2 感情認識

言語ベースの対話を基盤に、感情認識は言語コマンドに感情的な文脈を付加し、適応的な応答に情報を提供する。多モーダルな手がかりは意図推定を強化し、曖昧な発話を解決する。

ロボティクスの問題定義。ヒューマノイドは社会的に適切な行動のために人間の感情状態をリアルタイムで推定しなければならない。入力には通常以下が含まれる:

- ステレオまたは RGB カメラからの顔画像、
- マイクロホンアレイによる音声とプロソディ、
- 深度/姿勢推定からの身体姿勢、
- 高リスク環境ではオプションの生理信号 (PPG、EDA)。

技術的な目標は精度、人間の知覚閾値以下の遅延、説明可能性、プライバシーに配慮したセンシングである。

技術的分析。2 つの典型的な出力表現は離散的なカテゴリラベル (例: happy, sad, angry, neutral)

とバレンス・アラウサル座標のような連続的な感情空間である。連続表現はより滑らかな制御ポリシーを可能にする；カテゴリラベルは GROOT 内の決定木およびビヘイビアツリーを簡素化する。

主要なモデリング選択と公式：

1. モダリティごとの埋め込み抽出。 x_f, x_s, x_p を顔、音声、姿勢の入力とする。埋め込み関数 f_f, f_s, f_p はモダリティを固定長ベクトルに写像する：

$$[H]z_f = f_f(x_f), \quad z_s = f_s(x_s), \quad z_p = f_p(x_p). \quad (250)$$

実装： f_f は軽量 CNN バックボーン（MobileNetV3 または剪定済み ResNet18）、 f_s はスペクトログラム CNN に小さな時間モジュールを続け、 f_p は関節角度またはキーポイント差分に対する小さな MLP を用いる。

2. 融合戦略。アーリー融合は埋め込みを連結；レイト融合はモダリティ予測を集約する。実用的な学習可能な中間融合は注意重み α_m を用いた重み付き和を用いる：

$$[H]z = \sum_{m \in \{f,s,p\}} \alpha_m(z_m) z_m, \quad \alpha_m = \frac{\exp(w_m^\top z_m)}{\sum_k \exp(w_k^\top z_k)}. \quad (251)$$

最終的な感情確率は分類器 g から導出される：

$$[H]\hat{y} = \text{softmax}(Wg(z) + b), \quad (252)$$

ここで \hat{y} はクラス確率のベクトルである。

3. 確率的融合の代替は条件付き独立性の下でベイズ則を用いる：

$$[H]P(E | X) = \frac{P(E) \prod_m P(X_m | E)}{\sum_{E'} P(E') \prod_m P(X_m | E')}. \quad (253)$$

これは解釈可能な尤度を提供するがモダリティ尤度モデルを必要とする。

リアルタイム制約。全パイプライン遅延 T_{total} は $T_{\text{total}} \leq T_{\text{max}}$ を満たし、通常 $T_{\text{max}} \approx 300$ ms が応答性のある顔反応のための目安である。遅延を分解：

$$[H]T_{\text{total}} = T_{\text{acq}} + T_{\text{pre}} + T_{\text{inf}} + T_{\text{comm}}, \quad (254)$$

フレームスキップ、モデル量子化、Xavier/Jetson でのエッジ推論により各項を最適化する。

実装スケッチ。以下の PyTorch ベースのリストは最小限の多モーダル融合推論ノードを示す。顔と音声の学習済み埋め込みと小さな MLP 融合ヘッドを前提とする。ノードはオンボード GPU 向けに設計され、選択されたラベルを ROS2 トピックにパブリッシュする。

コードサンプル 106 最小限の多モーダル感情融合ノード（推論のみ）

```
import os
import time
from typing import Tuple

import numpy as np
import torch
```

```

import torch.nn as nn
import rclpy
from rclpy.node import Node
from std_msgs.msg import Int32, Float32MultiArray

# --- モデル定義 ---
class FusionHead(nn.Module):
    def __init__(self, emb_dim: int = 512, hidden: int = 128, n_classes: int = 6):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(emb_dim, hidden),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Linear(hidden, n_classes)
        )

    def forward(self, z: torch.Tensor) -> torch.Tensor:
        return self.mlp(z)

# --- 推論ラッパー ---
class MultimodalInference:
    def __init__(self,
                 face_path: str,
                 speech_path: str,
                 fusion_path: str,
                 device: str = "cuda"):
        self.device = torch.device(device if torch.cuda.is_available() else "cpu")

        # TorchScriptモデルを読み込み
        self.face_embedder = torch.jit.load(face_path, map_location=self.device).eval()
        self.speech_embedder = torch.jit.load(speech_path, map_location=self.device).eval()

        # FusionHeadを構築し重みを読み込み
        self.fusion = FusionHead(emb_dim=512, hidden=128, n_classes=6).to(self.device)
        self.fusion.load_state_dict(torch.load(fusion_path, map_location=self.device))
        self.fusion.eval()

    @torch.no_grad()
    def infer(self, face_frame: torch.Tensor, audio_buffer: torch.Tensor) -> Tuple[int, float]:
        # 入力をデバイスへ移動

```

```

face_frame = face_frame.to(self.device, non_blocking=True)
audio_buffer = audio_buffer.to(self.device, non_blocking=True)

zf = self.face_embedder(face_frame)
zs = self.speech_embedder(audio_buffer)
z = torch.cat([zf, zs], dim=-1) # 結合
logits = self.fusion(z)
probs = torch.softmax(logits, dim=-1).cpu().numpy()
label = int(np.argmax(probs))
return label, probs

# --- ROS 2 ノード ---
class MultimodalNode(Node):
    def __init__(self):
        super().__init__("multimodal_inference")
        self.declare_parameter("face_model_path", "face_embedder.pt")
        self.declare_parameter("speech_model_path", "speech_embedder.pt")
        self.declare_parameter("fusion_model_path", "fusion_head.pt")

        face_path = self.get_parameter("face_model_path").value
        speech_path = self.get_parameter("speech_model_path").value
        fusion_path = self.get_parameter("fusion_model_path").value

        self.inference = MultimodalInference(face_path, speech_path, fusion_path)

# サブスクライバ／パブリッシャ
self.face_sub = self.create_subscription(
    Float32MultiArray, "/face_tensor", self.face_cb, 1)
self.audio_sub = self.create_subscription(
    Float32MultiArray, "/audio_tensor", self.audio_cb, 1)
self.label_pub = self.create_publisher(Int32, "/predicted_label", 1)
self.prob_pub = self.create_publisher(Float32MultiArray, "/predicted_probs", 1)

self.face_tensor = None
self.audio_tensor = None

def face_cb(self, msg: Float32MultiArray):
    self.face_tensor = torch.tensor(msg.data, dtype=torch.float32).unsqueeze(0)
    self.try_infer()

```

```

def audio_cb(self, msg: Float32MultiArray):
    self.audio_tensor = torch.tensor(msg.data, dtype=torch.float32).unsqueeze(0)
    self.try_infer()

def try_infer(self):
    if self.face_tensor is None or self.audio_tensor is None:
        return
    label, probs = self.inference.infer(self.face_tensor, self.audio_tensor)
    self.label_pub.publish(Int32(data=label))
    self.prob_pub.publish(Float32MultiArray(data=probs.flatten().tolist()))

def main(args=None):
    rclpy.init(args=args)
    node = MultimodalNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()

```

データと学習に関する考察。文化的小および人口統計学的バイアスを避けるため多様なデータセットを用いる。Isaac Sim からのシミュレートされた顔データと実世界コーパスを組み合わせ、遮蔽と照明を増強する。敵対的損失または特徴アライメントによるドメイン適応を適用し、シミュレーションからハードウェアへの信頼性の高い転送を実現する。

評価指標と運用テスト：

- 人口統計グループにわたるバランスの取れた精度、
- 遅延パーセンタイル（95 番目）、
- シナリオごとの混同行列（会話、高モーション）。

遮蔽、複数話者干渉、ヘッドホン使用のような故障モードを明らかにするため、制御された環境と生態学的環境の両方でテストする。

設計上のトレードオフとリスク：

- プライバシー：継続的な音声/映像キャプチャは機密データ漏洩のリスクがある。オンデバイス推論、選択的バッファリング、同意に基づくキャプチャ方針を実装する。
- 誤検出/見逃し：誤分類は不適切なロボット応答を生み、信頼または安全性を損なう。保守的な動作閾値と確認対話を用いる。

- ・リソース制約：重いモデルは精度を向上させるが電力および熱予算を超える。量子化された小さなバックボーンと、最初に粗い分類器を実行するカスケードモデルを優先する。
- ・倫理的利用：雇用またはケアに影響する自動プロファイリングまたは決定を人間の監視なしで行うことを避ける。

ヒューマノイドシステムへの運用上の影響：

- ・GROOT 内のビヘイビアツリーに感情出力を信関値を伴うソフト述語として統合する。
- ・連続的なバレンス・アラウサル出力を滑らかな姿勢および音声適応に用いる。
- ・監査および安全性レビューをサポートするため、説明可能性とロギングを優先する。

全体として、ヒューマノイドのための堅牢な感情認識は、多モーダル融合、エッジ対応アーキテクチャ、慎重なデータセット運用、安全性とユーザ信頼を維持するための運用上の保護策を要する。

31.3 ジェスチャベースの制御

ジェスチャベースの制御は、人型ロボットが実環境で人間の身体動作を確実に解釈できるようにする実用的な課題に対処する。要件には、低遅延認識、安全上重要なコマンドに対する高精度、遮蔽下でのグレースフルな劣化、ジェスチャとロボット動作の明確な対応関係が含まれる。エンジニアリングの目的は、連続的なマルチモーダルセンサストリームを、遅延が制限され予測可能な故障モードを持つ離散的で実行可能な意図に変換することである。

技術解析は、センシングとセグメンテーション、表現と分類、コマンド仲裁と平滑化の 3 段階で進める。

1. センシングとセグメンテーション

- ・深度カメラ、ステレオビジョン、ウェアラブルデバイスの慣性計測装置（IMU）をセンサ融合により組み合わせる。融合は遮蔽や照明変化に対する頑健性を向上させる。
- ・時間的セグメンテーションは連続運動からジェスチャ候補を切り出す。一般的な手法は、関節速度 $v_i(t)$ から運動エネルギー信号 $M(t)$ を計算し、

$$M(t) = \sum_{i=1}^n \|v_i(t)\|_2^2,$$

として $M(t)$ が活動閾値を超えたときにセグメンテーションをトリガする。適応的閾値は雑多な環境での誤検出を減らす。

2. 表現と分類

- ・正規化された関節座標、相対角度、時間微分を用いてジェスチャを表現する。胴体位置と肩間距離で正規化しスケール不変性を達成する。
- ・PCA またはオートエンコーダで空間データを圧縮し、姿勢構造を保ちながら次元削減する。
- ・時間的ダイナミクスを確率モデルまたは深層ネットワークでモデル化する：
 - 隠れマルコフモデル（HMM）は構造化された短いジェスチャで学習データが限られている場合に有効。
 - Long Short-Term Memory（LSTM）ネットワークはより豊富なジェスチャ語彙にスケールし可変時間を扱う。
- ・分類器出力を各ジェスチャクラス k に対する信頼度スコア p_k に変換する。ソフトマックス

を用いてネットワークのロジット z_k から確率を生成する：

$$[H]p_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}. \quad (255)$$

決定規則 $p_k \geq p_{th}$ を適用してコマンドを受理する。安全上重要な動作には p_{th} を保守的に設定する。

3. コマンド仲裁と平滑化

- ・認識されたジェスチャを GROOT のビヘイビアツリーアクションまたは ROS2 アクションサーバにマッピングし動作実行する。
- ・時間的平滑化を導入し振動的コマンド列を回避する。クラススコアに対する指数移動平均はちらつきを減らす：

$$[H]\hat{p}_k(t) = \alpha p_k(t) + (1 - \alpha)\hat{p}_k(t - 1), \quad (256)$$

ここで $\alpha \in (0, 1]$ は応答性と安定性のトレードオフである。

- ・不可逆動作には短い確認ウィンドウを実装し、時間ウィンドウ内での繰り返し認識または音声・注視によるマルチモーダル確認を要求する。

実装例：ROS2 互換 Python ノードは骨格姿勢を購読し高レベルジェスチャコマンドをパブリッシュする。以下のリストはセグメンテーション、分類呼び出し、信頼度フィルタリング、ビヘイビアトリックへのパブリッシュを示す。

```
import rclpy from rclpy.node import Node from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import String from typing import List, Dict, Tuple import numpy as np import threading import time

class GestureRecognizer(Node):
    def __init__(self) -> None:
        super().__init__(self, 'gesture_recognizer')

        qos = QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT, history=HistoryPolicy.KEEP_LAST, depth=10)

        self.sub = self.create_subscription(String, 'skeleton_stream', self.cb_skeleton, qos)
        self.pub = self.create_publisher(String, 'gesture_cmd', 10)

        self.buffer: List[np.ndarray] = []
        self.lock = threading.Lock()

        self.motion_enerresh = 0.02
        self.p_th = 0.85
        self.alpha = 0.3
        self.smoothed: Dict[str, float] = {}

        self.timer = self.create_timer(0.1, self.process_buffer)  # 10Hz 処理

        def cb_skeleton(self, msg: String) -> None:
            pose = self.deserialize(msg.data)
            with self.lock:
                self.buffer.append(pose)

        def process_buffer(self) -> None:
            with self.lock:
                if len(self.buffer) < 2:
                    return
            buf = self.buffer.copy()

            if self.motion_energy(buf) > self.motion_enerresh:
                pred, conf = self.classify(buf)
                self.smoothed[pred] = self.alpha * conf + (1 - self.alpha) * self.smoothed.get(pred, 0.0)
                if self.smoothed[pred] >= self.p_th:
                    out = String()
                    out.data = pred
                    self.pub.publish(out)
                    with self.lock:
                        self.buffer.clear()

            def motion_energy(self, buf: List[np.ndarray]) -> float:
                if len(buf) < 2:
                    return 0.0
                diffs = np.diff(np.array(buf), axis=0)
                return float(np.mean(np.sum(diffs**2, axis=1)))

            def classify(self, buf: List[np.ndarray]) -> Tuple[str, float]:
                実際のモデル推論に置き換える
                return ("wave", 0.9)
```

```
def deserialize(self, s: str) -> np.ndarray: return np.fromstring(s, sep=',')
def main(args=None): rclpy.init(args=args) node = GestureRecognizer() try: rclpy.spin(node) except
KeyboardInterrupt: pass finally: node.destroy_node() rclpy.shutdown()
```

運用上の考慮事項とエンジニアリングのトレードオフ：

- ・遅延対精度：時間的平滑化（ α を小さく）を強めると安定性が向上するが遅延が増える。タスクの重要性に応じて α を設定する。
- ・語彙サイズ：ジェスチャセットが大きくなると学習データが増え誤分類リスクが高まる。安全タスクではコンパクトで区別しやすい語彙を優先する。
- ・センサ配置：頭部搭載深度センサは胴体・手の視認性を向上させるが、システム複雑性とメンテナンスが増大する。
- ・マルチモーダル確認：高リスクコマンドには注視または言語確認を要求し、誤検出を軽減する。
- ・環境リスク：動的な群衆、反射面、悪条件照明はビジュアルトラッキングを劣化させる。信頼度が低い場合は安全なスタンバイへ移行するなどフォールバック動作を設計する。

人型ロボットへのジェスチャベース制御の導入は、アクセシビリティと状況適応性を向上させる。エンジニアは、閾値処理、マルチモーダル融合、保守的な動作仲裁を通じて、応答性、頑健性、安全性をバランスさせる必要がある。

32 HRI の応用

32.1 医療におけるロボット

直感的なインタラクション設計とマルチモーダル感情認識に関する先行議論を踏まえ、本小節ではそれらの原則を具体的な医療タスクへ適用する。ユーザ中心のインタラクションモードと感情認識センシングを、安全で効果的なヒューマノイド支援のためのエンジニアリングソリューションへ結びつける。

問題定義. 医療におけるヒューマノイドロボットは、患者の安全・尊厳・臨床ワークフローを維持しながらタスクを遂行しなければならない。典型的なタスクは以下の通り：

- ・歩行訓練中の誘導的移動支援；
- ・手術室における器具受け渡し；
- ・ベッドサイドモニタリングと服薬リマインダ；
- ・認知療法と孤立緩和のための社会的エンゲージメント。

各タスクは、信頼できる知覚、拘束された運動、透明な意思決定、規制レベルの安全性を要求する。

技術分析. 設計を支配する 2 つの相互作用サブシステムは、インタラクション品質（信頼、快適性、透明性）と運動安全性（衝突回避、荷重取り扱い）である。インタラクション品質を、接近・操作中に制御スタックが最小化できる複合快適性指標 C で定量化する：

$$[H]C = w_d \cdot \frac{1}{1+d} + w_g \cdot (1-G) + w_e \cdot E, \quad (257)$$

ここで d は患者までの距離、 $G \in [0, 1]$ は患者とロボット間の正規化視線一致度、 $E \in [0, 1]$ は感情センシングから推定された情緒的不快度、 w_i はタスク重み付き係数である。 C が小さいほど快適性が

高い。

運動安全性のため、制御バリア関数 (CBF) を課す。 $h(x)$ を符号付き安全余裕とし、安全な場合に正とする。標準的な CBF 制約は安全集合の前方不変性を保証する：

$$[H]\dot{h}(x, u) + \alpha(h(x)) \geq 0, \quad (258)$$

ここで u は制御入力、 α はクラス \mathcal{K} 関数、通常 $\alpha(s) = \kappa s$ で $\kappa > 0$ 。制御レートで二次計画 (QP) を解くことで、快適性コストまたはタスク誤差も最小化する最も安全な u を得る。

実装パターン。知覚、インタラクションモデル、CBF 制約付き制御を層状アーキテクチャへ統合する：

1. 知覚層：

- ・プロクセミクスと姿勢のための深度・RGB；
- ・潜在疼痛検出のための熱・接触センサ；
- ・対話と緊急キーワードのための音声。

2. インタラクション層：

- ・式 (1) を用いて C を計算；
- ・意図認識とポリシー選択を実行 (ビヘイビアツリーまたは RL フォールバック)。

3. 制御層：

- ・CBF を用いた QP ベース逆運動学 (式 (2))；
- ・器具受け渡しのためのトルク・接触制約を課す。

実践的アルゴリズム。全身制御のために 100–200 Hz で高速 QP ソルバを用いる。目的関数はタスク追従と快適性最小化をブレンドする：

$$\min_u \|Ju - v_{\text{des}}\|^2 + \lambda_C C(u)$$

ただし

$$\dot{h}(x, u) + \kappa h(x) \geq 0, \quad \text{torque_limits, joint_limits, contact_constraints.}$$

ここで J はタスクヤコビアン、 v_{des} は所望タスク速度、 λ_C はバランス係数。重み選択は応答性と患者快適性に影響する。

現実的例：歩行支援。ロボットは先導速度に追従しながら安全余裕を維持しなければならない。横方向隔離を d_{lat} 、ロボット速度を v_r とする。横方向速度に対し、快適性に比例した軟制約を実装する：

$$v_{\text{lat}}^{\text{max}} = v_0 \cdot \exp(-\beta d_{\text{lat}}),$$

隔離が減少すると横方向運動を削減する。

コードスニペットは、プロクセミック快適性を CBF 制約付き速度指令へ融合する ROS2 互換 Python ノードを示す。リストは知覚ノードが人間の姿勢と感情推定を公開することを前提とする。

コードサンプル 107 ROS2 ノード：プロクセミック CBF 速度コントローラ

```
import rclpy
from rclpy.node import Node
```

```

from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from geometry_msgs.msg import Twist
from std_msgs.msg import Float32
import numpy as np
from typing import Optional
import osqp
from scipy import sparse

```

```

class ProxemicCBF(Node):
    def __init__(self) -> None:
        super().__init__('proxemic_cbf')

        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )

        self.sub_pose = self.create_subscription(
            Float32, 'human_distance', self.pose_cb, qos)
        self.sub_affect = self.create_subscription(
            Float32, 'human_discomfort', self.affect_cb, qos)

        self.pub_cmd = self.create_publisher(Twist, 'cmd_vel', qos)

        self.timer = self.create_timer(0.01, self.control_loop)

        self.d: float = 2.0
        self.e: float = 0.0
        self.v_max: float = 1.0
        self.v_min: float = 0.0
        self.d_safe: float = 0.6
        self.kappa: float = 3.0
        self.lambda_c: float = 0.5

    def pose_cb(self, msg: Float32) -> None:
        self.d = msg.data

    def affect_cb(self, msg: Float32) -> None:

```

```

        self.e = msg.data

def control_loop(self) -> None:
    v_des = 0.5
    C = 1.0 / (1.0 + self.d) + self.e
    h = self.d - self.d_safe

    # OSQPで解くQP:  $\frac{1}{2} u^T P u + q^T u \quad \text{s.t.} \quad l \leq A u \leq u$ 
    P = sparse.csc_matrix([[1.0 + self.lambda_c * C]])
    q = np.array([-v_des])
    A_sparse = sparse.csc_matrix([[ -1.0]])
    l = np.array([-np.inf])
    u = np.array([-self.kappa * h])

    prob = osqp.OSQP()
    prob.setup(P, q, A_sparse, l, u, verbose=False)
    res = prob.solve()

    if res.info.status != 'solved':
        self.get_logger().warn('QP failed')
        return

    u_opt = np.clip(res.x[0], self.v_min, self.v_max)

    cmd = Twist()
    cmd.linear.x = float(u_opt)
    self.pub_cmd.publish(cmd)

def main() -> None:
    rclpy.init()
    node = ProxemicCBF()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

```
if __name__ == '__main__':
    main()
```

設計上の考慮とトレードオフ：

- ・知覚遅延はコントローラの実行可能性を低下させる；プロクセミックループではエンドツーエンド遅延を 100 ms 未満を目指す。
- ・重みチューニング (w_i , λ_C , κ) はタスク性能と知覚快適性をトレードオフする。
- ・過剰保守的 CBF は緊急時のタスク完了を妨げる；モード依存緩和と臨床家オーバーライドを実装する。
- ・感情センシング使用时，プライバシーとデータ取り扱いが重要；ストリームを暗号化し，最小特微量のみを保存する。

運用上のリスクと緩和策：

- ・感情信号の誤分類は不適切な停止または接近を引き起こす；マルチモーダル融合と信頼度閾値で緩和する。
- ・受け渡し中の物理接触は患者への傷害リスクを伴う；コンプライアントアクチュエータ，リアルタイム力センシング，意図的両手把持を用いる。
- ・規制適合には文書化された危害分析，フェイルセーフ停止，検証済み制御ソフトウェアが必要。

エンジニアリングへの影響：感情認識プロクセミクスをリアルタイム制御ループへ統合する。快適性指標を臨床的に検証する。応答性，頑健性，プライバシーをバランスさせ，医療環境における安全で受け入れ可能なヒューマノイド挙動を実現する。

32.2 教育用ヒューマノイドシステム

医療ロボットにおける社会的合図と安全性に関する前節の議論を踏まえ、教育ヒューマノイドは同様の知覚・対話戦略を用いて学習を足場立てし、指導を個別化し、教室の安全を維持する。これらのシステムは、教育学的目標、リアルタイム適応、騒がしい人間環境における堅牢な安全制約のバランスを取る必要がある。

問題定義：コア STEM 概念を教え、個々の生徒に応じてレッスンペースを適応させ、教師と連携するヒューマノイドロボットを展開する。主要な技術目標は、信頼できる生徒状態推定、教育的行動のための方策選択、センサ駆動型安全性、教師にとって解釈可能な振る舞いである。

技術分析：

- ・生徒状態推定。生徒の潜在的知識とエンゲージメントを隠れ状態 s_t としてモデル化する。観測 o_t はマルチモーダルセンサから得られる：音声認識、視線追跡、顔感情、タスク遂行。再帰的ベイズフィルタは解釈可能な信念更新を提供する：

$$[H]P(s_t | o_{1:t}) \propto P(o_t | s_t) \sum_{s_{t-1}} P(s_t | s_{t-1})P(s_{t-1} | o_{1:t-1}). \quad (259)$$

この信念は離散確率ベクトルとして表現され、不確実性を意識した教育的選択を支援する。

- ・チュータリングのための方策選択。レッスン選択を逐次決定問題として扱う。各候補行動 a に対

して、期待教育的効用を計算する：

$$[H]U(a) = \sum_s P(s \mid o_{1:t}) R(a, s), \quad (260)$$

ここで $R(a, s)$ は状態 s を与えたときの期待学習効果とエンゲージメントコストを定量化する。安全制約の下で $\arg \max_a U(a)$ を選択する。

- 安全性とプロクセミクス。低レベルアドミタンスまたはインピーダンスコントローラを用いて安全距離とコンプライアントな対話を維持する。矢状距離 $d(t)$ に対して、PD コントローラが接近速度を調整できる：

$$[H]v_{\text{cmd}}(t) = K_p (d_{\text{desired}} - d(t)) + K_d (\dot{d}_{\text{desired}} - \dot{d}(t)). \quad (261)$$

受動的挙動を保証するため K_p, K_d に厳格な境界を設定する。力・トルクリミットと冗長接触検出を用いる。

- 振る舞いアーキテクチャ。高レベル教育のための記号的振る舞い木と、微小適応のための学習方策を組み合わせる。振る舞い木は教師に解釈可能性とフェイルセーフ出口を提供する。強化学習 (RL) はシミュレーションでパラメータを調整でき、木内の監視安全ノードによって制約が強制される。

実装（システム構成要素と技術制約）：

- センサ：ジェスチャと物体姿勢のためのステレオ RGB-D カメラ、話者定位のためのマイクアレイ、慣性安全監視のための IMU。各センサをキャリブレーションし、タイムスタンプ付きセンサバス（例：ROS2 QoS を低遅延に調整）で出力を融合する。
- シミュレーション優先トレーニング：Isaac Sim を用いてカリキュラム対話トレースを収集し、RL エージェントの報酬シェイピングを調整する。ドメインランダムマイゼーションとハードウェアインザループテストで転移を検証する。
- 教師インザループ制御：振る舞い木状態と生徒モデル信念を教師コンソールに公開し、介入とロギングを可能にする。

実用的アルゴリズムパターン（信念駆動型レッスン選択）。スニペットは離散信念を更新し、期待効用を計算し、選択された行動を公開する単純な ROS2 スタイルノードを実装する。プレースホルダを実際の知覚モジュールと R テンプレートに置き換える。

コードサンプル 108 信念駆動型レッスンセレクトノード（例示的）。

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from rcl_interfaces.msg import SetParametersResult
from typing import Dict, Any, Optional
import json
import numpy as np
from ament_index_python import get_package_share_directory
```

```

import os

from tutorial_interfaces.msg import SensorObs, LessonCmd
# プロジェクト固有のメッセージ

class LessonSelector(Node):
    """レッスン選択ノード：観測に基づくベイズ更新＋期待効用最大化"""

    def __init__(self) -> None:
        super().__init__('lesson_selector')

        # パラメータ宣言
        self.declare_parameter('likelihood_path', '')
        self.declare_parameter('transition_path', '')
        self.declare_parameter('reward_path', '')
        self.add_on_set_parameters_callback(self._param_cb)

        # QoS：センサ観測はベストエフォート、レッスン命令は信頼性重視
        sub_qos = QoSProfile(depth=10, reliability=ReliabilityPolicy.BEST_EFFORT)
        pub_qos = QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE)

        self._sub = self.create_subscription(
            SensorObs, 'sensor_obs', self._obs_cb, sub_qos)
        self._pub = self.create_publisher(LessonCmd, 'lesson_cmd', pub_qos)

        self._states = ['novice', 'intermediate', 'advanced']
        self._belief: Dict[str, float] = {s: 1.0 / len(self._states) for s in self._states}

        self._likelihood: Dict[str, Dict[str, float]] = {}
        self._transition: Dict[str, Dict[str, float]] = {}
        self._reward: Dict[str, Dict[str, float]] = {}

        self._load_models() # ファイル読み込み

    def _param_cb(self, params) -> SetParametersResult:
        """パラメータ変更時に再読み込み"""
        for p in params:
            if p.name.endswith('_path'):
                self._load_models()

```

```

        break
    return SetParametersResult(successful=True)

def _load_models(self) -> None:
    """JSONファイルから確率・報酬モデルを読み込む"""
    pkg_dir = get_package_share_directory('lesson_selector')
    try:
        with open(os.path.join(pkg_dir, self.get_parameter('likelihood_path').value), 'r') as f:
            self._likelihood = json.load(f)
        with open(os.path.join(pkg_dir, self.get_parameter('transition_path').value), 'r') as f:
            self._transition = json.load(f)
        with open(os.path.join(pkg_dir, self.get_parameter('reward_path').value), 'r') as f:
            self._reward = json.load(f)
        self.get_logger().info("モデル読み込み完了")
    except Exception as e:
        self.get_logger().error(f"モデル読み込み失敗: {e}")

def _obs_cb(self, msg: SensorObs) -> None:
    """観測到着時：ベイズ更新→行動選択→publish"""
    obs = msg.observation # string フィールドと仮定

    # 予測ステップ
    prior = {
        s: sum(self._transition[prev][s] * self._belief[prev] for prev in self._states)
        for s in self._states
    }

    # 更新ステップ
    posterior = {
        s: self._likelihood[s].get(obs, 1e-6) * prior[s]
        for s in self._states
    }
    norm = sum(posterior.values())
    if norm > 0:
        self._belief = {s: posterior[s] / norm for s in self._states}
    else:
        self.get_logger().warn("正規化定数ゼロ：信念更新スキップ")

    # 期待効用最大化
    best_action = max(

```

```

        self._reward.keys(),
        key=lambda a: sum(self._belief[s] * self._reward[a][s] for s in self._states)
    )

    # メッセージ生成 & 送信
    cmd = LessonCmd()
    cmd.action = best_action
    self._pub.publish(cmd)

def main(args=None):
    rclpy.init(args=args)
    node = LessonSelector()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

設計上の考慮事項とトレードオフ：

- センサ選択対プライバシー：高忠実度カメラはモデリングを改善するが、同意と保存の懸念を引き起こす。
- 解釈可能性対性能：振るまい木は教師の監視を助けるが、最適 RL 方策を制約する可能性がある。
- シミュレーション忠実度対転移リスク：積極的ドメインランダムマイゼーションは sim-to-real ギャップを減らす、トレーニング複雑性を増大させる。

技術的影響と運用上のリスク：

- 正確な生徒モデリングはラベル付きデータに比例する。貧弱なモデルは教育的有害な行動を生む。
- リアルタイム制約は、知覚、信念更新、作動に遅延予算を要求する。予算超過は応答性を低下させる。
- 安全臨界コントローラは受動的ダイナミクスを保証しなければならない。ハードウェアインザループテストなしのコントローラパラメータ調整は安全でない接触を危惧する。
- 展開には、データ保持、同意、教師オーバーライドに関する方針が必要で、責任を制限する。

32.3 障害者支援ロボティクス

前の小節では、教育文脈における適応的相互作用パラダイムと医療における臨床ワークフローを確立した。それらの考え—パーソナライゼーション、安全最優先設計、センサ駆動型適応—は、障害者

のための支援ヒューマノイドシステムに直接適用される。

支援ヒューマノイドロボットは、利用者の運動、感覚、または認知能力を拡張または代替する。問題定義：摂食、更衣、物体取り出し、移動支援などの日常生活動作（ADL）を信頼性高く安全かつパーソナライズされた実行を可能にする。主要な技術目標は予測可能な物理接触、堅牢な意図推定、低遅延共有制御である。以下、それらの目標を実現する制御定式、センシング選択、ソフトウェアパターン、実装プリミティブを分析する。

技術分析

- 共有制御アーキテクチャ：自律軌道計画とユーザ駆動コマンドを融合する。共有制御は認知負荷を軽減し安全でない動作を防ぐ。2つの並列モジュールを実装する：
 1. 意図推定（上位レベル）で確率的意図推定値 I を生成する。
 2. 低レベル準拠コントローラが安全な相互作用を実施し参照軌道 x^{ref} に追従する。
- 意図推定。センサ観測 O を離散的意図仮説 I に写像するために確率的フィルタリングを用いる。ベイズ則より

$$[H]P(I | O) = \frac{P(O | I) P(I)}{\sum_j P(O | I_j) P(I_j)}. \quad (262)$$

ここで $P(O | I)$ は連続信号（EMG、ジョイスティック）にはガウス尤度で、ジェスチャにはカテゴリカル分布でモデルできる。

- 運動・接触制御。安全な物理支援にはインピーダンスまたはアドミタンス制御が業界標準である。連続アドミタンス関係式は

$$[H]M\ddot{x} + D\dot{x} + K(x - x_0) = F_{\text{ext}}, \quad (263)$$

ただし M, D, K は仮想質量、ダンピング、剛性、 F_{ext} は計測された相互作用力である。 K と D を調整することで追従精度とコンプライアンスをトレードオフする。

- 相互作用制約下での最適計画。接触を要するタスクでは制約付き有限ホライズン最適制御問題を定式化する。典型的な2次コストは

$$[H] \min_{u_{0:N-1}} \sum_{k=0}^{N-1} \|x_{k+1} - x_{k+1}^{\text{ref}}\|_Q^2 + \|u_k\|_R^2 \quad (264)$$

ただし動力学、作動器限界、接触力制約 $\|f_{\text{contact}}\| \leq f_{\text{max}}$ に従う。モデル予測制御（MPC）は変化する意図確率 $P(I | O)$ 下でリアルタイム再計画を提供する。

実装プリミティブとセンサ

- センサスイート：手首力・トルクセンサ、接触位置特定用分布型触覚スキン、物体・姿勢用 RGB-D、ユーザ意図用ウェアラブル IMU/EMG、音声コマンド用マイク。多モーダル融合により雑多または騒噪環境での堅牢性が向上する。
- ソフトウェアスタック：知覚・制御用 ROS2 ノード、ADL サブタスクのシーケンシング用ビヘイビアツリー（GROOT）、LfD（実演から学習）方策の訓練と安全エンベロープ検証用 Isaac Sim。
- 学習とパーソナライゼーション：好ましい支援行動の実演を収集；ガウス過程回帰または確率的運動プリミティブを用いて変動性を符号化。オンライン適応により $P(I)$ 事前分布とコントローラ利得をパフォーマンスに基づき更新する。

実用的制御アルゴリズム（技術レシピ）

1. センサストリームを融合しベイズフィルタで $P(I | O)$ を計算する。
2. $\max_I P(I | O)$ が信頼閾値を超えたら対応するタスク計画を確定；さもなければユーザ確認を要求する。
3. 接触・作動器制約を考慮した参照軌道を生成するため MPC レイヤを実行する。
4. 力制限を実施し閾値違反で動作を停止するインピーダンスコントローラで実行する。

コード例：測定力を目標エンドエフェクタ速度に変換する簡略化 ROS2 アドミタンスコントローラ。ノードは/force_sensor を購読し/ee_velocity を配信する。コメントは簡潔に。

コードサンプル 109 安全な支援到達動作のための ROS2 アドミタンスコントローラ

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Wrench, Twist
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
import numpy as np

class AdmittanceController(Node):
    def __init__(self) -> None:
        super().__init__('admittance_controller')

    # パラメータ宣言
    self.declare_parameter('mass', 1.0, ParameterDescriptor(
        type=ParameterType.PARAMETER_DOUBLE,
        description='Virtual mass [kg]'))
    self.declare_parameter('damping', 20.0, ParameterDescriptor(
        type=ParameterType.PARAMETER_DOUBLE,
        description='Virtual damping [Ns/m]'))
    self.declare_parameter('stiffness', 50.0, ParameterDescriptor(
        type=ParameterType.PARAMETER_DOUBLE,
        description='Virtual stiffness [N/m]'))
    self.declare_parameter('dt', 0.01, ParameterDescriptor(
        type=ParameterType.PARAMETER_DOUBLE,
        description='Control period [s]'))
    self.declare_parameter('max_vel', 0.2, ParameterDescriptor(
        type=ParameterType.PARAMETER_DOUBLE,
        description='Velocity limit [m/s]'))

    # パラメータ取得
```

```

self.M = self.get_parameter('mass').value
self.D = self.get_parameter('damping').value
self.K = self.get_parameter('stiffness').value
self.dt = self.get_parameter('dt').value
self.max_vel = self.get_parameter('max_vel').value

# 購読・配信
self.force_sub = self.create_subscription(
    Wrench, '/force_sensor', self.force_cb, 10)
self.vel_pub = self.create_publisher(Twist, '/ee_velocity', 10)

# 内部状態 (1D)
self.x = 0.0
self.x_dot = 0.0
self.last_force = 0.0

# タイマー
self.timer = self.create_timer(self.dt, self.loop)

def force_cb(self, msg: Wrench) -> None:
    self.last_force = msg.force.z

def loop(self) -> None:
    f_ext = self.last_force
    # 半陰的オイラー積分
    x_ddot = (f_ext - self.D * self.x_dot - self.K * self.x) / self.M
    self.x_dot += x_ddot * self.dt
    self.x += self.x_dot * self.dt

# 速度制限
self.x_dot = np.clip(self.x_dot, -self.max_vel, self.max_vel)

# 指令速度配信
twist = Twist()
twist.linear.z = self.x_dot
self.vel_pub.publish(twist)

def main(args=None):
    rclpy.init(args=args)

```

```

node = AdmittanceController()
try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

```

設計上の考慮事項、トレードオフ、運用上のリスク

- 安全性対自律性：自律性を高めるとタスク完了が速まるが故障深刻度が上昇する。常に保守的な力閾値とハードストップを含める。
- 遅延：通信遅延が大きいと共有制御が劣化する。低レベルアドミタンスはロボット上で局所ループを維持；必要なら意図推定はコンパニオン機器で実行する。
- 知覚誤差：意図を誤解釈すると安全でない動作を引き起こす。確認ダイアログと多重冗長センサを用いる。
- 過信とユーザ訓練：ユーザがヒューマノイドを過度に依存する可能性がある。透明なフィードバックと簡単な上書き制御を設計する。
- 規制・プライバシー制約：EMG、映像、音声データは安全な取り扱いと同意を要する。

技術的含意：インピーダンス／アドミタンス利得を選びコンプライアンスと要求位置精度をバランスさせる。局所高速安全コントローラを優先し Isaac Sim でコントローラを検証する。残留リスクを形式的安全エンベロープで定量化しフォールバック動作を組み込む。

自律航行

33 SLAM の基礎

33.1 ヒューマノイドの同時位置推定と地図構築を理解する

これまでの章でのセンサ統合と運動制御の概念を踏まえ、ここではヒューマノイドナビゲーションのための同時位置推定と地図構築（SLAM）を定式化し、脚式運動特有の要素と知覚の制約を強調する。焦点は運用面であり、断続的な接触、動的障害物、視覚的遮蔽に頑健な地図を構築しながら、ヒューマノイドの姿勢を推定する方法を示す。

問題設定と運用上の意義。ヒューマノイドロボットはバランスを保ちながら衝突を回避し、屋内の雑然とした空間を移動しなければならない。SLAM（Simultaneous Localization and Mapping）は、ロボットの軌道推定とノイズの多いセンサデータから一貫した地図構築という 2 つの結合した問題に対処する。ヒューマノイドにとって SLAM は以下に対処する必要がある：

- 歩行中や足滑り時に生じる大きく不連続なオドメトリ誤差；
- 胴体や頭部の動作による頻繁な視点変化；
- センサおよび演算装置の搭載重量制限からくる効率的アルゴリズムの要求。

技術分析：状態表現とモデル。センサとタスクに整合した状態表現を選択する。一般的な 2 つの選択肢：

1. 特徴ベース EKF SLAM：状態ベクトルはロボット姿勢とランドマーク位置を結合する、

$$[H]x_k = \begin{bmatrix} x_{r,k} \\ m_1 \\ \vdots \\ m_N \end{bmatrix}, \quad x_{r,k} \in \text{SE}(3), \quad (265)$$

ここで m_i は 3D 特徴座標を表す。EKF は線形化された運動モデル $f(\cdot)$ と観測モデル $h(\cdot)$ を用いる。

2. グラフベース（ポーズグラフ）SLAM：ノードはロボット姿勢、エッジは相対観測とランドマーク観測を符号化する。最適化は以下のエネルギーを最小化する

$$[H]E(X) = \sum_{(i,j) \in \mathcal{E}} \|\text{Log}(T_{ij}^{-1}T_i^{-1}T_j)\|_{\Sigma_{ij}}^2, \quad (266)$$

ここで $T_i \in \text{SE}(3)$ は姿勢変換、 T_{ij} は相対変換である。

センサ融合はヒューマノイドの SLAM を固定化する。典型的なセンサスイート：

- IMU（高レート慣性データ）によるプラインテグレーションを用いた短期運動予測。
- ステレオまたは RGB-D カメラによる視覚特徴とループクロージャ。
- 構造化環境での計測的マッピングのための 2D/3D LiDAR。
- 力・トルクおよび足底接触センサにより、足が静止している際に信頼できる零速度更新を生成。

ヒューマノイド向けアルゴリズム構成と適応。

- 運動予測：IMU プラインテグレーションをキーフレーム間で用いて相対姿勢事前分布を生成。プラインテグレーションは多様体構造を尊重し線形化誤差を低減する。
- 観測処理：静止足場を検出し零速度更新（ZUPT）を実行して立脚相中のドリフトを低減する。
- ループクロージャ：記述子マッチングと幾何検証による頑健な場所認識で長い廊下に起因する地図不整合を低減する。
- 地図表現：ループクロージャ用のスパースランドマーク集合と即時衝突回避用の密な占有格子を組み合わせたハイブリッド地図を維持する。

EKF SLAM 方程式（コンパクト）。予測：

$$[H]\hat{x}_{k|k-1} = f(\hat{x}_{k-1}, u_k), \quad P_{k|k-1} = F_k P_{k-1} F_k^\top + Q_k. \quad (267)$$

観測 z_k に対する更新：

$$[H]K_k = P_{k|k-1} H_k^\top (H_k P_{k|k-1} H_k^\top + R_k)^{-1}, \quad \hat{x}_k = \hat{x}_{k|k-1} + K_k (z_k - h(\hat{x}_{k|k-1})). \quad (268)$$

これらを実装するには SE(3) 上でのヤコビアン導出と一貫した線形化点が必要である。

実装スケッチ：IMU プラインテグレーションと視覚ループクロージャを伴う増分的ポーズグラフ最適化。コードリストは、IMU と距離・方位特徴を用いたヒューマノイド向けの最小 EKF スタイル予

測・更新ループを示す。これは組み込み実装の出発点であり、IsaacSimでのプロトタイピングにも使える。

コードサンプル 110 IMU と距離・方位特徴を用いたヒューマノイド向け最小 EKF SLAM ループ

```
import numpy as np
from typing import Tuple, List, Iterable, Optional
import scipy.linalg as la

# 状態: [px,py,pz,qx,qy,qz,qw, m1x,m1y,...] ; P 共分散
StateVector = np.ndarray
CovMatrix    = np.ndarray
ImuSample    = np.ndarray
Measurement = Tuple[np.ndarray, int] # (z, landmark_id)

def quat_multiply(q1: np.ndarray, q2: np.ndarray) -> np.ndarray:
    """四元数積 (Hamilton product)"""
    w1, x1, y1, z1 = q1
    w2, x2, y2, z2 = q2
    return np.array([
        w1*w2 - x1*x2 - y1*y2 - z1*z2,
        w1*x2 + x1*w2 + y1*z2 - z1*y2,
        w1*y2 - x1*z2 + y1*w2 + z1*x2,
        w1*z2 + x1*y2 - y1*x2 + z1*w2
    ])

def quat_inv(q: np.ndarray) -> np.ndarray:
    """四元数逆"""
    return np.array([q[0], -q[1], -q[2], -q[3]]) / (q @ q)

def integrate_pose(pose: np.ndarray, imu: ImuSample, dt: float) -> np.ndarray:
    """SE(3) 上でIMUを積分 (簡易版) """
    p, q = pose[:3], pose[3:7]
    acc, gyro = imu[:3], imu[3:6]
    # 角速度→四元数更新
    dq = np.array([1.0, 0.5*gyro[0]*dt, 0.5*gyro[1]*dt, 0.5*gyro[2]*dt])
    q_new = quat_multiply(q, dq)
    q_new /= la.norm(q_new)
    # 加速度→速度→位置 (重力補償省略)
    p_new = p + 0.5 * acc * dt**2
    return np.concatenate([p_new, q_new])
```

```

def compute_F(pose: np.ndarray, imu: ImuSample, dt: float) -> np.ndarray:
    """状態遷移ヤコビアン (簡易) """
    dim = len(pose)
    F = np.eye(dim)
    # 位置・姿勢の線形化 (簡略)
    F[:3, :3] += np.eye(3) * 0.01 # ダミー
    return F

def measurement_model(state: StateVector, idx: int) -> Tuple[np.ndarray, np.ndarray]:
    """ランドマーク観測モデル (方位+距離) """
    p = state[:3]
    lm = state[7+3*idx:7+3*(idx+1)]
    d = lm - p
    dist = la.norm(d)
    z_hat = np.array([d[0]/dist, d[1]/dist, d[2]/dist, dist])
    # 観測ヤコビアン (簡易)
    H = np.zeros((4, len(state)))
    H[:3, :3] = -np.eye(3)/dist
    H[:3, 7+3*idx:7+3*(idx+1)] = np.eye(3)/dist
    H[3, :3] = -d/dist
    H[3, 7+3*idx:7+3*(idx+1)] = d/dist
    return z_hat, H

def predict(state: StateVector, P: CovMatrix, imu: ImuSample,
            dt: float, Q: CovMatrix) -> Tuple[StateVector, CovMatrix]:
    """予測ステップ"""
    pose = state[:7]
    new_pose = integrate_pose(pose, imu, dt)
    F = compute_F(pose, imu, dt)
    state[:7] = new_pose
    P = F @ P @ F.T + Q
    return state, P

def update(state: StateVector, P: CovMatrix, z: np.ndarray, R: CovMatrix,
            landmark_idx: int) -> Tuple[StateVector, CovMatrix]:
    """更新ステップ"""
    h, H = measurement_model(state, landmark_idx)
    y = z - h
    S = H @ P @ H.T + R

```

```

K = la.solve(S, H @ P).T # 数値安定版
state += K @ y
I_KH = np.eye(P.shape[0]) - K @ H
P = I_KH @ P @ I_KH.T + K @ R @ K.T # Joseph form
return state, P

def init_state() -> StateVector:
    """初期状態"""
    return np.array([0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0])

def init_cov(dim: int = 7) -> CovMatrix:
    """初期共分散"""
    return np.eye(dim) * 1.0

def sensor_stream() -> Iterable[Tuple[ImuSample, List[Measurement], float]]:
    """シミュレーション／ROS 2 トピックからIMUと特徴観測を供給"""
    # 実際にはROS 2 subscriber等で置換
    while True:
        yield np.zeros(6), [], 0.01

def main() -> None:
    state = init_state()
    P = init_cov()
    Q_imu = np.eye(7) * 0.01
    R_feat = np.eye(4) * 0.05

    for imu, measurements, dt in sensor_stream():
        state, P = predict(state, P, imu, dt, Q_imu)
        for z, idx in measurements:
            state, P = update(state, P, z, R_feat, idx)

if __name__ == "__main__":
    main()

```

設計上のトレードオフと実装メモ。

- EKF SLAM は計算負荷が軽いがランドマーク数に対してスケールしにくい。スパース化またはサブマッピングを用いる。
- ポーズグラフ最適化は大規模地図とループクロージャに対応するが、バッチまたは増分ソルバーとより多くのメモリを要求する。
- ビジュアル SLAM は豊富な特徴を提供するが低照度では失敗する。LiDAR は計測的精度を与える

が搭載重量と消費電力が増える。

- ・ キーフレーム間で IMU プリインテグレーションを用いることで CPU 負荷を下げ非線形最適化の条件付けを改善する。

エンジニアリング上の影響、トレードオフ、リスク。

- ・ 演算対地図精度：オンボード演算能力の限界によりスパースグラフと密占有格子の選択が迫られる。
- ・ 動的運動中の安定性：不正確な足底接触検出は大きなオドメトリ誤差を生む。プロプライオセプティブセンサを統合して確実に立脚を検出する。
- ・ 安全性：SLAM の失敗は人が存在する環境で衝突を引き起こす。姿勢不確実性閾値に対するウォッチドッグと安全停止動作を実装する。
- ・ 検証：ハードウェア展開前に Isaac Sim で擾乱をシミュレートしてテストする。ドリフトを露呈するため長いループクロージャを含める。

これらの運用処方、ヒューマノイドナビゲーションのための SLAM バリエーションとセンサスイートを選択する際の精度、演算、安全性のバランスを指針とする。

33.2 ヒューマノイドのためのマッピング技術

これらのマッピング技術は、SLAM の基礎、特に一貫した空間表現を構築しながら姿勢を推定する必要性に直接基づいている。表現の選択と統合パイプラインは、移動するセンサマウント、多層環境、接触駆動のオドメトリといったヒューマノイド特有の制約を考慮する必要がある。

問題定義. ヒューマノイドは、垂直特徴、階段、動的エージェントを持つ 3 次元の構造化された屋内環境で動作する。センサを可動部（頭部、胴体）に搭載しており、生の測定値を破損させる身体運動が発生する。したがって、マッピングシステムは、リアルタイムの位置推定、足取り計画、障害物回避をサポートする、多解像度でメモリ効率の良いマップを生成しなければならない。

マップ表現とアルゴリズムの技術的分析.

- ・ メトリック 2D 占有率グリッドは、平面歩行廊下や高速な衝突チェックに依然として有用である。メモリが軽く、古典的なプランナと統合しやすい。
- ・ 3D ボクセルまたはオクツリーマップ（例：OctoMap）は体積占有率を表現し、多解像度クエリを可能にする。オクツリーは $\log(N)$ のメモリスケーリングと衝突チェックのための高速なレイキャスティングを提供する。
- ・ 切り捨て符号付き距離場 (TSDF) は、操縦や接触計画のための滑らかな表面を生成する。TSDF は深度測定値を符号付き距離関数に統合し、マーチングキューブによる表面抽出をサポートする。
- ・ サーフエルと高密度点ベースマップは、表面要素ごとの法線と信頼度を維持し、光度整合と意味マッピングを可能にする。
- ・ 位相的およびハイブリッドマップは、長距離構造をグラフノードに圧縮する。大規模建物でのグローバル計画と再位置推定に効率的である。
- ・ 意味マップは、幾何学にオブジェクトラベル、アフォーダンス、ナビゲーション制約を付加する。ヒューマノイドにとって、意味情報は人間を意識したナビゲーションとインタラクティブタスクを支援する。

コア数学演算. 占有率統合には、数値アンダーフローを回避し、増分的バイズ更新を可能にするため、対数オッズが頻繁に使用される。 l_t を時刻 t の対数オッズ、 l_0 を事前分布、 $L(z_t|x_t)$ をロボット姿勢 x_t での測定 z_t の逆センサモデルとする。更新は：

$$[H]l_t = l_{t-1} + L(z_t | x_t) - l_0. \quad (269)$$

姿勢グラフ最適化は、SE(3) 上の相対姿勢制約を最小化する。ノード i と j の間の相対測定 T_{ij} に対して、残差は群対数を使用する：

$$[H] \min_{\{T_k\}} \sum_{(i,j) \in \mathcal{E}} \|\log(T_{ij}^{-1} T_i^{-1} T_j)\|_{\Sigma_{ij}}^2. \quad (270)$$

ここで Σ_{ij} は制約 (i, j) の情報行列である。

センサとマウントの考慮事項. 多モーダル融合を使用：

- 視覚慣性オドメトリ（VIO）は、頭部トルクと高周波運動を補償する。
- 車輪または足部オドメトリと接触フラグは、立位と歩行中に追加の制約を提供する。
- LiDAR は、特に低テクスチャ廊下でループクロージャのための安定したレンジジオメトリを提供する。

振り子効果を最小化するようにセンサをマウントする。主要深度センサを胴体近くにマウントすると、頭部マウントと比較して回転速度が低減する。頭部マウントセンサが必要な場合は、マッピングの前に IMU ベースの運動補償とローリングシャッター補正を統合する。

パイプライン実装推奨事項.

1. ローカルマッピング：高レート深度または LiDAR フレームを短命のサブマップに蓄積する。高密度再構成には TSDF を、ナビゲーションにはボクセル占有率を使用する。
2. オドメトリ：高周波姿勢推定のために VIO または LiDAR オドメトリを実行する。IMU 事前積分を姿勢グラフ事前分布と融合する。
3. ループクロージャとグローバルマッピング：NetVLAD や Bag-of-Words のような外観記述子で場所の再訪を検出する。姿勢グラフに制約を追加し、増分的に最適化する。
4. マップメンテナンス：オクツリープルーニングで圧縮し、タスク固有クエリをサポートするため意味ラベルをリーフノードに格納する。

CPU 制約のヒューマノイドコントローラに適したミニマリスト占有率更新例が、対数オッズ則を示す。この Python スニペットは、ボクセルグリッドでのレイごとの更新を示す。

コードサンプル 111 ボクセルグリッドのための単純な対数オッズ占有率更新

```
import numpy as np
from typing import Tuple, Sequence

# センサモデルの対数オッズ定数
LOG_ODDS_PRIOR: float = 0.0
LOG_ODDS_OCC: float = 0.85
LOG_ODDS_FREE: float = -0.4
```

```
LOG_ODDS_MIN: float = -4.0
LOG_ODDS_MAX: float = 4.0
```

```
def update_voxel_logodds(
    grid: np.ndarray,
    ray_voxels: Sequence[Tuple[int, int, int]],
    hit: bool,
) -> None:
    """
    3D占有グリッドの対数オッズを、1本のレイで更新する（ベクトル化版）

    Parameters
    -----
    grid: np.ndarray
        3D配列（shape=(X,Y,Z)）． 対数オッズ値を保持．
    ray_voxels: Sequence[Tuple[int, int, int]]
        レイに沿った（終端を除く）ボクセルインデックスのリスト
    hit: bool
        レイが占有セルで終端した場合True
    """
    if not ray_voxels:
        return

    # フリースペース更新（ベクトル化）
    free_delta = LOG_ODDS_FREE - LOG_ODDS_PRIOR
    idx_array = np.asarray(ray_voxels, dtype=int)
    grid[tuple(idx_array.T)] += free_delta
    np.clip(
        grid,
        LOG_ODDS_MIN,
        LOG_ODDS_MAX,
        out=grid,
    )

    # ヒット時のみ終端を占有更新
    if hit:
        end = ray_voxels[-1]
        grid[end] += LOG_ODDS_OCC - LOG_ODDS_PRIOR
        grid[end] = np.clip(grid[end], LOG_ODDS_MIN, LOG_ODDS_MAX)
```

エンジニアリングへの影響、トレードオフ、リスク。

- ・計算量vs 忠実度：TSDF とサーフェルは高忠実度を与えるが、高密度融合には GPU 加速が必要。オクツリまたはスパースサブマップは CPU とメモリ使用量を低減する。
- ・リアルタイム安全性：古くまたは一貫性のないマップは安全でない足場を引き起こす可能性がある。即時衝突チェックのため、常に短期間のローカル障害物層と直接センサ読み取り値を保持する。
- ・ループクロージャ遅延：積極的なグローバル最適化は姿勢ジャンプを引き起こす可能性がある。滑らかな再位置推定戦略による継続的マッピングを使用する。
- ・動的環境：移動する人間や物体は、持続的な偽障害物を回避するために時間フィルタリングまたは動的オブジェクトマスキングを必要とする。
- ・マルチセッションマッピング：グローバル記述子と幾何学的整合による頑健なマップマージを確保する。マップの整合失敗はフロア間でドリフト蓄積を引き起こす。

設計者は、バッテリー、計算、タスク、予想される環境ダイナミクスという運用制約に整合した表現と融合戦略を選択しなければならない。

33.3 精度向上のためのセンサ融合

前の小節では、ヒューマノイドプラットフォームが用いる SLAM の基礎とマッピング表現を確立した。これらの考えを発展させ、本小節では補完的なセンサを融合することで二足歩行システムの局在化誤差を削減し、マッピングを安定化する方法に焦点を当てる。

センサ融合問題の定式化。ヒューマノイドロボットは歩行、操縦、インタラクションのために正確な姿勢と地図推定を必要とする。単一センサは特徴的な故障モードを持つ：IMU はドリフトし、カメラは低照度で失敗し、LiDAR は近距離障害物を見逃し、関節エンコーダは足の滑りを無視する。センサ融合は測定値を組み合わせ、ロボット状態の統計的に整合性のとれた、分散の小さい推定値を生成する。融合を不確実性下での状態推定として定式化し、時刻 k の状態ベクトル x_k は通常、ベース姿勢、ベース速度、センサバイアスを含む。

確率定式化と代表的な推定器。離散時間運動・観測モデルは

$$[H]x_k = f(x_{k-1}, u_k) + w_k, \quad z_k = h(x_k) + v_k, \quad (271)$$

であり、 u_k は制御または IMU 入力、 $w_k \sim \mathcal{N}(0, Q_k)$ はプロセス雑音、 $v_k \sim \mathcal{N}(0, R_k)$ は観測雑音である。実用的な推定器の 2 つのファミリーがヒューマノイド SLAM を支配する：

1. 再帰フィルタ（EKF/UKF/IEKF）：リソース制約ハードウェアにおける低遅延・固定遅延パイプラインに適する。現在の推定値周辺で線形化し、共分散 P_k を持つガウス信念を伝播する。
2. 最適化/因子グラフ法（GTSAM, Ceres）：複数の姿勢とランドマーク上の残差を最小化するバッチまたは増分解決策。非線形性とマルチセンサ制約を頑強に扱う。

ヒューマノイドで運用上有用な EKF 方程式は：

$$[H]\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k), \quad P_{k|k-1} = F_k P_{k-1|k-1} F_k^\top + Q_k, \quad (272)$$

$$[H]K_k = P_{k|k-1} H_k^\top (H_k P_{k|k-1} H_k^\top + R_k)^{-1}, \quad \hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k (z_k - h(\hat{x}_{k|k-1})), \quad (273)$$

ただし $F_k = \partial f / \partial x$ 、 $H_k = \partial h / \partial x$ 。これらの演算は IMU からの予測とビジョンまたは LiDAR からの補正を実装する。

ヒューマノイド融合の重要なエンジニアリング考慮事項。

- マルチレート・非同期データ：IMU は高レートデータ（100–2000 Hz）を生成；カメラと LiDAR は低レートフレームを提供する。バッファキューとタイムスタンプ補間を用いる。更新前に測定タイムスタンプへ予測ステップを適用する。
- フレームとエクストリンシクス：base_link から各センサへの剛体変換を維持する。キャリブレーション誤差はバイアスとして現れ融合を劣化させる。同次変換で表現し、線形化を通じて不確実性を伝播する。
- 接触と運動学的制約：足の接触と関節エンコーダを擬似測定として活用する。足が静止時にゼロ速度更新（ZUPT）を組み込みドリフトを抑制する。
- 頑強な外れ値除去：マハラノビスゲーティングとグラフ最適化での頑強な損失関数を用い、動的障害物または誤ったビジュアルマッチから推定値を汚染するのを防ぐ。

最適化ベース融合。すべてのモダリティからの残差和を目的関数として定式化する：

$$[H] \min_X \sum_i \|r_{\text{imu},i}(X)\|_{\Sigma_{\text{imu},i}^{-1}}^2 + \sum_j \|r_{\text{vis},j}(X)\|_{\Sigma_{\text{vis},j}^{-1}}^2 + \sum_k \|r_{\text{kin},k}(X)\|_{\Sigma_{\text{kin},k}^{-1}}^2, \quad (274)$$

ここで X は姿勢とバイアスの集合。ソルバーは因子を増分的に追加し、計算量とメモリを制限するための周辺化戦略を適用する。

実用的実装パターン。二層アーキテクチャを用いる：

- 低レベル高レート EKF またはストラップダウン積分による即時バランス・制御ループ。
- 中長期グラフ最適化器が定期的にループクロージャを受け入れグローバルマップを改良する。

例：IMU と姿勢測定（LiDAR/ビジュアルオドメトリから）の簡潔な EKF 融合。コードは状態伝播と測定更新を示す。明快さのため実装を最小限に留める。

コードサンプル 112 ヒューマノイド向け IMU + 姿勢測定の最小 EKF

```
import numpy as np
from typing import Optional, Tuple

class HumanoidEKF:
    """
    拡張カルマンフィルタによる人型ロボットの自己位置推定
    状態: u[px, upy, upz, uvx, uvy, uvz, ubax, ubay, ubaz]
    """

    def __init__(self,
                  x0: np.ndarray,
                  P0: np.ndarray,
                  Q: np.ndarray,
```

```

        R_pose: np.ndarray) -> None:
    """
    Args:
        x0: 初期状態ベクトル (9,)
        P0: 初期共分散行列 (9,9)
        Q: プロセスノイズ共分散 (9,9)
        R_pose: 観測ノイズ共分散 (3,3)
    """
    assert x0.shape == (9,), "x0 must be shape (9,)"
    assert P0.shape == (9, 9), "P0 must be shape (9,9)"
    assert Q.shape == (9, 9), "Q must be shape (9,9)"
    assert R_pose.shape == (3, 3), "R_pose must be shape (3,3)"

    self.x = x0.copy()
    self.P = P0.copy()
    self.Q = Q.copy()
    self.R_pose = R_pose.copy()

    # 観測行列 (位置のみ観測)
    self.H_pose = np.zeros((3, 9))
    self.H_pose[:3, :3] = np.eye(3)

    def predict(self,
                imu_omega: np.ndarray,
                imu_accel: np.ndarray,
                dt: float) -> None:
        """
        IMU情報を用いて状態を予測
        Args:
            imu_omega: 角速度 (3,)
            imu_accel: 加速度 (3,)
            dt: 時刻差 [s]
        """
        assert imu_omega.shape == (3,), "imu_omega must be shape (3,)"
        assert imu_accel.shape == (3,), "imu_accel must be shape (3,)"
        assert dt > 0, "dt must be positive"

        p = self.x[0:3]
        v = self.x[3:6]
        b_a = self.x[6:9]

```

```

# バイアス補正後の加速度
acc = imu_accel - b_a

# 状態遷移（等加速度モデル）
p_new = p + v * dt + 0.5 * acc * dt ** 2
v_new = v + acc * dt

self.x[0:3] = p_new
self.x[3:6] = v_new

# 線形化された状態遷移行列
F = np.eye(9)
F[0:3, 3:6] = np.eye(3) * dt
F[3:6, 6:9] = -np.eye(3) * dt

# 共分散予測
self.P = F @ self.P @ F.T + self.Q

def update_pose(self,
                  pose_meas: np.ndarray,
                  R_pose: Optional[np.ndarray] = None) -> None:
    """
    位置観測を用いて状態を更新
    Args:
        pose_meas: 位置観測 [px, py, pz] (3,)
        R_pose: 観測ノイズ共分散 (3,3). Noneの場合はコンストラクタの値を使用
    """
    assert pose_meas.shape == (3,), "pose_meas must be shape (3,)"

    R = R_pose if R_pose is not None else self.R_pose

    # 予測観測
    z_pred = self.H_pose @ self.x

    # イノベーション共分散
    S = self.H_pose @ self.P @ self.H_pose.T + R

    # カルマンゲイン
    K = self.P @ self.H_pose.T @ np.linalg.inv(S)

```

```

# 状態更新
self.x += K @ (pose_meas - z_pred)

# 共分散更新 (Joseph form for numerical stability)
I_KH = np.eye(9) - K @ self.H_pose
self.P = I_KH @ self.P @ I_KH.T + K @ R @ K.T

def get_state(self) -> Tuple[np.ndarray, np.ndarray]:
    """
    現在の状態と共分散を取得
    Returns:
    x: 状態ベクトル (9,)
    P: 共分散行列 (9,9)
    """
    return self.x.copy(), self.P.copy()

```

設計・運用のトレードオフ。

- 遅延 vs 精度：高周波フィルタリングは遅延を減らすがバイアスを蓄積する可能性がある。周期的なグローバル最適化はドリフトを減らすが計算コストが高い。
- センサ選択：冗長モダリティの追加は精度を向上させるが重量、電力、キャリブレーション作業を増やす。ヒューマノイドには頭部・胴体用の軽量・低電力センサを優先する。
- キャリブレーションと同期：エクストリンシクスのミスアライメントまたはクロックスキューは系統誤差を引き起こす。頑強なオートキャリブレーションルーチンとハードウェアタイムスタンブへの投資が必要。

故障モードとリスク。

- 過信共分散はモデリング誤差存在時にフィルタ発散を引き起こす。
- 動的環境と移動領域は一部のビジュアル SLAM パイプラインの静的マップ仮定を破壊する。
- 足の滑りは運動学的制約を無効化；接触ベース更新を検出・ゲートする。

運用上の影響。リアルタイムバランス制御を満たしながらドリフトが制限されたグローバルマップを維持するため多層融合を実装する。高速制御ループはリアルタイムプロセッサで実行し、重いグラフ最適化はコプロセッサで実行するように計算を割り当てる。歩行中の壊滅的な局在化エラーを防ぐため、頑強なキャリブレーション、時刻同期、外れ値除去を最優先とする。

33.4 ケーススタディ：屋内ナビゲーション

これまでのセンサ融合とマッピング技術の議論を発展させ、本ケーススタディではそれらの原理を人型ロボットの実用的な屋内ナビゲーション問題に適用する。視覚、深度、LiDAR、慣性データとポーズグラフマッピングを組み合わせ、オフィス状環境で頑健でメトリックに整合した地図を生成する。

問題定義。研究用の人型ロボットはオフィスフロアを自律的に歩行し、ワークステーションを点検

して基地に戻らなければならない。課題は狭い廊下、デスク下の遮蔽、動く人、床面の反射率変化、胴体が引き起こすセンサフレームの攪乱である。タスクの制約は：

- 各ゴールでのテーブルトップ操作にメトリック位置推定精度が 0.1 m より良いこと；
- 30 minutes の連続運転でドリフトが有界であること；
- オンボード計算資源が単一 GPU とマルチコア CPU に限られること。

技術分析。設計は計算複雑度、センサ冗長性、リアルタイム要件をトレードオフしなければならない。補完的なセンサを活用するハイブリッド SLAM パイプラインを選択する：

- IMU を高レート運動事前分布と転倒検出に；
- 車輪/脚オドメトリと関節エンコーダを接地時の運動学的制約に；
- 3D LiDAR を長距離幾何特徴と特徴の乏しい廊下での頑健なループクロージャに；
- RGB-D カメラを高密度局所マッピングとセマンティック手がかり（机、椅子、ドア）に。

状態表現は離散的なロボットポーズを $SE(3)$ で表すポーズグラフを用いる。グラフ最適化は複数の因子（相対スキャンマッチング、視覚オドメトリ、IMU 事前積分、ループクロージャ制約）の残差を最小化する。大域目的関数は

$$[H]x^* = \arg \min_x \sum_{(i,j) \in \mathcal{E}_{\text{geom}}} \|r_{ij}^{\text{geom}}(x)\|_{\Sigma_{ij}}^2 + \sum_{k \in \mathcal{E}_{\text{IMU}}} \|r_k^{\text{IMU}}(x)\|_{\Sigma_k}^2 + \sum_{\ell \in \mathcal{E}_{\text{loop}}} \|r_\ell^{\text{loop}}(x)\|_{\Sigma_\ell}^2, \quad (275)$$

ここで r_{ij}^{geom} は ICP またはスキャンマッチング残差、 r_k^{IMU} は事前積分 IMU 残差、 r_ℓ^{loop} は場所認識によるループクロージャ制約である。

IMU 事前積分はノードタイムスタンプ間で用いられ、最適化変数を削減する慣性因子を生成する。2 ポーズ (R_i, p_i) と (R_j, p_j) に事前積分測定 $\Delta R_{ij}, \Delta v_{ij}, \Delta p_{ij}$ が与えられたとき、最適化で用いられる残差は標準的な事前積分形式に従う：

$$r_{ij}^{\text{IMU}} = \begin{bmatrix} \log(\Delta R_{ij}^\top (R_i^\top R_j)) \\ R_i^\top (p_j - p_i - v_i \Delta t + \frac{1}{2} g \Delta t^2) - \Delta p_{ij} \\ R_i^\top (v_j - v_i + g \Delta t) - \Delta v_{ij} \end{bmatrix}. \quad (276)$$

これは時変バイアス下で整合性を保ち、必要に応じて速度の周辺化をサポートする。

実装手順とエンジニアリング選択：

1. 時刻同期：ハードウェアタイムスタンプで全センサを共通クロックに対して同期する。ソフトウェアレベル補間で小さなオフセットを補正する。
2. キャリブレーション：RGB-D, LiDAR, IMU, 胴体フレーム間の外参キャリブレーションを運動ベース手法で実施。変換を TF ツリーに保存しリアルタイム参照する。
3. フロントエンド：軽量視覚オドメトリ（例：RGB-D 用に改変した ORB-SLAM2）をカメラレートで高密度局所追跡に実行。LiDAR スキャンマッチャ（例：LOAM 系）を並列化し長距離特徴に用いる。
4. データ対応：視覚特徴の Bag-of-Words と LiDAR ヒストグラムの幾何記述子を組み合わせたキー付きハイブリッド場所認識でループクロージャ候補をトリガする。
5. バックエンド：IMU 因子、幾何制約、ループクロージャを含む増分ポーズグラフ最適化（例：GTSAM iSAM2）を使用。GPU 加速線形ソルバが利用可能な場合は重い最適化を実行する。

6. 安全層：反応的障害物回避モジュールがグローバル地図更新中でも局所コストマップを用いて安全な足場を確保する。

実用的パラメータ選択：

- キーフレーム挿入は 0.5–1.0 m ごとまたは回転変位が 15° を超えたとき。
- LiDAR スキャンマッチングはエッジ受容閾値を 0.05 m。
- IMU バイアス再推定は 10 s ごとにバイアスドリフトを抑制。

最小構成の ROS2 プロトタイプはこれらの入力をポーズグラフアップデータに融合する方法を示す。ノードは /imu/data, /camera/depth/points, /velodyne/points を購読し、最適化ポーズを公開する。

コードサンプル 113 IMU, RGB-D, LiDAR を統合したシンプルな ROS2 ポーズグラフアップデータ

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import Imu, PointCloud2
from geometry_msgs.msg import PoseWithCovarianceStamped
from nav_msgs.msg import Odometry
import threading
import copy
from typing import Optional

# 独自ライブラリ（GTSAMラッパー、ICP、RGB-Dオドメトリ）を仮定
from gtsam_wrapper import GTSAMWrapper
from rgb_d_odometry import run_rgb_d_odometry
from scan_matcher import scan_match, loop_candidate, loop_constraint

class PoseGraphUpdater(Node):
    def __init__(self) -> None:
        super().__init__('pose_graph_updater')

        # QoS：ベストエフォート+適切なキュー深さ
        imu_qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=200
        )
        sensor_qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
```

```

        history=HistoryPolicy.KEEP_LAST,
        depth=10
    )

    # サブスクライバ
    self.imu_sub = self.create_subscription(
        Imu, '/imu/data', self.imu_cb, imu_qos)
    self.rgbd_sub = self.create_subscription(
        PointCloud2, '/camera/depth/points', self.rgbd_cb, sensor_qos)
    self.lidar_sub = self.create_subscription(
        PointCloud2, '/velodyne/points', self.lidar_cb, sensor_qos)

    # パブリッシャ
    self.pose_pub = self.create_publisher(
        PoseWithCovarianceStamped, '/pose_graph/pose', 10)

    # スレッドセーフなバッファとロック
    self.imu_buffer: list[Imu] = []
    self.buffer_lock = threading.Lock()

    # GTSAMラッパー初期化
    self.optimizer = GTSAMWrapper()

    # 最適化実行フラグ
    self.optimize_needed = False
    self.optimize_timer = self.create_timer(0.05, self.try_optimize)

# 20Hz

    # ---- コールバック群 ----
    def imu_cb(self, msg: Imu) -> None:
        with self.buffer_lock:
            self.imu_buffer.append(msg)

    def rgbd_cb(self, msg: PointCloud2) -> None:
        odom = run_rgbd_odometry(msg)
        if odom is not None:
            self.optimizer.add_odometry_factor(odom)
            self.optimize_needed = True

    def lidar_cb(self, msg: PointCloud2) -> None:

```

```

        rel = scan_match(msg)
        if rel is None:
            return
        self.optimizer.add_geometry_factor(rel)
        if loop_candidate(rel):
            loop = loop_constraint(rel)
            if loop is not None:
                self.optimizer.add_loop_factor(loop)
        self.optimize_needed = True

# ---- 最適化 ----
def try_optimize(self) -> None:
    if not self.optimize_needed:
        return
    self.optimize_needed = False
    self.optimizer.update()
    pose = self.optimizer.get_current_pose()
    self.publish_pose(pose)

# ---- パブリッシュ ----
def publish_pose(self, pose) -> None:
    msg = PoseWithCovarianceStamped()
    msg.header.stamp = self.get_clock().now().to_msg()
    msg.header.frame_id = 'map'
    msg.pose = pose # 仮定: poseはPoseWithCovariance型
    self.pose_pub.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = PoseGraphUpdater()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

```
if __name__ == '__main__':  
    main()
```

運用上の影響とトレードオフ：

- LiDAR を追加するとテクスチャの乏しい廊下でループクロージャ信頼性が向上するが、計算量と重量が増加する。
- RGB-D は人型ロボットの胴体近くのテーブルトップタスクをサポートする。ただし、近距離の日射窓は深度読みを劣化させる。
- 積極的なキーフレーム挿入はドリフトを低減するがグラフサイズが増加；周辺化方針でメモリと最適化遅延を制御する。
- IMU 事前積分は高レート安定性を維持するが、正確なバイアス推定を必要とし、衝撃後に失敗する可能性がある。

設計上のトレードオフとリスク：

- 視覚 SLAM のみに依存すると特徴のない廊下で壊滅的ドリフトのリスクがある。ハイブリッドシステムの方が頑健である。
- 胴体とセンサ間の外参キャリブレーションが不適切だと、整合しているが不正確な地図が生成され、操縦タスクに悪影響を及ぼす。
- リアルタイム保証には綿密なプロファイリングが必要：バックエンド最適化は分離しないと制御ジッタを引き起こす可能性がある。

エンジニアはフィデューシャルマーカと床面測量でマッピング精度を検証すべきである。センサ冗長性とフェイルセーフ動作を優先し、アイドル区間中の偶発的な大域最適化に計算資源を割り当てること。

34 経路計画アルゴリズム

34.1 グラフベース手法 (A*、ダイクストラ法)

前述の SLAM 由来の占有格子および融合センサ推定値は、ここで用いられる環境表現を提供する。これらの地図は、A*やダイクストラ法といったグラフベース計画手法にとってコスト面およびノード集合となる。

グラフベース経路計画は、グラフ $G = (V, E)$ 上の最短経路問題としてナビゲーションを定式化する。ノード V は離散的なロボット姿勢または接地位置を表す。エッジ E は姿勢間の実行可能な遷移を結ぶ。ヒューマノイドロボットの場合、ノード状態には位置 (x, y) と方位 θ が一般に含まれ、歩行位相や歩数インデックスを含むこともある。計画の目的は、運動学および安定性制約を満たしながら、累積コストを最小化するノード列を見つけることである。

問題定義。以下が与えられたとする：

- 障害物占有率とクリアランスコストを持つ地図、
- 開始姿勢 $s \in V$ と目標姿勢 $g \in V$ 、

- ・ヒューマノイドの動的および安定性制約、

総コスト

$$[H]C(\pi) = \sum_{i=0}^{n-1} c(v_i, v_{i+1}), \quad (277)$$

を最小化する経路 $\pi = \{v_0 = s, v_1, \dots, v_n = g\}$ を見つける。ここで $c(\cdot, \cdot)$ は移動エネルギー、時間、安定性余裕、リスクを符号化する。

中核アルゴリズム。A*とダイクストラ法はヒューリスティックの使用においてのみ異なる。 $g(n)$ を開始からノード n へのコスト、 $h(n)$ を目標へのヒューリスティック推定値と定義する。A*の評価は

$$[H]f(n) = g(n) + h(n). \quad (278)$$

ダイクストラ法は $h(n) = 0$ の A*である。非負エッジコストにおける最適性のため、ヒューリスティックは許容的 ($h(n) \leq h^*(n)$) であり、可能なら一貫性を持つ必要がある：

$$[H]h(n) \leq c(n, m) + h(m) \quad \forall (n, m) \in E, \quad (279)$$

これは単調な f 値と効率的な探索を保証する。

ヒューマノイドのための状態とコストの設計。ロボットの能力を反映するように状態とエッジを選ぶ：

- ・低コスト解像度でベース並進のための格子または navmesh ノード。
- ・足-to-足遷移のための格子または足跡グラフノード。
- ・高コスト回転をペナルティとして付加する方位を含む拡張ノード。

コスト成分を構築する：

- ・幾何コスト：距離または移動時間、
- ・クリアランスペナルティ：最近傍障害物までの距離の逆数、
- ・歩行実行可能性：足配置が到達可能性に違反する場所に大きなペナルティ、
- ・安定性余裕ペナルティ：余裕が小さいほどコストが増加。

実用的な複合エッジコストは

$$[H]c(u, v) = w_d \cdot d(u, v) + w_c \cdot \phi_{\text{clear}}(v) + w_s \cdot \phi_{\text{stab}}(u, v), \quad (280)$$

で与えられる。重み w_d, w_c, w_s はプラットフォームごとに調整される。ここで d はユークリッド距離、 ϕ_{clear} はクリアランスをペナルティに写像し、 ϕ_{stab} は遷移に対する動的安定性を評価する。

ヒューリスティックの選択。ベースレベル計画では、公称歩長でスケールしたユークリッド距離が許容的ヒューリスティックを生む。方位を意識した計画では回転コストを含める：

$$[H]h(n) = \alpha \cdot \|p_n - p_g\|_2 + \beta \cdot |\text{wrap}(\theta_n - \theta_g)|, \quad (281)$$

$\alpha, \beta \geq 0$ は h が許容的であるように選ぶ。歩数制限がある場合、最大歩幅で距離を割り許容性を保持する。

実装上の注意と最適化。

- ロボットのエンベロープに等しい占有膨張を用いてクリアランスを確保する。
- 8 連結またはより高解像度の格子を用いてギザギザ軌道を減らす。
- 足跡グラフのための到達可能性マップを事前計算し実行不可能エッジを除去する。
- オープン集合に二分ヒープまたはフィボナッチヒープを用い $O(|E| + |V| \log |V|)$ 実行時間を達成する。
- 地図が頻繁に更新される場合は増分計画 (D* Lite) を検討する。

以下の Python 実装は、ヒューマノイドナビゲーションに適応した格子ベース A*を示す。クリアランスペナルティと方位旋回コストを追加して歩行と回転労力を近似する。

コードサンプル 114 ヒューマノイドベースレベル計画のためのクリアランスと旋回コストを含む A*, label

```
import heapq
import math
from typing import Tuple, Dict, List, Iterable, Optional

Node = Tuple[int, int, int] # (x, y, theta_index)

class GridMap:
    """軽量インターフェース：占有・クリアランス両方を返す"""
    def __init__(self, occ, clearance):
        self.occ = occ
        self.clearance = clearance
        self.h, self.w = occ.shape

    def in_bounds(self, x: int, y: int) -> bool:
        return 0 <= x < self.w and 0 <= y < self.h

    def is_occupied(self, x: int, y: int) -> bool:
        return bool(self.occ[y, x])

    def clearance_at(self, x: int, y: int) -> float:
        return float(self.clearance[y, x])

def heuristic(n: Node, goal: Node, alpha: float = 1.0, beta: float = 0.5) -> float:
    dx = n[0] - goal[0]
    dy = n[1] - goal[1]
    dist = math.hypot(dx, dy)
    dtheta = abs(((n[2] - goal[2] + math.pi) % (2 * math.pi)) - math.pi)
    return alpha * dist + beta * dtheta
```

```

def neighbors(node: Node, grid: GridMap, theta_steps: int = 8) -> Iterable[Node]:
    x, y, t = node
    # 8近傍+回転3通り
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1),
                   (-1, -1), (1, 1), (-1, 1), (1, -1)]:
        nx, ny = x + dx, y + dy
        if not grid.in_bounds(nx, ny) or grid.is_occupied(nx, ny):
            continue
        for dt in (-1, 0, 1):
            nt = (t + dt) % theta_steps
            yield (nx, ny, nt)

def edge_cost(u: Node, v: Node, clearance_grid: GridMap, max_step: float = 1.0) -> float:
    dist = math.hypot(v[0] - u[0], v[1] - u[1])
    clear_pen = max(0.0, 1.0 - clearance_grid.clearance_at(v[0], v[1]))
    turn = abs(v[2] - u[2])
    return dist + 2.0 * clear_pen + 0.2 * turn

def astar(start: Node, goal: Node, grid: GridMap) -> Optional[List[Node]]:
    open_set: List[Tuple[float, Node]] = []
    heapq.heappush(open_set, (0.0, start))

    g: Dict[Node, float] = {start: 0.0}
    parent: Dict[Node, Node] = {}

    while open_set:
        _, current = heapq.heappop(open_set)
        if current == goal:
            break

        for nbr in neighbors(current, grid):
            tentative = g[current] + edge_cost(current, nbr, grid)
            if tentative < g.get(nbr, math.inf):
                g[nbr] = tentative
                f = tentative + heuristic(nbr, goal)
                heapq.heappush(open_set, (f, nbr))
                parent[nbr] = current

```

```

else:
    return None # 経路なし

# 経路再構成
path = []
cur = goal
while cur in parent:
    path.append(cur)
    cur = parent[cur]
path.append(start)
return path[::-1]

```

エンジニアリング上のトレードオフと運用上のリスク。

- 解像度対計算：より細かい離散化は経路品質を向上させるが、方位や歩行状態を追加するとノード数は指数的に増加する。
- 安全性対機敏性：大きなクリアランスはより安全な計画を生むが、遅いまたは非効率的な動作となる。
- SLAM 誤差：地図の不正確さは安全でないエッジを生じる；保守的な膨張とセンサ検証を用いる。
- 動的障害物： A^* は静的；移動物体には再計画または増分アルゴリズムを用いる。
- リアルタイム制約：重いグラフ展開を GPU にオフロードするか、階層計画—粗いグローバル計画、細かいローカルプランナーを用いる。

実用的な展開では、グラフベースグローバルプランナをローカルで動的に実行可能なコントローラと結合する。エッジコストがヒューマノイドの歩行とバランス限界を反映することを確保する。ハードウェア試験前に、ランダムセンサノイズを伴うシミュレーション（Isaac Sim）で検証し、運用リスクを低減する。

34.2 サンプリングベース手法（RRT, PRM）

これまでの議論では、離散グラフ探索と占有グリッド上の連続計画を対比し、解像度が最適性やコストに与える影響を示した。サンプリングベースプランナは、固定離散化の多くの限界を、連続構成空間を確率的に探索することで解決する。

サンプリングベース手法は、ロボット構成空間 C 内でサンプルを生成することで経路計画問題を解く。衝突のない状態を $C_{\text{free}} \subset C$ と表記する。計画問題は、開始 q_s から目標 q_g を結ぶ連続経路 $\gamma: [0, 1] \rightarrow C_{\text{free}}$ を見つけることである。形式的には、

$$[H]\gamma(0) = q_s, \quad \gamma(1) = q_g, \quad \gamma(t) \in C_{\text{free}} \forall t \in [0, 1]. \quad (282)$$

ヒューマノイドロボットでは、 C には基底姿勢、関節角度、接触モードパラメータが含まれることが多い。高次元化とバランス制約により、サンプリング戦略の設計が重要となる。

コアルゴリズムと性質：

- Rapidly-exploring Random Trees (RRT)： q_s を根とする木を、 $q_{\text{rand}} \sim \text{Sampler}()$ を繰り返しサンプリングし、最近傍ノード q_{near} （メトリクス下）を見つけ、 q_{rand} 方向にステアリングして q_{new} を

生成し、衝突チェック後に追加することで構築する。RRT は確率的完全性を持つ：無限サンプルで（存在すれば）経路発見確率が 1 に収束する。

- RRT-Connect： q_s と q_g から 2 本の木を成長させ、接続を試みる。cluttered 環境での実用速度を改善する。
- RRT*：経路コストを削減するよう木を再配線する変種で、漸近最適性を達成する。再配線ステップでは親関係を更新してコストを最小化する：

$$[H]c(q_{\text{new}}) = \min_{q \in \mathcal{N}(q_{\text{new}})} (c(q) + \text{cost}(q, q_{\text{new}})). \quad (283)$$

- Probabilistic Roadmaps (PRM)： C_{free} 内でノードをサンプリングし、近傍ノードをローカルプランナで接続してグラフを保存する。ロードマップ構築コストを再利用できるため、マルチクエリタスクに有利。PRM*は接続半径をスケーリングさせることで漸近最適性を保証する PRM 拡張である。

ヒューマノイドシステムへの実用上の考慮：

1. 構成空間構造：

- 連続と離散（接触）変数が混在し C を拡張する。ハイブリッド表現を用いるか、計画を足歩計画と関節レベル運動計画に分解する。
- バランスと安定性制約が実行可能構成を低次元多様体に制限する。その多様体上で直接サンプリングすると、足歩問題で成功率が高まる。

2. サンプリング戦略：

- 一様関節空間サンプリングは、狭い通路や接触面近傍ではしばしば失敗する。
- ゴールバイアスや障害物近傍でのガウスサンプリングは、狭い通路での成功率を改善する。
- タスク空間サンプリング（重心や足位置をサンプリング）と逆運動学（IK）を組み合わせることで、無駄なサンプルを削減する。
- ブリッジテストサンプリングは、ヒューマノイド足場に関連する狭い通路を発見するのに役立つ。

3. 距離メトリクスとステアリング：

- 基底姿勢と脚関節を優先し、指の微動を軽視する重み付き関節空間距離を用いる。
- 足歩計画では、各足に対して SE(2) メトリクスと離散足接触ラベルを定義する。
- ステアリングは運動学的限界と衝突のない補間を尊重すべきである。キノダイナミック計画では、制御入力を伴う動的方程式を積分する。

4. 衝突チェックとローカルプランナ：

- ヒューマノイドボディには正確な衝突モデルと、補間状態間での連続時間衝突チェックが必要である。
- 掃引体積、保守的前進、サブステップチェックを用いて、運動補間中の見落とし衝突を回避する。
- 遅延衝突チェックは、高価な衝突テストを候補経路発見後まで遅延させる。

5. キノダイナミクスと動的バランス：

- 動的マニューバでは、計画をキノダイナミックとして扱う：状態には速度と制御が含まれ、衝突のない軌道は動的方程式を満たす。

- 補間中に ZMP（零力矩点）を支持多角形内に保つなど、安定性制約を組み込む。

6. 計算加速：

- 衝突クエリと最近傍探索を並列化する。空間ハッシュや KD 木を最近傍探索に用いる。
- シミュレーションでは、GPU ベース衝突検出や NVIDIA Isaac Sim を活用して、バッチ衝突チェックと物理検証を行う。

アルゴリズム選択指針：

- 静的環境で多数クエリを計画する場合（倉庫ヒューマノイドの繰り返し経路など）は PRM を用いる。
- cluttered 室内を単発・短時間で航行する場合は RRT-Connect を用いる。
- 経路最適性が重要で計算時間に余裕がある場合（エネルギー効率の良い長距離移動など）は RRT* を用いる。
- 動的歩行・走行では、制御入力と状態微分が重要になるため、キノダイナミック RRT 変種を用いる。

実装例（平面二脚足歩抽象化への簡略 RRT）：

```
import math
import random
from typing import Dict, List, Optional, Tuple

# 3-DoF 姿勢型エイリアス
Pose = Tuple[float, float, float]

class Node:
    """RRT木のノード：姿勢と親ノードを保持"""
    def __init__(self, q: Pose, parent: Optional["Node"] = None) -> None:
        self.q: Pose = q
        self.parent: Optional["Node"] = parent

def distance(a: Pose, b: Pose) -> float:
    """SE(2)距離：位置+角度を重み付けて評価"""
    dx = a[0] - b[0]
    dy = a[1] - b[1]
    dtheta = math.atan2(math.sin(a[2] - b[2]), math.cos(a[2] - b[2]))
    return math.hypot(dx, dy) + 0.1 * abs(dtheta)
```

```

def sample_random(bounds: Dict[str, float]) -> Pose:
    """設定空間から一様ランダムに姿勢をサンプリング"""
    x = random.uniform(bounds["xmin"], bounds["xmax"])
    y = random.uniform(bounds["ymin"], bounds["ymax"])
    theta = random.uniform(-math.pi, math.pi)
    return (x, y, theta)

def nearest(tree: List[Node], q_rand: Pose) -> Node:
    """tree中でq_randに最も近いノードを返す"""
    return min(tree, key=lambda n: distance(n.q, q_rand))

def steer(q_from: Pose, q_to: Pose, step: float = 0.2) -> Pose:
    """固定ステップ長でSE(2)線形補間"""
    dist = distance(q_from, q_to)
    if dist <= step:
        return q_to
    alpha = step / dist
    x = q_from[0] + alpha * (q_to[0] - q_from[0])
    y = q_from[1] + alpha * (q_to[1] - q_from[1])
    theta = q_from[2] + alpha * math.atan2(
        math.sin(q_to[2] - q_from[2]), math.cos(q_to[2] - q_from[2])
    )
    theta = math.atan2(math.sin(theta), math.cos(theta)) # [-pi, pi] 正規化
    return (x, y, theta)

def reconstruct_path(leaf: Node) -> List[Pose]:
    """ゴールノードから根まで逆順にパスを再構成"""
    path: List[Pose] = []
    curr: Optional[Node] = leaf
    while curr is not None:
        path.append(curr.q)
        curr = curr.parent
    path.reverse()
    return path

def plan_rrt(

```

```

    q_start: Pose,
    q_goal: Pose,
    bounds: Dict[str, float],
    max_iter: int = 5000,
    goal_sample_rate: float = 0.05,
    step_size: float = 0.2,
    goal_tol: float = 0.3,
) -> Optional[List[Pose]]:
    """
    単純RRTプランナ
    collision_checkは呼び出し側で実装すること
    """
    tree: List[Node] = [Node(q_start)]

    for _ in range(max_iter):
        # ゴールバイアス：確率的に q_goal をサンプリング
        if random.random() < goal_sample_rate:
            q_rand = q_goal
        else:
            q_rand = sample_random(bounds)

        q_near = nearest(tree, q_rand)
        q_new = steer(q_near.q, q_rand, step_size)

        # 衝突チェック（未実装：呼び出し側で定義）
        if not collision_check(q_near.q, q_new):
            continue

        tree.append(Node(q_new, q_near))

        # ゴール到達判定
        if distance(q_new, q_goal) < goal_tol:
            if collision_check(q_new, q_goal):
                return reconstruct_path(Node(q_goal, tree[-1]))

    return None

# 呼び出し側で実装必須
def collision_check(q_from: Pose, q_to: Pose) -> bool:

```

```

"""ダミー衝突チェック：常にFalse（衝突あり）を返す"""
return False

```

エンジニアリングへの影響, トレードオフ, 運用上のリスク:

- トレードオフ: RRT 変種は高速発見と非最適経路をトレードし, RRT*は高コストで最適性を回復する。PRM は重い前処理をクエリ間で償却する。
- 設計選択: サンプラ設計, メトリクス重み付け, ステアリング関数は, ヒューマノイド足場・バランス制限付き計画の成否に強く影響する。
- 運用上のリスク: 不適切なサンプリングや不十分な衝突チェックは, 不安定または安全でない動作を生む可能性がある。ハードウェア実行前に物理シミュレーション (例: Isaac Sim) で計画を検証する。
- 安全慣行: 許容関節速度を制限し, 軌道内で ZMP またはキャプチャ領域制約を検証し, センサがバランス喪失を示した場合にリアルタイムで中止する監視を実行する。

34.3 パスプランニングへのハイブリッドアプローチ

ヒューマノイドパスプランニング問題の定義. ヒューマノイドロボットは, 静的および動的障害物が存在する環境で, 衝突を回避し動的に実行可能な軌道を開始点から目標点まで生成しなければならない. 制約は以下の通りである:

- 多数の関節による高次元の構成空間,
- 足部およびバランスの非ホロノミック的な制約 (支持多角形, ゼロモーメントポイント),
- 離散的な足踏み決定と連続的な全身運動,
- センサノイズと遅延を伴うリアルタイム実行.

技術分析: アーキテクチャパターン. ロボティクス文献と実践には, 4 つの実用的なハイブリッドアーキテクチャが存在する:

1. 階層的グローバル・ローカル: 低解像度のグローバルプランナ (グリッド, グラフ) が大まかな経路を計算する. ローカルプランナ (サンプリングベースまたは最適化ベース) はセグメントを運動学的・動的に実行可能な動作に細分化する.
2. 足踏み中心ハイブリッド: 離散的な足踏みプランナが足接地位置の列を生成し, 各ステップはバランスおよび衝突制約を満たすようローカル動作プランナによって検証または改良される.
3. 任意時再計画ループ: 高速の非最適プランナが即座の参照を提供し, 低速の最適プランナがコントローラが高速参照を追従している間に経路を改善する.
4. 結合最適化・サンプリング: サンプリングシードが軌道最適化 (例: CHOMP, TrajOpt) を初期化し, サンプリングの頑健性と最適化の滑らかさを組み合わせる.

長距離最適性と短距離実行可能性の間での意思決定を促進するハイブリッドコストブレン드의数学的定式化. C_{global} をグローバル経路コスト, C_{local} を障害物への近接や動力学を含むローカル実行可能性コストとする. セグメント s に対するブレード目的関数は

$$J_s = (1 - \lambda) C_{\text{global},s} + \lambda C_{\text{local},s}, \quad \lambda \in [0, 1]. \quad (284)$$

として書ける． λ を調整することで，公称最適性または動的実行可能性のどちらかを優先する．実際には λ は時変であり，障害物への接近時やバランス余裕が狭まる際に増加する．

ローカル軌道最適化はヒューマノイドの動力学を尊重しなければならない．連続時間最適制御部分問題は

$$\begin{aligned} \min_{x(\cdot), u(\cdot)} \quad & \int_{t_0}^{t_f} \|u(t)\|_R^2 + w_{\text{smooth}} \|\ddot{q}(t)\|^2 dt + w_{\text{coll}} C_{\text{coll}}(x(t)), \\ \text{s.t. } \quad & \dot{x} = f(x, u), \\ & h_{\text{ZMP}}(x, u) \leq 0, \quad h_{\text{collision}}(x) \leq 0, \\ & q_{\min} \leq q(t) \leq q_{\max}, \quad u_{\min} \leq u(t) \leq u_{\max}, \end{aligned} \tag{285}$$

として定式化される．ここで x は一般化座標と速度を積み上げ， u はアクチュエータトルクまたは関節速度指令であり， h_{ZMP} は支持多角形内の圧力中心制限を課す．

実装戦略とアルゴリズムステップ．頑健なヒューマノイド用ハイブリッドプランナは通常，以下のループを実装する：

- ステップ 1：バランス余裕のため障害物膨張を施したコストマップ上で A^* を用いて大まかなグローバル経路を計算する．
- ステップ 2：センサ視界と制御遅延に応じたサイズの重複するローカルウィンドウにグローバル経路を分割する．
- ステップ 3：各ウィンドウに対し，足踏みプリミティブと近似動力学に制約された運動力学サンプラー（RRT^{*}，kinodynamic-RRT）を実行する．
- ステップ 4：最良のサンプリング経路で軌道最適化をウォームスタートし，厳密な動力学と衝突制約を課す．
- ステップ 5：時間パラメータ化された軌道をモデル予測制御（MPC）に送り追従させる．逸脱が許容値を超えたり新たな障害物が現れたら MPC は再計画する．

実装例．以下のリストは簡潔なハイブリッドループスケルトンを示す．このコードは API の詳細ではなくアーキテクチャを重視する．プレースホルダをプラットフォーム固有の ROS2，Isaac Sim，またはコントローラインタフェースに置き換えること．

コードサンプル 116 ヒューマノイド用ハイブリッドグローバル・ローカルプランニングループ

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from nav_msgs.msg import Path, Odometry
from geometry_msgs.msg import Twist, PoseStamped
from sensor_msgs.msg import LaserScan
import numpy as np
from typing import Optional, List
import time
```

```

# 自作モジュール（同一パッケージ内に配置）
from global_planner import AStarPlanner
from local_sampler import KinodynamicRRTStar
from trajectory_optimizer import TrajectoryOptimizer
from mpc_controller import MPCController
from state_estimator import StateEstimator
from obstacle_detector import ObstacleDetector

class HybridPlanner(Node):
    def __init__(self) -> None:
        super().__init__('hybrid_planner')

        # QoS：実機・シミュレータ両対応
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        # パラメータ
        self.declare_parameter('goal_x', 0.0)
        self.declare_parameter('goal_y', 0.0)
        self.declare_parameter('local_horizon', 5.0)          # [m]
        self.declare_parameter('replan_threshold', 0.3)

        # [m] 障害物変化閾値

        self.goal = np.array([
            self.get_parameter('goal_x').value,
            self.get_parameter('goal_y').value
        ])

        # プランナ・コントローラ初期化
        self.global_planner = AStarPlanner()
        self.local_sampler = KinodynamicRRTStar()
        self.optimizer = TrajectoryOptimizer()
        self.mpc = MPCController()
        self.state_estimator = StateEstimator()
        self.obstacle_detector = ObstacleDetector()

```

```

# 購読・配信
self.scan_sub = self.create_subscription(
    LaserScan, '/scan', self.scan_callback, qos)
self.odom_sub = self.create_subscription(
    Odometry, '/odom', self.odom_callback, qos)
self.cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)
self.path_pub = self.create_publisher(Path, '/global_path', 10)

# 内部状態
self.robot_pose = np.zeros(3)      # [x, y, yaw]
self.latest_scan: Optional[LaserScan] = None
self.global_path: Optional[List[np.ndarray]] = None

# タイマー駆動 (100 Hz)
self.timer = self.create_timer(0.01, self.step)

self.get_logger().info('HybridPlannerReady.')

# ---- コールバック ----
def scan_callback(self, msg: LaserScan) -> None:
    self.latest_scan = msg

def odom_callback(self, msg: Odometry) -> None:
    p = msg.pose.pose
    self.robot_pose = np.array([p.position.x, p.position.y, self._yaw_from_quat(p.

# ---- 主ループ ----
def step(self) -> None:
    if self.latest_scan is None:
        return # センサ未受信

# 状態推定更新
self.state_estimator.update(self.robot_pose, self.latest_scan)

# ゴール到着判定
if np.linalg.norm(self.robot_pose[:2] - self.goal) < 0.2:
    self.cmd_pub.publish(Twist()) # 停止
    self.get_logger().info('GoalReached.')
    rclpy.shutdown()
    return

```

```

# グローバル経路生成（初回 or リプラン）
if self.global_path is None:
    self.global_path = self.global_planner.compute(
        self.robot_pose[:2], self.goal)
    self._publish_path(self.global_path)

# ローカルウィンドウ抽出
local_window = self._extract_window(
    self.global_path, self.robot_pose, self.get_parameter('local_horizon').value)

# 動的障害物変化チェック
if self.obstacle_detector.significant_change(
    self.latest_scan, self.get_parameter('replan_threshold').value):
    self.global_path = None # 次ループで再計算
    return

# ローカル経路生成
sampled_traj = self.local_sampler.rrt_star(
    local_window, self.robot_pose, self.latest_scan)

if sampled_traj is None:
    # フォールバック：グローバル経路を単純追従
    control_reference = self._follow_global_coarse(self.global_path)
else:
    # 最適化 + MPC
    opt_traj = self.optimizer.solve(sampled_traj)
    control_reference = self.mpc.track(opt_traj, self.robot_pose)

# 指令値送信
self._send_reference(control_reference)

# ---- ヘルパ ----
def _extract_window(self, path: List[np.ndarray], pose: np.ndarray, horizon: float)
    """path内で現在位置からhorizon以内の点列を返す"""
    dists = [np.linalg.norm(p - pose[:2]) for p in path]
    idx = np.searchsorted(dists, horizon, side='right')
    return path[:idx+1]

def _follow_global_coarse(self, path: List[np.ndarray]) -> Twist:

```

```

"""単純なPure_Pursuit風追従（フォールバック用）"""
if len(path) < 2:
    return Twist()
target = path[1]
dx = target[0] - self.robot_pose[0]
dy = target[1] - self.robot_pose[1]
theta = self.robot_pose[2]
e_theta = np.arctan2(dy, dx) - theta
v = 0.5
w = 2.0 * np.arctan2(np.sin(e_theta), np.cos(e_theta))
twist = Twist()
twist.linear.x = v
twist.angular.z = w
return twist

def _send_reference(self, ref: Twist) -> None:
    self.cmd_pub.publish(ref)

def _publish_path(self, path: List[np.ndarray]) -> None:
    msg = Path()
    msg.header.frame_id = 'map'
    msg.header.stamp = self.get_clock().now().to_msg()
    for p in path:
        ps = PoseStamped()
        ps.pose.position.x = p[0]
        ps.pose.position.y = p[1]
        msg.poses.append(ps)
    self.path_pub.publish(msg)

    @staticmethod
    def _yaw_from_quat(q) -> float:
        import tf_transformations
        return tf_transformations.euler_from_quaternion([q.x, q.y, q.z, q.w])[2]

def main(args=None):
    rclpy.init(args=args)
    node = HybridPlanner()
    rclpy.spin(node)
    node.destroy_node()

```

```
rclpy.shutdown()
```

```
if __name__ == '__main__':  
    main()
```

エンジニアリング上の考慮事項とトレードオフ。ハイブリッドアプローチを選択または調整する際、以下の要因を評価する：

- ・計算量 vs 安全性：高密度サンプリングと厳密な最適化は CPU と遅延を増加させる。遅延削減のため GPU ベースの衝突検査や並列サンプリングなどのハードウェア加速を用いる。
- ・センシング視界と再計画頻度：短い視界は古い地図のリスクを減らすがプランナのチャーンを増加させる。ウィンドウサイズをセンサレンジとコントローラ帯域に合わせる。
- ・バランス余裕と保守性：ZMP 安全性を維持するため障害物を膨張させるが、過剰な膨張は狭い通路を塞ぐ。
- ・足踏み離散化：粗い足踏み格子は計画を簡素化するが複雑地形で実行可能解を見失う可能性がある。
- ・知覚ノイズに対する頑健性：確率的障害物モデルを統合し、高不確実性下では式 (1) の保守的なラムダスケジューリングを優先する。

運用上のリスクには予期せぬ動的障害物、状態推定の遅延、最適化器動力学と低レベルコントローラ間のモデルミスマッチが含まれる。これらは保守的な安全バッファ、非同期再計画、階層的フォールバック動作によって軽減する。

34.4 実世界の例：障害物回避

先ほど議論したハイブリッドおよびサンプリングベースのアイデアは、静的および動的な障害物の中で動作するヒューマノイドに対する実用的な戦略を促す。以下の例では、グローバルグラフプランナ、ローカルサンプリング、および運動学的／動的チェックを組み合わせ、実際のヒューマノイドプラットフォームに対して安全で実行可能な操縦を生成する方法を示す。

問題定義。ヒューマノイドは屋内ワークスペースで開始姿勢からゴールへ移動しなければならない。障害物には静的な家具、小さな散らばりやすい物体、および移動する人間が含まれる。プランナは、全身および足置き場に対して、衝突のない、運動学的に到達可能で、動的に安定した軌道を提供しなければならない。主要な制約は：

- ・配置空間における衝突回避、
- ・バランス（ゼロモーメント点または支持多角形の制約）、
- ・アクチュエータトルクおよび関節限界、
- ・遅延およびセンサ不確実性。

技術的分析。衝突幾何学を保守的に扱い、ロボットの体積投影によって障害物を膨張させる。ロボット本体集合を体座標系で R 、世界座標系での障害物集合を O とする。剛体並進に対する配置空間障害物は

$$[H]C_{\text{obs}} = \{x \in \mathbb{R}^2 \mid (R+x) \cap O \neq \emptyset\} = O \oplus (-R), \quad (286)$$

ここで \oplus はミンコフスキー和を表す。ヒューマノイドの場合、これを胴体姿勢と離散的足踏み系列を組み合わせた低次元近似に拡張する。脚および胴体に対して保守的なエンベロープを用いてグローバル経路を計画する。

移動時間、クリアランス、および安定性をトレードオフするための複合コスト関数を設計する：

$$[H]J = \alpha L + \beta \int_0^T \frac{1}{d_{\min}(t)} dt + \gamma \min_t m_{\text{stab}}(t), \quad (287)$$

ここで L は経路長、 $d_{\min}(t)$ は最小障害物距離、 $m_{\text{stab}}(t)$ は支持多角形内で正の安定性マージンである。運動学的到達可能性およびトルク限界の下で J を最小化することを目指す。

アルゴリズムアプローチ。信頼できる障害物回避のための3層シーケンスを実装する：

1. グローバルプランナ（グラフベース）：コストマップ上で A^* を用いて粗いコリドーを計算する。膨張した障害物および意味コスト（例：人間ゾーンを高くペナルティ）を用いる。出力は長距離ルーティングを導く。
2. ローカル運動動的プランナ（サンプリングベース）：グローバルコリドーに制限された状態空間で RRT* または運動動的 RRT を実行する。プランナは胴体姿勢および速度対をサンプリングし、足接触を不安定化しないための最大許容胴体加速度などの動的制約を課す。
3. 足踏みおよび全身マップ：胴体軌道を離散的足踏み系列に変換する。支持多角形制約の下で足踏み配置および重心軌道に対して制約付き非線形計画問題を解く。各足踏きを逆運動学（IK）および衝突チェックで検証する。

動的障害物対処。短い予測ホライズンの運動モデル（等速または学習済み歩行者モデル）で移動障害物を予測する。時間パラメータ化された衝突チェックを用いる：ロボット掃引集合 $S_r(t)$ および障害物軌道 $o(t)$ に対して、

$$[H] \forall t \in [0, T] \quad S_r(t) \cap o(t) = \emptyset. \quad (288)$$

を要求する。高速障害物が脅威となる場合は、反応戦略（速度障害物または時間スケーリング）に切り替える。時間スケーリングは元の軌道を係数 $s(t) \geq 1$ で伸ばし、予測衝突を回避するが完全な幾何再計画は行わない。

実装（実用的スニペット）。以下の Python 風ルーチンは、グローバル計画、ローカル RRT*、および足踏みチェックの統合を示す。コストマップおよび動的障害物予測を生成する知覚モジュールへのアクセスを仮定する。

コードサンプル 117 RRT*、足踏み検証、動的障害物チェックを統合したローカルプランナ

```
import time
import numpy as np
from typing import Optional, List, Tuple
from dataclasses import dataclass

# ROS2 関連（必要に応じてアンコメント）
# import rclpy
# from geometry_msgs.msg import PoseStamped
```

```

# from nav_msgs.msg import Path

@dataclass
class State:
    x: float
    y: float
    theta: float
    v: float

@dataclass
class MotionPlan:
    traj: List[State]
    footsteps: List[np.ndarray] # 各足位置のリスト

def plan_to_goal(
    start: State,
    goal: State,
    costmap: np.ndarray,
    dyn_preds: List[np.ndarray],
    max_plan_time: float = 1.0,
    replan_threshold: int = 3
) -> Optional[MotionPlan]:
    """
    グローバル経路をA*で生成し、RRT*で局所軌道を最適化して足位置を決定
    """
    # A* でグローバルコリドーを取得
    corridor = astar_global(start, goal, costmap)
    if not corridor:
        return None

    # RRT* 初期化
    tree = RRTStar(state_dim=4)
    tree.add_root(start)

    start_time = time.time()
    replan_count = 0

    while (time.time() - start_time) < max_plan_time:
        # バイアス付きサンプリング
        x_rand = sample_in_corridor(corridor)

```

```

x_near = tree.nearest(x_rand)
x_new = steer_kinodynamic(x_near, x_rand)

# 衝突チェック
if not collision_check_kin(x_new, costmap):
    continue
if time_collides_with_preds(x_new, dyn_preds):
    continue

tree.add_node(x_new, parent=x_near)

# ゴール到達判定
if tree.has_solution(goal):
    break

if not tree.has_solution(goal):
    return None

traj = tree.reconstruct_path(goal)
footsteps = footstep_planner(traj)

# 各足位置を検証
for fs in footsteps:
    if not ik_solve(fs):
        fs_adj = adjust_footstep(fs)
        if not ik_solve(fs_adj):
            replan_count += 1
            if replan_count >= replan_threshold:
                return None
            continue
    if not stability_margin_ok(fs):
        return None

return assemble_motion_plan(traj, footsteps)

```

エンジニアリングにおける影響とトレードオフ：

- 計算コスト：運動動的チェックおよび全身 IK を伴う RRT*は高コストである。ノード削減のためコリドー誘導サンプリングを用いる。
- 知覚遅延：予測誤差はリスクを増大させる。人間近傍ではクリアランスを増加または歩行を遅くする。

- ・保守性対機敏性：安全を保証するためモデルを膨張させる；これはワークスペースを減らしより長い経路を強制する可能性がある。
- ・回復挙動：滑りや突発的人間侵入には反射制御を予約する。反射層なしのプランナのみの方は動的人間環境に不十分である。

運用上のリスク：センサ遮蔽、人間意図の誤予測、アクチュエータ飽和は安全でない相互作用をもたらす可能性がある。設計には段階的なフォールバック挙動、安定性マージンのリアルタイム監視、およびハードウェア展開前のシミュレーションにおける厳格な検証を含めなければならない。

35 ナビゲーションの課題

35.1 動的環境の取り扱い

これまでに導入したナビゲーション基本機能と知覚パイプラインを基に、本小節では歩行者、車両、操作可能な物体が動く環境でヒューマノイドロボットを頑健に動作させるための具体的な手法に焦点を当てる。予測、制約付き運動計画、安定性を意識した制御を統合し、リアルタイムに衝突を回避しながらバランスを維持することを重視する。

問題定義と工学的制約。ヒューマノイドは全身の安定性、アクチュエータ限界、リアルタイム遅延を尊重しながら衝突を回避しなければならない。主な制約は以下の通りである：

- ・不確定な運動と部分的観測を持つ動的障害物；
- ・ロボットのハイブリッド歩行ダイナミクス（離散的足底接触と連続的胴体運動）；
- ・有効な反応時間を短縮するセンシングおよび演算遅延。

実用的な解決策は、短い予測地平と制約を意識したモデル予測制御（MPC）を組み合わせ、歩容と重心を調整することである。

動的障害物の予測モデリング。オンボード計画では、予測精度を計算速度とトレードオフする低次の確率運動モデルを用いる。定速度モデルにガウス過程ノイズを加えたものが一般的である：

$$[H]o(t + \Delta t) = o(t) + v_o(t) \Delta t + w, \quad w \sim \mathcal{N}(0, \Sigma_w). \quad (289)$$

$v_o(t)$ はカルマンフィルタ、または向きや非線形ダイナミクスが重要な場合は拡張カルマンフィルタ（EKF）で推定する。衝突確率を制限し安全距離を拡張するため、共分散 $\Sigma_o(t + \Delta t)$ を保持する。

MPC における衝突制約。運動学的到達可能性と安定性マージンを尊重する漸次最適化を定式化する。ロボット状態を x_k 、制御入力を u_k とする。標準的な制約付き MPC は以下のように記述される：

$$\begin{aligned} \min_{u_{0:N-1}} \quad & \sum_{k=0}^{N-1} \|x_k - x_k^{\text{ref}}\|_Q + \|u_k\|_R \\ [H] \quad \text{s.t.} \quad & x_{k+1} = f(x_k, u_k), \\ & \text{dist}(p_k, o_k^{\text{pred}}) \geq d_{\text{safe}}(\Sigma_o), \quad \forall k, \\ & u_{\min} \leq u_k \leq u_{\max}, \end{aligned} \quad (290)$$

ここで $\text{dist}(\cdot, \cdot)$ はロボットフットプリントまたは歩容領域と予測障害物位置の適切な距離指標である。確率的な安全性を達成するため、障害物共分散で d_{safe} を膨張する。

安定性を意識した制約. バランスが危険な動作では, キャプチャポイントまたはゼロモーメントポイント (ZMP) 制約を f に組み込むか不等式制約として追加する. 実用的な簡易化として, 重心 (CoM) の投影をマージン Δ_{stab} だけ拡張した許容支持多角形内に強制する:

$$[H]\Pi(\text{CoM}_k) \in \mathcal{S}_k \ominus B(0, \Delta_{\text{stab}}). \quad (291)$$

ここで Π は CoM を地面に投影し, \ominus は半径 Δ_{stab} のボールによる形態学的侵食を表す.

リアクティブレイヤと計画階層. 高速リアクティブレイヤを低速 MPC と組み合わせる:

1. 高レート障害物追跡器がセンサ融合から位置と共分散を予測する.
2. 高速リアクティブモジュールが, 差し迫ったリスクが検出された場合に即座の衝突回避基本動作 (例: 横歩き, 停止) を計算する.
3. MPC は低レートで歩容列と全身運動を洗練し, 安定性とアクチュエータ限界を尊重する.

この階層は最悪遅延を削減しながら, 地平にわたる軌道最適性を保持する.

実装スケッチ. 以下の Python スニペットは最小限のループを示す: カルマンフィルタで障害物を追跡し, 位置を予測し, クリアランスを評価し, MPC ソルバを呼び出す. コメントは簡潔で実用的である.

コードサンプル 118 ヒューマノイドナビゲーションのためのリアクティブ予測と MPC 呼出し

```
import numpy as np
from typing import List, Tuple, Dict, Optional
import logging

# ロギング設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# 定数定義
STATE_DIM = 4
POS_DIM = 2
DEFAULT_PROCESS_NOISE = 0.01
DEFAULT_BASE_MARGIN = 0.6
DEFAULT_CONFIDENCE_COEFF = 2.0 # 95%信頼区間近似

class ObstaclePredictor:
    """障害物の定速度予測を行うクラス"""

    def __init__(self, dt: float, process_noise: float = DEFAULT_PROCESS_NOISE):
        self.dt = dt
        self.Q = process_noise * np.eye(STATE_DIM)
        self.F = np.array([
```

```

        [1, 0, dt, 0],
        [0, 1, 0, dt],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

```

```

def predict(self, state: np.ndarray, P: np.ndarray, steps: int) -> Tuple[List[np.ndarray], List[np.ndarray]]:
    """障害物の将来状態を予測"""
    if state.shape != (STATE_DIM,) or P.shape != (STATE_DIM, STATE_DIM):
        raise ValueError(f"Invalid state or covariance shape")

    preds, Ps = [], []
    x = state.copy()
    Pk = P.copy()

    for _ in range(steps):
        x = self.F @ x
        Pk = self.F @ Pk @ self.F.T + self.Q
        preds.append(x[:POS_DIM].copy())
        Ps.append(Pk[:POS_DIM, :POS_DIM].copy())

    return preds, Ps

```

```

class SafetyCalculator:
    """安全距離を計算するクラス"""

    @staticmethod
    def compute_safe_distance(P: np.ndarray,
                              base_margin: float = DEFAULT_BASE_MARGIN,
                              confidence_coeff: float = DEFAULT_CONFIDENCE_COEFF) -> float:
        """共分散行列から安全距離を計算"""
        if P.shape != (POS_DIM, POS_DIM):
            raise ValueError(f"Invalid covariance shape")

        eigvals = np.linalg.eigvals(P)
        sigma = np.sqrt(max(np.real(eigvals)))
        return base_margin + confidence_coeff * sigma

```

```

class MPCPlanner:
    """MPCベースの経路計画クラス（ブレースホルダー）"""

```

```

def plan(self,
        current_state: Optional[np.ndarray],
        predicted_obs: List[np.ndarray],
        safe_dist: float) -> Dict[str, List[Tuple[float, ...]]]:
    """MPCによる経路計画（実装は外部ライブラリに依存）"""
    # 実際の実装ではCasADiやCVXPYを使用
    logger.info(f"Planning with {len(predicted_obs)} obstacles, safe_dist={safe_dist}")

    return {
        "footsteps": [(0.5, 0.0)],
        "com": [(0.0, 0.0, 0.9)]
    }

def main():
    """メイン制御ループ"""
    # 初期化
    predictor = ObstaclePredictor(dt=0.1)
    safety_calc = SafetyCalculator()
    planner = MPCPlanner()

    # 障害物初期状態
    obs_state = np.array([2.0, 0.5, -0.2, 0.0])
    P = np.eye(STATE_DIM) * 0.05

    try:
        # 障害物予測
        preds, Ps = predictor.predict(obs_state, P, steps=10)

        # 安全距離計算
        safe_dist = safety_calc.compute_safe_distance(Ps[0])

        # MPC計画
        plan = planner.plan(
            current_state=None,
            predicted_obs=preds,
            safe_dist=safe_dist
        )

        logger.info("Planning completed successfully")

```

```

except Exception as e:
    logger.error(f"Planning failed: {e}")
    raise

if __name__ == "__main__":
    main()

```

設計上のトレードオフと運用上のリスク。これらの手法をヒューマノイドに展開する際は、以下を考慮する：

- ・計算負荷対予測地平：地平が長いほど先見性が向上するが、解時間が増加する；
- ・保守的な共分散膨張は衝突を減らす、不要な停止によりタスク失敗が増加する；
- ・リアクティブ動作は接触やアクチュエータ限界を違反すればロボットを不安定にするリスクがある；
- ・センサ遮蔽とマルチエージェント相互作用には、明示的な通信または保守的な行動モデルが必要である。

これらのトレードオフをハードウェア試験前にシミュレーションで定量化し、実地試験では遅延、予測誤差、安定性マージンを計測する。

35.2 凹凸のある地形を航行する

動的環境での再計画手法を踏まえ、ロボットは歩容選択と歩行実行時に空間的に変化する地面形状と局所的な接触不確定性を推論しなければならない。本小節では問題を定式化し、センシングと制御要件を分析し、足場選択と安全な歩容実行のための簡潔な実装スケッチを示す。

問題の記述と運用目標。

- ・問題：二足歩行ロボットは、足の寸法スケールで高さや傾斜が変化する地形を通過しなければならない。センサ測定にはノイズと遅延が伴う。アクチュエータ限界とバランス制約が実行可能な歩容を制限する。
- ・目標：(1) 滑りが少なく安定性の高い運動学的に到達可能な足場を選択、(2) 歩行中に重心（CoM）およびゼロ・モーメント・ポイント（ZMP）制約を満たすこと、(3) モデルおよび知覚の不確定性にリアルタイムで反応する。

地形表現とセンシング。

- ・地形を高さ関数 $h(x, y)$ とそれに付随する共分散 $\Sigma_h(x, y)$ として表現する。ステレオ深度、3D LiDAR、およびオンボード IMU による重力補正を融合して h を構築する。ロボット骨盤中心とした局所高さマップ `height_map` を定義する。
- ・セルごとに幾何学的特徴を推定する：局所法線 $\mathbf{n}(x, y)$ 、傾斜大きさ $\theta(x, y) = \arccos(\mathbf{n} \cdot \hat{\mathbf{z}})$ 、セル内標本の分散として計算される粗さ $r(x, y)$ 。

足場の実行可能性と安定性指標。

- 運動学的到達可能性：候補足位置は遊脚の逆運動学（IK）および関節限界制約を満たす必要がある。
- 接触安定性：局所接触面の向きと摩擦を評価する。小さな平面状の足の場合、予期される衝撃と定常剪断力を面に投影して必要な接線力を近似する。
- 複合コスト関数を用いて足場をランク付けする。傾斜許容度、粗さ、到達余裕、安定余裕をトレードオフする重みを設計する。よく用いられる形は：

$$[H]C(x, y) = w_s \cdot \theta(x, y) + w_r \cdot r(x, y) + w_k \cdot d_{ik}(x, y) + w_z \cdot \max(0, d_{zmp}(x, y)), \quad (292)$$

ここで d_{ik} は IK 残差または到達可能領域までの距離、 d_{zmp} は予測 ZMP の支持多角形外の符号付き距離である。

バランスと動的制約。

- 簡易予測歩行のための線形倒立振子キャプチャポイント [?] を定義する：

$$\xi = x + \frac{v}{\omega_0}, \quad \omega_0 = \sqrt{\frac{g}{z_{\text{CoM}}}}, \quad (293)$$

現在の CoM 水平位置 x と速度 v を用いる。実行可能な歩容は、歩容完了後にキャプチャポイントが新たな支持多角形内に入るように足を配置しなければならない。

- 遊脚期間中は全身制御により ZMP 制約を課す。瞬時 ZMP \mathbf{p}_{ZMP} は $\mathbf{p}_{\text{ZMP}} \in \mathcal{P}_{\text{support}}$ を満たし、二次計画（QP）コントローラ内の線形不等式制約として実現できる。

計画アプローチと不確定性対処。

- 歩容配置と CoM 軌道のための短時間ホライズンモデル予測コントローラ（MPC）を用いる。MPC は 1-3 歩のホライズンにわたり歩行タイミングと足位置を最適化する。
- 測定共分散によりコストを膨張させて知覚不確定性を考慮する。 (x, y) の候補に対し、 $\text{trace}(\Sigma_h(x, y))$ に比例したペナルティを加える。
- 反応的再計画：遊脚中に足場候補を評価し、測定地形が閾値を超えて逸脱した場合に再計画する。

アルゴリズム概要。

1. センサデータを取得・融合し、height_map と不確定性を更新する。
2. 運動学的到達範囲内に候補足セルのグリッドを生成する。
3. 各候補に対して評価する：
 - 傾斜 θ 、粗さ r 、
 - IK 実行可能性 d_{ik} 、
 - 予測 ZMP 逸反 d_{zmp} 、
 - 不確定性ペナルティ。
4. 関節・接触制約を満たす最小コスト系列を選択する短時間ホライズン MPC/QP を解く。
5. 全身コントローラで遊脚を実行し接触を監視する。予期せぬ接触または滑りが発生したら中断して再計画する。

実装スケッチ。

コードサンプル 119 候補足場生成とスコアリング (簡略版)

```

import numpy as np
from typing import List, Dict, Tuple, Optional
import logging

# グローバルロガー
logger = logging.getLogger(__name__)

def score_candidates(
    height_map: np.ndarray,
    cov_map: np.ndarray,
    robot_model,
    params: Dict[str, float],
    map_index_to_world: callable,
    estimate_normal: callable,
    local_roughness: callable,
) -> List[Dict[str, Tuple[float, float, float] | float]]:
    """
    高さマップと共分散マップから次の脚位置候補をコストでソートして返す。
    """
    if height_map.shape != cov_map.shape[:2]:
        raise ValueError("height_mapとcov_mapの形状が不一致")

    candidates: List[Dict] = []

    # マルチスレッド化を見越してループはシンプルに保つ
    for (i, j), h in np.ndenumerate(height_map):
        try:
            x, y = map_index_to_world(i, j)
            n = estimate_normal(height_map, i, j)
            theta = np.arccos(np.clip(np.dot(n, [0, 0, 1]), -1.0, 1.0))
            r = local_roughness(height_map, i, j)

            reachable, d_ik = robot_model.ik_reachability(x, y, h)
            if not reachable:
                continue

            zmp_pen = predict_zmp_violation(robot_model, x, y, h)

```

```

uncert = float(np.trace(cov_map[i, j]))

cost = (
    params["ws"] * theta
    + params["wr"] * r
    + params["wk"] * d_ik
    + params["wz"] * max(0.0, zmp_pen)
    + params["wu"] * uncert
)

candidates.append({"pos": (x, y, h), "cost": cost})
except Exception as e:
    # セーフティ：予期せぬエラーで全体を止めない
    logger.debug(f"skip_cell_{(i,j)}:{e}")
    continue

# コスト昇順でソート
return sorted(candidates, key=lambda c: c["cost"])

```

制御統合と制約。

- 軌道追従、関節限界、接触力を統合する全身 QP を用いる：
 - 追従誤差と駆動努力を最小化する。
 - 線形化された力学、一方向接触制約、摩擦円錐近似を考慮する。
- 摩擦円錐を多面体近似により線形化し、接触力に対する線形不等式 $A_f f \leq b_f$ を得る。

設計上のトレードオフと運用上のリスク。

- トレードオフ：
 - 足場探索領域を大きくすると頑健性は向上するが計算量と遅延が増大する。
 - 重量級センサスイート（LiDAR + ステレオ）は知覚を改善するが電力とペイロードが増加する。
 - 柔軟・コンプライアントな足部は接触許容性を高めるが ZMP の精密制御は低下する。
- リスク：
 - *height_map* の機体座標系への位置合わせ誤差は危険な足衝突を引き起こす。
 - モデル化されていない表面コンプライアンスは沈み込みと ZMP 誤差を過小評価する。
 - 過度の再計画頻度は CPU を過負荷させ制御更新を遅延させる。

エンジニアはセンサ精度、オンボード演算、機械的コンプライアンスをミッション固有の安全余裕に見合うようバランスさせるべきである。

35.3 センサノイズへの対処

不整地での踏み付け位置推定や動的環境での運動予測に続き、センサノイズが位置推定・知覚品質の主要な制限要因となる。小さなノイズ誘起誤差を放置すると、ヒューマノイド歩容や経路計画において大きな姿勢誤差に累積する。

問題定義. ヒューマノイドロボットはIMU、関節エンコーダ、ステレオまたはRGB-Dカメラ、ライダ、足部力・トルクセンサ、場合によっては磁気センサなどの異種センサに依存する。各センサは以下をもたらす：

- 確率的ノイズ（ゼロ平均、しばしばガウス分布としてモデル化）；
- 系統的バイアス（ゆっくり変化するオフセット）；
- 文脈依存のアーティファクト（照明、マルチパス、接触振動）。

技術的目標は、CPU、レイテンシ、プラットフォーム振動の制約下で信頼性の高い状態推定とマッピングを行うことである。

技術解析. ほとんどのフィルタで用いられる離散時間プロセス・観測モデルを表現する：

$$[H]x_{k+1} = f(x_k, u_k) + w_k, \quad z_k = h(x_k) + v_k, \quad (294)$$

ここで $w_k \sim \mathcal{N}(0, Q_k)$ はプロセスノイズ、 $v_k \sim \mathcal{N}(0, R_k)$ は観測ノイズを表す。線形化更新のために、最適線形推定器（カルマンフィルタ）はカルマンゲインを計算する

$$[H]K_k = P_{k|k-1}H_k^\top (H_k P_{k|k-1}H_k^\top + R_k)^{-1}, \quad (295)$$

その後、状態と共分散を更新する。

ヒューマノイドナビゲーションの要点：

- IMU：ジャイロバイアス b_g と加速度バイアス b_a をモデル化し、連続から離散へのノイズ積分で不確実性を伝播する。Allan 分散測定を用いてノイズパラメータを決定する。
- Vision：特徴ごとの観測共分散とロバストマッチ棄却（RANSAC）の両方を含める。ローリングシャッタとモーションブラーは実効的な R を増大させる。
- 足部接触・関節エンコーダ：量子化とバックラッシュが非ガウスノイズを生じる。接触イベントをハイブリッド観測として扱い、接触確率をモデル化する。
- Lidar：距離依存ノイズ；近接面は遠距離戻りより分散が小さい。マルチパスは非ガウス外れ値を引き起こす。

センサ融合戦略と可観測性. 可観測性に基づいて融合アーキテクチャを選択し、以下を計算する：

1. 高速IMU姿勢用相補フィルタと低速視覚またはライダドリフト補正。
2. 高周波足部衝撃を含むヒューマノイド運動向けにIMU事前積分を備えた Visual-Inertial Odometry (VIO)。
3. 計算を制限するための状態周辺化と共に、マップ整合性のためのスライディングウィンドルバンドル調整。
4. 長期ドリフト補正のためのファクタグラフまたはポーズグラフ最適化。

外れ値と非ガウスノイズへのロバスト性. 技術：

- 非確実観測を棄却するマハラノビスゲーティング： $(z - h(\hat{x}))^T S^{-1} (z - h(\hat{x})) < \gamma$ なら受理。
- 非線形最小二乗内でロバスト損失関数（Huber、Cauchy）を用いて単一点影響を低減。
- 視覚またはライダマッチングでの幾何学的関連付け向け RANSAC。

実装上の考慮事項. リアルタイムヒューマノイドシステムはセンサレートとレイテンシを管理する必要がある。IMU 更新を高レートで実行し、視覚またはライダ更新を低レートで受信する非同期マルチレート融合を用いる。可能な場合は厳密なタイムスタンプ付与とハードウェアレベル同期を確保する。

例：マハラノビスゲーティングと適応共分散スケーリングを備えた単純線形観測更新。以下のコードスニペットは、高レベル融合モジュールに適した実用的な棄却・更新ステップを示す。

コードサンプル 120 カルマン観測更新：マハラノビスゲーティングと適応共分散スケーリング付き

```
import numpy as np
from typing import Tuple, Optional

def kalman_measurement_update(
    x: np.ndarray,
    P: np.ndarray,
    z: np.ndarray,
    H: np.ndarray,
    R: np.ndarray,
    gate_thresh: float = 9.21,
    adaptive: bool = True,
    eps: float = 1e-9
) -> Tuple[np.ndarray, np.ndarray, Optional[float]]:
    """
    カルマン観測更新（外れ値ゲート付き）
    """
    z_pred = H @ x
    y = z - z_pred
    S = H @ P @ H.T + R

    # 数値安定なコレスキー分解で逆行列を回避
    try:
        L = np.linalg.cholesky(S)
        maha = float(y.T @ np.linalg.solve(S, y))
    except np.linalg.LinAlgError:
        # 正定でなければ疑似逆行列でフォールバック
        S_inv = np.linalg.pinv(S, rcond=eps)
        maha = float(y.T @ S_inv @ y)
```

```

# 外れ値ゲート
if maha > gate_thresh and adaptive:
    scale = maha / gate_thresh
    R *= scale ** 2 # 共分散を大きくして信頼度を下げる
    S = H @ P @ H.T + R
    L = np.linalg.cholesky(S)

# カルマンゲイン (コレスキー分解で効率化)
K = np.linalg.solve(L, np.linalg.solve(L.T, H @ P.T)).T
x += K @ y
P = (np.eye(P.shape[0]) - K @ H) @ P

return x, P, maha

```

チューニングとキャリブレーション. センサノイズをオフラインでキャリブレーションする：

- IMU：Allan 分散を用いてホワイトノイズとバイアス不安定性を推定。
- カメラ：静止撮影を用いて画素ノイズ分散を計算。
- Lidar：距離と入射角の関数として距離分散をフィッティング。

オンライン適応は条件変化時に有効。手法：

- 適応カルマンフィルタ：残差統計を用いて R_k または Q_k を推定。
- 接触イベント後のフィルタ過信を避けるための共分散インフレーション。
- 故障検出：正規化イノベーション二乗和（NIS）を監視し安全動作をトリガ。

計算上のトレードオフ. スライディングウィンドウ最適化はより良い大域整合性をもたらすが、CPU とメモリを多く消費する。EKF は軽量だが大きな非線形には劣る。ミッションプロファイルに応じてアーキテクチャを選択：

- 倉庫または製造用ヒューマノイド：予測可能な運動向け低レイテンシ EKF を優先。
- サービスまたは探索用ヒューマノイド：視覚ドリフト耐性のためスライディングウィンドウ付き VIO を選択。

技術的影響、トレードオフ、運用上のリスク。

- 過信共分散はフィルタ発散と歩容または衝突の潜在的危険を引き起こす。
- 反応不足フィルタは補正を遅延させドリフトを増大；不整地での歩行計画に影響。
- センサ冗長性は単一センサ故障を軽減するが、重量、電力、計算を増大。
- 厳密なタイムスタンプ付与とセンサの機械的隔離は相関振動ノイズを低減。
- 大きなイノベーションスパイクまたは位置推定喪失時に発動するウォッチドッグと安全転倒動作を実装。

設計上のトレードオフはサイズ、重量、計算、ミッション要求をバランスさせる必要がある。効果的なノイズ処理は、キャリブレート済みノイズモデル、ロバスト推定器、適応戦略を用いて安全かつ

信頼性の高いヒューマノイド自律性を確保する。

35.4 ロバスト性と効率の向上

前の小節では、計測不確実性と可変な接地位置が位置推定とバランスに与える悪影響を検討した。これらの分析は、計算・センシング・制御をトレードオフさせてロバスト性と効率の両方を高める統合戦略を促す。

問題定義。ヒューマノイドロボットは、バランスを保ちエネルギーを最小化しながら、がらんどうで動的な環境を横断しなければならない。ロバスト性とは、最悪の外乱やセンサ故障下でも安全性を維持することである。効率とは、安全性を損なうことなくエネルギー、遅延、計算を削減することである。工学上の問題は、不確実性、リソース制限、地形のばらつきを明示的に考慮した知覚・計画・制御スタックを設計することである。

技術分析。両目的を達成するには、3 レベルで調整された変更が必要である。

1. 知覚：確率的状態推定と信頼度指標を生成する。必要に応じてセンサ融合を用いて不確実性を縮小し、センサ損失下でも優雅に劣化する。
2. 計画：経路コストと探索ヒューリスティックにリスク指標を組み込み、高確率衝突を回避しながらエネルギー・時間効率の良いルートへ誘導する。
3. 制御：チューブベースモデル予測制御（MPC）などのロバストコントローラを用いて、計画軌道を追従しながら名義運動からの逸脱を境界付ける。

リスク考慮コスト定式化。安全性、エネルギー、時間をバランスさせる連続コスト汎関数を定義する：

$$[H]J = \int_{t_0}^{t_f} (\lambda_s c_{\text{safety}}(x(t)) + \lambda_e c_{\text{energy}}(u(t)) + \lambda_t) dt. \quad (296)$$

ここで c_{safety} は確率的地図から衝突確率を積分し、 c_{energy} は関節トルクからアクチュエータ電力を近似し、 λ_i は調整可能な重みである。衝突確率を明示的に記述することで、低リスク廊域を優先できる。

ガウスポーズ不確実性を伴う衝突確率。水平位置誤差がゼロ平均等方性ガウスで標準偏差 σ であり、最近障害物までのクリアランスが d_{\min} である局所化ヒューマノイドに対して、近似衝突確率は

$$[H]P_{\text{coll}} \approx 1 - \Phi\left(\frac{d_{\min} - r}{\sigma}\right), \quad (297)$$

である。ここで r はロボット半径、 Φ は標準正規 CDF である。この式はリスク閾値を導く：候補経路に対して $P_{\text{coll}} \leq p_{\max}$ を要求する。

チューブ MPC によるロバスト軌道追従。軌道 $x_{\text{nom}}(t)$ を生成する名義プランナを実装する。フィードバック則 $u = u_{\text{nom}} + K(x - x_{\text{nom}})$ を用いて実状態を名義軌道周りのチューブ内に閉じ込める。外乱上限 $\|w\|_{\infty} \leq w_{\max}$ を持つ LTI 線形化に対して、定常状態誤差上限は

$$[H]\|e\|_{\infty} \leq \|(I - (A + BK))^{-1}\|_{\infty} w_{\max}. \quad (298)$$

を満たす。この上限だけ状態制約を締めて、外乱下でも制約充足を保証する。

実装パターン（実践的選択）。

- ・ハイブリッドプランナ：大域プランナで粗い経路決めを行い、リスク考慮局所プランナで反応的衝突回避を行う。

- 適応センシング：障害物近傍や凹凸地形では LiDAR・ビジョンレートを上げる。自由空間移動中はセンシングを減らしてエネルギーを節約する。
- 任意時計画：漸進プランナ（例：D* Lite または RRT*）を計算上限付きで実行し、迅速に実行可能経路を提供し、時間の許す限り改善する。

具体的な算法的ビルディングブロック：

- リスク評価：(297) から候補セグメントごとに P_{coll} を計算。
- コスト集約：測定アクチュエータ電力モデルを用いて (296) を評価。
- コントローラ合成：線形化ダイナミクスに対して K をオフラインで計算し、(298) で締めた制約を課す MPC に組み込む。

コード例。スニペットは最小限のリスク重み付き局所ウェイポイントセレクトを示す。距離、衝突確率、推定エネルギーコストを組み合わせる候補ウェイポイントにスコアを付ける。実践ではおもちゃモデルをロボット固有の運動学・電力モデルに置き換える。

コードサンプル 121 Risk-weighted local waypoint scoring

```
import math
from typing import List, Tuple

import numpy as np
import numpy.typing as npt

class WaypointEvaluator:
    """ウェイポイントを安全性・エネルギー・進捗で評価"""

    def __init__(
        self,
        robot_radius: float = 0.35,
        pos_sigma: float = 0.15,
        lambda_s: float = 5.0,
        lambda_e: float = 1.0,
        lambda_d: float = 0.5,
        energy_factor: float = 10.0,
    ) -> None:
        self.r = robot_radius
        self.sigma = pos_sigma
        self.lambdas = np.array([lambda_s, lambda_e, lambda_d])
        self.energy_factor = energy_factor

    # 衝突確率を計算
```

```

def collision_prob(self, d_min: float) -> float:
    z = (d_min - self.r) / self.sigma
    return 1.0 - 0.5 * (1.0 + math.erf(z / math.sqrt(2.0)))

# エネルギー代用値
def energy_estimate(self, curr: npt.NDArray, goal: npt.NDArray) -> float:
    return float(np.linalg.norm(goal - curr)) * self.energy_factor

# スコアを算出
def score(
    self,
    curr: npt.NDArray,
    wp: npt.NDArray,
    d_min: float,
) -> float:
    p_coll = self.collision_prob(d_min)
    e_cost = self.energy_estimate(curr, wp)
    dist = float(np.linalg.norm(wp - curr))
    terms = np.array([p_coll, e_cost, dist])
    return float(np.dot(self.lambdas, terms))

# 最良ウェイポイントを選択
def select_best(
    self,
    curr: npt.NDArray,
    candidates: List[npt.NDArray],
    d_mins: List[float],
) -> Tuple[npt.NDArray, float]:
    scores = [self.score(curr, wp, dm) for wp, dm in zip(candidates, d_mins)]
    idx = int(np.argmin(scores))
    return candidates[idx], scores[idx]

# 使用例
if __name__ == "__main__":
    evaluator = WaypointEvaluator()

    curr = np.array([0.0, 0.0])
    candidates = [np.array([1.0, 0.0]), np.array([0.8, 0.3]), np.array([0.5, 0.9])]
    d_mins = [0.6, 0.4, 0.8]

```

```
best_wp, best_score = evaluator.select_best(curr, candidates, d_mins)
```

運用上の考慮事項とトレードオフ。

- 計算 vs 安全性：より厳しい確率制約は計算を増やす。任意時計画と優先センシングでバランスを取る。
- エネルギー vs 知覚精度：連続高レートセンシングは不確実性を減らすがバッテリーを消耗する。リスク指標に結びついた適応センサスケジューリングを用いる。
- コントローラの保守性：大きな誤差上限はより安全だが効率の悪い動作を生む。期待環境統計に合わせて K と外乱モデルを調整する。
- 現実世界のリスク：予期せぬ動的障害物、センサ遮蔽、不均一支持はガウス仮定を無効化する。実地試験で境界を検証し、フェイルセーフ停止挙動を実装する。

工学への影響。計画と制御内でリスクとエネルギーを定量化する層状アプローチを採用する。確率モデルを実証センサ誤差統計に対して検証する。優雅な劣化を設計する：高忠実度センシングが故障したら速度を下げ、クリアランスを増やし、保守的歩容モードに切り替える。これらの実践は、ミッションエネルギーと遅延を許容範囲内に保ちながら、ヒューマノイドの運用耐性を改善する。

ロボティクスのための強化学習

36 強化学習の基礎

36.1 重要概念：状態、行動、報酬

これまでに議論したシミュレーション設定とバランスメカニズムを基に、ヒューマノイドロボットの学習において用いられるコアな RL プリミティブを定式化する。これらのプリミティブは、制御目的を観測、指令、そして測定可能な学習信号にどう変換するかを決定する。

ヒューマノイド RL 問題は典型的にはマルコフ決定過程 (MDP) としてモデル化される。MDP は工学問題を簡潔に捉える：センサ情報が豊富な高次元状態を、ダイナミクスと接触制約の下で累積タスク性能を最大化するアクチュエータ指令に写像することである。形式的に、MDP はタプル $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ であり、 \mathcal{S} は状態、 \mathcal{A} は行動、 $P(s'|s, a)$ は遷移確率、 $R(s, a, s')$ は報酬、 $\gamma \in [0, 1)$ は割引率である。軌跡を評価する割引利得は

$$[H]G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad (299)$$

で与えられ、 r_t は時刻 t のスカラー報酬である。

状態設計（観測可能性と特徴選択）

- 問題定義：制御に必要な情報を含みつつ次元数を管理可能に保つ状態表現を選択する。
- ヒューマノイドにおける実用的構成要素：
 1. プロプライオセプション：関節位置 q 、関節速度 \dot{q} 、利用可能であればモータ電流またはトルク。
 2. 基部運動学：フローティングベース姿勢（例：クォータニオン）、IMU からの角速度、推定基

部並進速度。

3. 接触情報：足接触フラグ、接触力、推定圧力中心 (CoP)。
4. タスク特異的外部知覚：depth から得た障害物ベクトル、視覚的ターゲット位置、または人間との相互作用時の人間姿勢。
5. 導出特徴：重心 (CoM) 高さ、零力矩点 (ZMP) 誤差、周期的タスクのための位相変数。

工学上の考慮事項：

- 高周波プロプライオセプティブ信号は安定性に不可欠である。ローパスフィルタと一貫したサンプリングでエイリアシングを削減する。
- 部分的観測は一般的であり、短い履歴ウィンドウを含めるか、再発ポリシー (LSTM) を用いて時間的コンテキストを捉える。

行動空間の選択 (ローレベル対ハイレベル)

- ローレベル連続行動：直接関節トルク τ または速度設定値。微細な動的制御を可能にするが、正確なアクチュエータモデルと安定した方策学習を要する。
- ミドルレベル行動：ローレベル PD コントローラへ渡す関節位置または速度目標。動的表現力を犠牲にして学習を単純化する。
- ハイレベル抽象化：足踏み目標、所望 CoM 軌道、または有限パラメータ歩容生成器。ハードウェア安全性や接触計画が支配的な場合に有用。
- 離散行動：連続バランシングでは稀だがモード切替 (例：立つ／歩く／座る) には有用。

トレードオフ：

- トルク行動はエネルギー効率の良いコンプライアント動作を許容する。サンプル複雑性とシミュレーション忠実度要件を増大させる。
- 位置目標行動は学習を加速させ、アクチュエータダイナミクスへの過学習を削減し、シムツォリアル転移を容易にする。

報酬設計 (タスクシェイピングと頑健性)

- 問題定義：接触と外乱下で安全で頑健な動作へエージェントを導くスカラー信号を作成する。
- ヒューマノイドバランスおよび歩行の典型的報酬構成要素：
 1. タスク進行：前進速度追従または目標到達報酬。
 2. 安定性：負の ZMP 誤差または CoM 高さ逸散ペナルティ。
 3. エネルギー効率：二乗トルク $\sum \tau_i^2$ または消費電力をペナルティ。
 4. 滑らかさ：高いジャークまたは大きな行動差分をペナルティ。
 5. 安全性：転倒または関節限界違反に対する大きな負ペナルティ。
- 重み付き和で報酬を結合する。カリキュラムまたはシェイプ報酬は意図しない局所最適を避けるため慎重に用いる。

簡潔なベルマン期待は方策 π 下での期待状態価値を表現する：

$$[H]V^\pi(s) = \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s]. \quad (300)$$

この式はヒューマノイド RL で用いられる価値ベースおよび方策勾配法の基礎となる。

実装スケッチ：状態および報酬構築以下のリストは Isaac Sim 内または Gym ラッパー内で頻繁に実行される実用的 Python パターンを示す。コンパクトな状態ベクトルを構成し、シェイプされた報酬を計算する。コメントは簡潔で目的指向である。

コードサンプル 122 ヒューマノイドバランス用状態ベクトルと報酬の構築

```
import numpy as np
from typing import Sequence, Union, Optional

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import JointState, Imu
from geometry_msgs.msg import Vector3
from std_msgs.msg import Bool

class StateEstimator(Node):
    """
    ROS2 ノードとして動作する状態推定器。
    シミュレータ／実機のセンサ値を購読し、正規化された状態ベクトルを返す。
    """

    def __init__(self, joint_names: Sequence[str], foot_names: Sequence[str],
                  nominal_height: float = 1.0,
                  node_name: str = "state_estimator") -> None:
        super().__init__(node_name)

        self.joint_names = joint_names
        self.foot_names = foot_names
        self.nominal_height = nominal_height

        # 最新センサ値を保持
        self.q: Optional[np.ndarray] = None
        self.qd: Optional[np.ndarray] = None
        self.imu_ori: Optional[np.ndarray] = None
        self.imu_omega: Optional[np.ndarray] = None
        self.com: Optional[np.ndarray] = None
        self.foot_contacts: Optional[np.ndarray] = None

        # 購読
```

```

        self.create_subscription(JointState, "joint_states", self._cb_joint, 1)
        self.create_subscription(Imu, "imu/data", self._cb_imu, 1)
        self.create_subscription(Vector3, "com_est", self._cb_com, 1)
        for name in foot_names:
            self.create_subscription(Bool, f"{name}/contact", self._cb_contact(name),

def _cb_joint(self, msg: JointState) -> None:
    # 関節名順にソートしてから格納
    order = [msg.name.index(n) for n in self.joint_names]
    self.q = np.array([msg.position[i] for i in order], dtype=np.float32)
    self.qd = np.array([msg.velocity[i] for i in order], dtype=np.float32)

def _cb_imu(self, msg: Imu) -> None:
    o = msg.orientation
    self.imu_ori = np.array([o.x, o.y, o.z, o.w], dtype=np.float32)
    w = msg.angular_velocity
    self.imu_omega = np.array([w.x, w.y, w.z], dtype=np.float32)

def _cb_com(self, msg: Vector3) -> None:
    self.com = np.array([msg.x, msg.y, msg.z], dtype=np.float32)

def _cb_contact(self, name: str):
    def cb(msg: Bool) -> None:
        idx = self.foot_names.index(name)
        if self.foot_contacts is None:
            self.foot_contacts = np.zeros(len(self.foot_names), dtype=np.float32)
        self.foot_contacts[idx] = float(msg.data)
    return cb

def ready(self) -> bool:
    return all(x is not None for x in [self.q, self.qd, self.imu_ori,
                                       self.imu_omega, self.com, self.foot_contacts])

def get_state(self) -> np.ndarray:
    """
    正規化済み状態ベクトルを返す。ready()==Falseの場合は空配列。
    """
    if not self.ready():
        return np.empty(0, dtype=np.float32)

```

```

com_height = self.com[2]
zmp_err = 0.0 # 外部ノードで計算済みと仮定
state = np.concatenate([
    self.q,
    self.qd,
    self.imu_ori,
    self.imu_omega,
    [com_height, zmp_err],
    self.foot_contacts
]).astype(np.float32)
return state

```

```

class RewardCalculator:

```

```

    """

```

```

    状態・行動から報酬を計算。転倒判定はfoot_contact_sum==0。

```

```

    """

```

```

    def __init__(self, nominal_height: float = 1.0,
                  w_stab: float = 2.0,
                  w_height: float = 1.0,
                  w_energy: float = 0.001,
                  w_fall: float = 100.0) -> None:

```

```

        self.nominal_height = nominal_height
        self.w_stab = w_stab
        self.w_height = w_height
        self.w_energy = w_energy
        self.w_fall = w_fall

```

```

    def __call__(self, state: np.ndarray, action: np.ndarray,
                  next_state: np.ndarray) -> float:

```

```

        n_foot = len(next_state) - 7 # 末尾: [com_height, zmp_err, foot_contacts...]
        foot_contacts = next_state[-n_foot:]
        com_height = next_state[-(n_foot + 2)]
        zmp_err = next_state[-(n_foot + 1)]

```

```

        stability = -abs(zmp_err)
        height_pref = -abs(com_height - self.nominal_height)
        energy_penalty = -self.w_energy * np.sum(np.square(action))
        fall_penalty = -self.w_fall if foot_contacts.sum() == 0 else 0.0

```

```

reward = (self.w_stab * stability +
           self.w_height * height_pref +
           energy_penalty +
           fall_penalty)
return float(reward)

```

運用上の注意、トレードオフ、リスク

- 報酬スパース性は学習時間を増大させ脆い方策を引き起こす。中間シェイピングまたはデモンストラクションでブートストラップする。
- 高次元状態は正則化と慎重な正規化を要し、虚偽相関を避ける。
- 行動空間選択はシムツーリアル転移に影響する：位置目標コントローラは生トルク方策よりも頻繁に確実に転移する。
- 安全臨界展開では危険状態に対する明示的ペナルティ項とフォールバックローレベルコントローラを含めるべきである。
- 過度に複雑な報酬構造は故障モードを隠蔽する。厳選されたテスト外乱と敵対的摂動で検証する。

設計者はサンプル効率、頑健性、転移可能性をバランスさせながら状態、行動、報酬を選択しなければならない。各選択は最適化風景を変え展開時のハードウェアリスクに影響する。

36.2 探索と活用のトレードオフ

前小節では、ヒューマノイド制御問題を定義する MDP の要素——状態、行動、報酬——を確立した。その形式化を基に、本小節では学習エージェントが安全性およびハードウェア制約の下で、新しい動作の探索と既知の高報酬行動の活用をどのようにバランスさせるかを考察する。

問題設定. ヒューマノイド制御において探索は、タスク性能または頑健性を改善する新規方策を求める。活用は現在の方策を用いて測定報酬を最大化する。過度の探索は損傷、不安定化、転倒を招くリスクがあり、早熟な活用は局所最適に学習を閉じ込めるため、このトレードオフは極めて重要である。形式的に、エピソードホライズン T とステップ報酬 r_t が与えられるとき、累積リグレットは

$$[H]R_T = \sum_{t=1}^T (r_t^* - r_t), \quad (301)$$

で与えられ、 r_t^* は最適方策下での報酬である。 R_T を最小化するには、危険な行動を制限しつつ情報の多いデータを収集する戦略が必要である。

技術分析. 実用的な探索スキームは、不確かな状態-行動対を訪問するよう行動選択方策 $\pi(a|s)$ を改変する。主要なメカニズムは以下の通りである：

- 価値摂動. 加法的探索ボーナスは楽観的価値推定を生む：

$$[H]Q^+(s, a) = Q(s, a) + \beta \sqrt{\frac{\log N(s)}{N(s, a) + 1}}, \quad (302)$$

ここで $N(s, a)$ は訪問回数を数え、 β は楽観度を調整する。これは離散行動設定およびバランスプリミティブの離散化制御器で有効である。

- 確率の方策. Boltzmann (softmax) サンプリング

$$[H]\pi(a|s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)} \quad (303)$$

は温度 τ を用いて探索と活用を補間する。低い τ は活用を優先する。

- パラメータ空間ノイズ. 方策パラメータを出力ではなく摂動することで、時間的に一貫した探索的軌道を生成する。これはヒューマノイドの安定性に必要な滑らかな関節指令を保持する。
- 内在的動機付け. 予測誤差または状態訪問新奇性に比例した内部報酬 r^{int} を加える。エージェントは $r^{\text{ext}} + \eta r^{\text{int}}$ を最適化し、 η がタスクと好奇心をトレードする。

安全性制約は以下の制約付き最適化問題を導入する：

$$[H] \max_{\pi} \mathbb{E} \left[\sum_t r_t \right] \quad \text{s.t.} \quad \mathbb{E} \left[\sum_t c_t \right] \leq C_{\max}, \quad (304)$$

ここで c_t はコスト信号（例：トルクリミット、足滑りイベント）であり、 C_{\max} は許容リスクを上限とする。制約付き RL ソルバ (Lagrangian, CPO) は安全性を強制しつつ探索を許容する。

実装プリミティブとトレードオフ. 不整地踏み出しなどのヒューマノイドタスクでは、ハードウェアの脆弱性を考慮して探索プリミティブとスケジュールを選ぶ：

- アニール epsilon-greedy: 単純で離散モータプリミティブ選択に有効。イプシロンはエピソードごとに ϵ_0 から ϵ_{\min} へ減衰する。
- Boltzmann サンプリング: 離散化後の連続行動に対して滑らかな確率分布を与える、または SAC スタイルの確率の方策を経由する。
- アンサンブルとブートストラップ: 認識的不確実性を推定して指向的探索を行う；計算負荷は高いが制約チェックと組み合わせることでより安全。
- 安全行動射影: サンプルされた行動を安定性コントローラまたは安全フィルタを通して実行前にフィルタリングする。

以下の実用的 Python スニペットは、アニール epsilon-greedy ラップに安全射影を実装したものである。ラップは方策行動とランダム探索行動を選択し、射影は関節リミットと最大許容トルクを強制する。環境インタフェースを置き換えることで Isaac Sim または実機コントローラと統合できる。

コードサンプル 123 Epsilon-greedy ラップ with projection for humanoid action safety.

```
import numpy as np
import torch
import gymnasium as gym
from typing import Protocol, Tuple, Any

class PolicyProtocol(Protocol):
    """Policy must expose deterministic act()."""
    def act(self, state: np.ndarray) -> np.ndarray: ...
```

```

class ZMPModelProtocol(Protocol):
    """ZMPpredictor must expose predict() and is_within_support()."""
    def predict(self, state: np.ndarray, action: np.ndarray) -> np.ndarray: ...
    def is_within_support(self, zmp: np.ndarray) -> bool: ...


class EpsilonGreedySafe:
    """
    ε-greedy探索戦略に安全射影を組み込んだ行動選択クラス
    """

    def __init__(
        self,
        policy: PolicyProtocol,
        env: gym.Env,
        eps_start: float = 0.5,
        eps_min: float = 0.05,
        decay: float = 1e-4,
        zmp_model: ZMPModelProtocol | None = None,
        torque_rate_limit: float | None = None,
    ) -> None:
        self.policy = policy
        self.env = env
        self.eps = eps_start
        self.eps_min = eps_min
        self.decay = decay
        self.zmp_model = zmp_model
        self.torque_rate_limit = torque_rate_limit
        self._last_action: np.ndarray | None = None

    def reset(self) -> None:
        """エピソード開始時に内部状態をリセット"""
        self._last_action = None

    def step(self, state: np.ndarray, step_count: int) -> np.ndarray:
        # ε を指数減衰
        self.eps = max(self.eps_min, self.eps * (1.0 - self.decay))

        if np.random.rand() < self.eps:

```

```

        action = self.env.action_space.sample() # 安全なランダム探索
    else:
        action = self.policy.act(state)

    return self._safety_projection(action, state)

def _safety_projection(self, action: np.ndarray, state: np.ndarray) -> np.ndarray:
    # 関節限界クリップ
    low = self.env.action_space.low
    high = self.env.action_space.high
    action = np.clip(action, low, high)

    # トルク変化率制限
    if self.torque_rate_limit is not None and self._last_action is not None:
        delta = np.clip(
            action - self._last_action,
            -self.torque_rate_limit,
            self.torque_rate_limit,
        )
        action = self._last_action + delta

    # ZMP 安定性チェック
    if self.zmp_model is not None and self._predict_instability(state, action):
        action *= 0.5 # 保守的にスケールダウン

    self._last_action = action.copy()
    return action

def _predict_instability(self, state: np.ndarray, action: np.ndarray) -> bool:
    zmp = self.zmp_model.predict(state, action)
    return not self.zmp_model.is_within_support(zmp)

```

設計・チューニング指針：

- 式 (2) の β を選び、訪問の少ない行動へ探索を偏らせつつ攻撃的な動作を誘発しない。
- 多自由度ヒューマノイドにはパラメータ空間ノイズを用い、一貫した探索的軌道を得る。
- 内在報酬と信頼領域更新を組み合わせ、破局的な方策シフトを回避する。
- 実行時に安全フィルタを適用し、実世界制約を強制する。

エンジニアリングへの影響、トレードオフ、運用上のリスク：

- トレードオフ: 指向的探索はサンプル複雑性を減らすのが、信頼できる不確実性推定を要する。確

率的方策は実装が容易だが高周波関節指令を生じ、摩耗を増大させる。

- ・ハードウェアリスク: 無制限探索はアクチュエータリミットを超えたり姿勢を不安定化し、転倒・損傷を引き起こす。
- ・運用制約: 現地展開では探索をシミュレーションと安全な転移学習手法（保守的ファインチューニング、オンラインリスクモニタ等）に限定する。
- ・リソーストレードオフ: アンサンブル不確実性モデルは安全性を向上させるが計算・シミュレーション時間を増大させ、学習スループットに影響する。

設計者はサンプル効率、計算コスト、安全性をバランスさせる必要がある。探索戦略の選択には、ヒューマノイドダイナミクス、センシング精度、許容運用リスクを同時に考慮することが求められる。

36.3 ロボットのための報酬関数の設計

これらの設計推奨事項は、直前の状態-行動-報酬のセマンティクスと探索-活用のトレードオフに関する議論を直接踏まえている。報酬関数は、先に導入したタスク目的を符号化するとともに、ヒューマノイドロボットにとって安全で転移可能な行動へ探索を誘導しなければならない。

問題定義. 報酬関数は、タスク目標と安全制約を学習アルゴリズムへのスカラー報酬に変換する主要なエンジニアリング入力である。ヒューマノイドロボットでは、タスク遂行（移動、操縦）、安定性（バランス、足の接触）、効率性（エネルギー、作動器限界）、安全性（自己衝突、過大な力）という複数の目的をバランスさせる必要がある。設計の悪い報酬は、報酬ハッキング、脆いコントローラ、シミュレーションアーティファクトを悪用する方策を引き起こす。

技術的分析. 報酬を解釈可能な成分の重み付き和として扱う。各項を、ハードウェアでも取得可能なセンサまたはシミュレーション状態からの測定可能信号に対応するように定義する。典型的な構成は

$$[H]R(s, a, s') = \sum_i w_i r_i(s, a, s') + r_{\text{survival}} - r_{\text{fall}}, \quad (305)$$

であり、各成分 r_i は異なるエンジニアリング目的を対象とし、 $w_i > 0$ は調整可能な重みである。ヒューマノイドロボットに対する典型的な成分は以下の通り：

- ・直立性：起立した胴体の向きを促進する。ワールド z 単位ベクトルと胴体 z 軸を用いて計算し、 $r_{\text{upright}} = \max(0, n_{\text{torso}} \cdot z_{\text{world}})$ を得る。
- ・COM 追従：重心速度が所望速度 v^* に従うことを奨励する。 $r_{\text{com}} = -\alpha \|v_{\text{com}} - v^*\|^2$ を用いて逸脱をペナルティする。
- ・歩行精度：所望の足配置または位相での踵-つま先接触を報酬する。離散的な接触インジケータを用いて報酬項を形成する。
- ・エネルギーペナルティ：作動器トルクの二乗和をペナルティし、摩耗と発熱を低減する： $r_{\text{energy}} = -\beta \sum_j \tau_j^2$ 。
- ・接触安全：大きな接触インパルス大きさまたは禁止リンクへの接触をペナルティする： $r_{\text{contact}} = -\gamma \sum_c \|I_c\|^2$ 。
- ・タスク進行：ウェイポイントまたはターゲットへの前進進行。ポテンシャル関数 $\Phi(s)$ として表現する。最適方策を変えないようにポテンシャルベースのシェイピングを用いる：

$$[H]r_{\text{shape}}(s, s') = \gamma \Phi(s') - \Phi(s), \quad (306)$$

割引率 γ を伴う。ポテンシャルベースのシェイピングは方策不変性を保ちながら学習を加速する。

正規化とスケールリング. 各 r_i をその典型的な大きさの推定値で正規化してから重みを適用する。勾配を適切に保つため、オンライン正規化には実行平均と分散を用いる。組み合わせた報酬の大きさがタイムステップごとに $O(1)$ となるように重みを選ぶ。経験的に $\sum_i w_i \approx 1$ とし、感度分析により相対比を調整する。

共通の落とし穴の回避. 高密度報酬は学習を速めるが、長期目的と矛盾する局所最適を招くことがある。希薄報酬は探索を強制するが、慎重に設計されたカリキュラムまたは補助タスクを必要とする。報酬ハッキングを軽減するには：

- 意図しない近道を生む特権的シミュレータ変数へのアクセスを削除または制限する。
- IMU 姿勢、関節エンコーダ、力センサ、関節電流といったハードウェアで利用可能なセンサ由来信号を優先する。
- 関節速度に対するスペクトルエネルギー項を追加することで、長時間高周波振動といった虚偽の行動を明示的にペナルティする。

実装：実用的な報酬計算. 以下に、シミュレートされたヒューマノイド歩行タスクで複合報酬を計算する簡潔な Python 例を示す。コードは正規化、ポテンシャルベースのシェイピング、安全ペナルティを示す。

コードサンプル 124 Composite reward computation for humanoid walking (simulation).

```
import numpy as np
from typing import Dict, Any, Optional

import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32MultiArray

class RewardCalculator(Node):
    """
    ROS2 ノードとして報酬計算を提供。
    トピック経由で観測・行動を受信し、報酬を返す。
    """

    def __init__(self, node_name: str = "reward_calculator") -> None:
        super().__init__(node_name)

        # パラメータ宣言 & 取得
        self.declare_parameters(
            namespace="",
```

```

parameters=[
    ("v_target", [1.0, 0.0, 0.0]),
    ("alpha", 1.0),
    ("beta", 1e-4),
    ("gamma", 1e-3),
    ("foot_bonus", 5.0),
    ("discount", 0.99),
    ("forward_axis", [1.0, 0.0, 0.0]),
    ("w_upright", 1.0),
    ("w_com", 1.0),
    ("w_energy", 1.0),
    ("w_contact", 1.0),
    ("w_foot", 1.0),
    ("w_shaping", 1.0),
    ("fall_penalty", 100.0),
    ("reward_clip", 10.0),
],
)
self.params: Dict[str, Any] = {
    key: self.get_parameter(key).value for key in (
        "v_target", "alpha", "beta", "gamma", "foot_bonus", "discount",
        "forward_axis", "w_upright", "w_com", "w_energy", "w_contact",
        "w_foot", "w_shaping", "fall_penalty", "reward_clip"
    )
}
self.v_target = np.asarray(self.params["v_target"], dtype=np.float32)
self.forward_axis = np.asarray(self.params["forward_axis"], dtype=np.float32)

# 実行統計の初期化
self.running_stats: Dict[str, float] = {
    "upright_mean": 0.0,
    "upright_std": 1.0,
    "com_scale": 1.0,
}

# 購読・配信
self.obs_sub = self.create_subscription(
    Float32MultiArray, "/observation", self._obs_callback, 10
)
self.action_sub = self.create_subscription(

```

```

        Float32MultiArray, "/action", self._action_callback, 10
    )
    self.reward_pub = self.create_publisher(Float32MultiArray, "/reward", 10)

    self._latest_obs: Optional[Dict[str, np.ndarray]] = None
    self._latest_action: Optional[np.ndarray] = None

def _obs_callback(self, msg: Float32MultiArray) -> None:
    # メッセージを辞書にデコード
    obs = self._decode_obs(msg)
    self._latest_obs = obs
    self._try_compute()

def _action_callback(self, msg: Float32MultiArray) -> None:
    self._latest_action = np.asarray(msg.data, dtype=np.float32)
    self._try_compute()

def _try_compute(self) -> None:
    if self._latest_obs is None or self._latest_action is None:
        return
    reward = self.compute_reward(
        self._latest_obs, self._latest_action, self.running_stats, self.params
    )
    out_msg = Float32MultiArray()
    out_msg.data = [reward]
    self.reward_pub.publish(out_msg)

@staticmethod
def _decode_obs(msg: Float32MultiArray) -> Dict[str, np.ndarray]:
    # 簡易デコーダ：実際のメッセージ定義に合わせて調整
    arr = np.asarray(msg.data, dtype=np.float32)
    return {
        "torso_z": arr[0:3],
        "com_vel": arr[3:6],
        "com_pos": arr[6:9],
        "contact_impulses": arr[9:13],
        "correct_foot_phase": arr[13],
        "fell": bool(arr[14]),
    }

```

```

def compute_reward(
    self,
    obs: Dict[str, np.ndarray],
    action: np.ndarray,
    running_stats: Dict[str, float],
    params: Dict[str, Any],
) -> float:
    """
    観測と行動から報酬を計算。
    """
    # 起立性報酬
    upright = max(0.0, float(np.dot(obs["torso_z"], np.array([0.0, 0.0, 1.0]))))

    # 重心速度追従
    v_err = obs["com_vel"] - self.v_target
    com_term = -params["alpha"] * float(np.dot(v_err, v_err))

    # エネルギー消費ペナルティ
    energy_term = -params["beta"] * float(np.sum(np.square(action)))

    # 接触衝撃ペナルティ
    contact_term = -params["gamma"] * float(np.sum(np.square(obs["contact_impulses"])))

    # 足接地位相報酬
    foot_reward = params["foot_bonus"] * float(obs["correct_foot_phase"])

    # ポテンシャルベース shaping
    phi_s = float(np.dot(obs["com_pos"], self.forward_axis))
    # 次状態が無い場合同一フレームで近似（実環境では次状態を受け取る）
    shaping = (params["discount"] - 1.0) * phi_s

    # 正規化
    upright_n = (upright - running_stats["upright_mean"]) / (
        running_stats["upright_std"] + 1e-8
    )
    com_n = com_term / (running_stats["com_scale"] + 1e-8)

    # 重み付き合計
    reward = (
        params["w_upright"] * upright_n

```

```

        + params["w_com"] * com_n
        + params["w_energy"] * energy_term
        + params["w_contact"] * contact_term
        + params["w_foot"] * foot_reward
        + params["w_shaping"] * shaping
    )

    # 転倒ペナルティ
    if obs["fell"]:
        reward -= params["fall_penalty"]

    # クリップ
    return float(np.clip(reward, -params["reward_clip"], params["reward_clip"]))

def main(args=None):
    rclpy.init(args=args)
    node = RewardCalculator()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()

```

運用上の関連性. 上記の構造でシミュレーションで学習する際、ロボットのハードウェアとセンサ利用可能性の等価性を強制する。必要に応じてシミュレータ専用信号を推定器に置き換える。決定論的設定を悪用する方策を防ぐため、環境パラメータをランダム化する。

エンジニアリング上のトレードオフとリスク. 報酬の複雑さは解釈性を高めるが、調整コストを上昇させる。エネルギーに対する高いペナルティは、不整地で脆弱になることがある効率的な歩容を生む。強いシェイピングは収束を速めるが、最適でない移動プリミティブへ方策を偏らせるリスクがある。特権的シミュレーションメトリクスは学習を加速させるが、転移失敗を引き起こす。学習された行動は常にドメインランダム化条件およびハードウェア上の安全モニタで検証し、振動、作動器過熱、安全でない接触力を捕捉する。

36.4 ケーススタディ：ヒューマノイドのバランス制御

これまでの報酬設計と探索・活用のトレードオフに関する議論を踏まえ、本ケーススタディではこれらの原則を実用的なヒューマノイドのバランス維持タスクに適用する。問題を定式化し、安全性とエネルギーコストを含むコンパクトな報酬を導出し、安定した方策クラスと最適化手法を選択し、Isaac Sim ベースの環境に適した最小限の学習ループを提示する。

問題設定と MDP 定式化. タスクは外部からの押し力やセンサノイズを受けた後もヒューマノイドを直立姿勢に保つことである。これをエピソード型マルコフ決定過程 (MDP) としてモデル化する：

- 状態 s_t : 胴体姿勢 (ロール, ピッチ, ヨー), 角速度, 関節位置・速度, 重心 (CoM) オフセット, 足底接触フラグ, IMU 計測値.
- 行動 a_t : 足首, 膝, 股関節の連続関節トルク指令で, アクチュエータトルク制限を受ける.
- 遷移 $P(s_{t+1} | s_t, a_t)$: ノイズと遅延を含むシミュレーション上の物理ベースダイナミクス.
- 報酬 r_t : 直立性, 安定性, エネルギー効率, 転倒ペナルティを組み合わせた高密度報酬.

報酬の設計. 報酬は工学目的と学習しやすさを整合させる必要がある。物理的に意味のある項の重み付き和を用いる：

$$[H]r_t = w_u \exp(-\alpha \|\theta_t\|) - w_e \sum_j |\tau_{j,t} \dot{q}_{j,t}| - w_j \sum_j |\ddot{q}_{j,t}| - w_f \mathbf{1}\{\text{fallen}_t\}, \quad (307)$$

ここで θ_t は垂直方向に対するピッチ・ロールベクトル, $\tau_{j,t}$ は関節トルク, $\dot{q}_{j,t}$ は関節速度, $\mathbf{1}\{\text{fallen}_t\}$ は転倒フラグである。直立維持を優先しエネルギーとジャークをペナルティするよう w と α を選ぶ。姿勢に対する指数関数は逸出を小さくし勾配を飽和させない。

方策クラスとアルゴリズム選択. ガウス平均ネットワークと状態依存共分散でパラメータ化された確率的連続方策 $\pi_\phi(a | s)$ を用いる。推奨アルゴリズム：

- Proximal Policy Optimization (PPO)：頑健で単純、コンタクトを伴うダイナミクスに対するオンポリシー探索に優れる。
- Soft Actor-Critic (SAC)：オフポリシー、サンプル効率が良く、シミュレーション予算が制約される場合に有用。

バランスングでは、クリッピング目的関数を持つ PPO がコンタクトタスクで安定収束を示すことが多い。ハードウェアへの継続的な微調整を行う場合は SAC も検討できる。

探索戦略と安全制約. 行動ノイズと制約付き行動を組み合わせ危険なトルクを回避する：

- 状態依存適応ノイズ (パラメータ空間ノイズ) を用い、エピソードを通じて一貫した探索的行動を促す。
- トルククリッピングと、トルクまたは CoM が安全閾値を超えた場合に高い負の報酬とエピソードリセットをトリガーする安全監視を実装する。

これにより、危険だが報酬の高い攻略法を方策が発見することを防ぐ。

実装上の注意点. ドメインランダムマイゼーションと段階的カリキュラムを用いる：

1. 質量分布, 足底摩擦, センサノイズを現実的な範囲でランダム化する。
2. 小さな押し力から始め、徐々に外乱の大きさを増やす。

3. 外乱のないエピソードを定期的に挿入し、待機動作を安定化させる。

これにより頑健性が加速し、狭いシミュレーション領域への過学習が軽減される。

Isaac Sim 用のカスタム gym ライクラッパーを使った最小学習ループ例。リストは報酬ラップと stable-baselines3 による PPO 学習を示す。コメントは簡潔で実用的である。

コードサンプル 125 ヒューマノイドバランス用最小 PPO 学習ループ (例示)

```
import os
from typing import Any, Dict, Tuple

import gym
import numpy as np
from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import CheckpointCallback, EvalCallback
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.vec_env import DummyVecEnv, VecNormalize

class IsaacHumanoidRewardWrapper(gym.Wrapper):
    """Isaac環境の報酬を上書きし、直立・省エネ・滑らかさを考慮"""

    def __init__(self, env: gym.Env):
        super().__init__(env)
        self._prev_action: np.ndarray = np.zeros(env.action_space.shape)

    def reset(self, **kwargs) -> Dict[str, np.ndarray]:
        obs = self.env.reset(**kwargs)
        self._prev_action = np.zeros(self.env.action_space.shape)
        return obs

    def step(self, action: np.ndarray) -> Tuple[Dict[str, np.ndarray], float, bool, Dict]:
        obs, _, done, info = self.env.step(action)

        # 直立性：ロール・ピッチのノルムをペナルティ
        theta = obs["torso_angles"][:2]
        upright = 5.0 * np.exp(-2.0 * np.linalg.norm(theta))

        # エネルギー効率：トルク×速度
        energy = -0.01 * np.sum(np.abs(action * obs["joint_vel"]))
```

```

        # ジャーク：行動変化を滑らかさの代理
        jerk = -0.005 * np.sum(np.abs(action - self._prev_action))

        # 転倒
        fall = -50.0 if obs.get("fallen", False) else 0.0

        reward = float(upright + energy + jerk + fall)

        self._prev_action = action.copy()
        return obs, reward, done, info

def make_env(rank: int = 0) -> gym.Env:
    """環境生成：ラップ+監視+ベクトル化用"""
    def _init() -> gym.Env:
        env = gym.make("IsaacHumanoidBalance-v0")
        env = IsaacHumanoidRewardWrapper(env)
        env = Monitor(env, filename=None) # 統計記録
        return env

    return _init

if __name__ == "__main__":
    # 並列環境（ここでは1つ）
    n_envs = 1
    vec_env = DummyVecEnv([make_env(i) for i in range(n_envs)])
    vec_env = VecNormalize(vec_env, norm_obs=True, norm_reward=True)

    # 保存先
    log_dir = "./runs/ppo_balance"
    os.makedirs(log_dir, exist_ok=True)

    # コールバック：定期的に評価&チェックポイント保存
    eval_env = DummyVecEnv([make_env(999)])
    eval_env = VecNormalize(eval_env, training=False, norm_obs=True, norm_reward=False)
    eval_callback = EvalCallback(
        eval_env,
        best_model_save_path=log_dir,
        log_path=log_dir,

```

```

        eval_freq=50_000 // n_envs,
        deterministic=True,
        render=False,
    )
    ckpt_callback = CheckpointCallback(
        save_freq=200_000 // n_envs,
        save_path=log_dir,
        name_prefix="ppo_humanoid",
    )

    # 学習
    model = PPO(
        "MlpPolicy",
        vec_env,
        verbose=1,
        tensorboard_log=log_dir,
        n_steps=2048,
        batch_size=64,
        n_epochs=10,
        learning_rate=3e-4,
        clip_range=0.2,
        gamma=0.99,
        gae_lambda=0.95,
        max_grad_norm=0.5,
        ent_coef=0.0,
        vf_coef=0.5,
    )
    model.learn(total_timesteps=5_000_000, callback=[eval_callback, ckpt_callback])

    # 保存
    model.save(os.path.join(log_dir, "final_model"))
    vec_env.save(os.path.join(log_dir, "vec_normalize.pkl"))

```

評価と転送に関する考察. 以下の条件下で方策を検証する:

- 未経験の質量・慣性摂動.
- ハードウェアに合致する遅延とセンサドロップパターン.
- 異なる床面摩擦とコンプライアンス.

実機導入前にシステム同定を行い, シミュレーションダイナミクスをハードウェアに合わせる.
工学上の含意, トレードオフ, リスク.

- 報酬設計のトレードオフ：転倒ペナルティを強くすると機敏性を犠牲にした保守的な方策が得られる。
- アルゴリズムのトレードオフ：PPO はチューニングが単純だが SAC よりサンプルが必要。
- 安全性リスク：シミュレータの不正確さを悪用し unsafe なトルクを出力する場合がある；ランタイム安全監視を必ず実装する。
- 運用上のリスク：センサドリフトやアクチュエータ劣化によりバランス余裕が減少；メンテナンスとオンライン適応を計画する。

37 シミュレーションにおけるロボットの訓練

37.1 Isaac Sim を RL 環境に使用する

これまでに議論した RL の基礎と報酬設計の原則を踏まえ、本小節では Isaac Sim 環境を人間型コントローラの訓練用に構築する方法を示す。重点は学習安定性と sim-to-real 転移に影響を与える実践的なエンジニアリング選択にある。

問題：高 DoF 人間型をシミュレーション内でバランス、歩行、操縦タスクを実行できるよう訓練しながら、動力学の忠実度を保ち効率的な並列訓練を可能にする。主な制約には現実的な接触動力学、アクチュエータモデリング、センサ遅延、GPU 加速スケーリングが含まれる。成功には、状態・行動・環境セマンティクスを学習アルゴリズムの要求とハードウェア展開の両方に合わせて構造化することが必要である。

技術分析

- 環境状態と動力学：人間型を構成 q 、速度 \dot{q} 、制御入力 a を持つ関節剛体システムとしてモデル化する。RL エージェントが使用する離散時間動力学は

$$[H]x_{t+1} = f_{\Delta}(x_t, a_t) + w_t, \quad (308)$$

と表され、ここで $x = [q, \dot{q}]$ 、 f_{Δ} は時間刻み Δt におけるシミュレータの離散積分器、 w_t はモデル化されていない擾乱とセンサ雑音をモデル化する。アクチュエータレベル制御では PD または逆動力学マッピングを含める：

$$[H]\tau = K_p(q_{\text{des}} - q) + K_d(\dot{q}_{\text{des}} - \dot{q}), \quad (309)$$

およびロボット動力学

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J(q)^T F_{\text{contact}}. \quad (310)$$

- 物理忠実度 vs スループット：高忠実度（小さな積分刻み、多くのサブステップ、豊富な接触モデル）はより現実的な f_{Δ} を生むが秒あたりサンプル数を減らす。時間刻み Δt とサブステップ数を選び、安定性と GPU バッチサイズをバランスさせる。典型的な選択： $\Delta t \in [0.01, 0.02]$ s、内部サブステップ 4-8 が人間型接触安定性に対して有効。
- 観測と行動：観測には局所ベース姿勢、関節位置・速度、IMU 読み、接触フラグ、オプションでレンダリング画像または深度マップを含めるべきである。行動は次のものになり得る：

1. 低レベルトルク、
2. PD 内ループの位置設定値、
3. 高レベルターゲット（歩行位置、COM 設定値）。

シミュレートされた PD 内ループによる位置設定値を使用するとサンプル効率が向上し、一般的なハードウェア制御スタックと一致する。

Isaac Sim における実装設計図

1. ステージとアセット設定：
 - 正確な慣性特性と関節制限を持つ人間型 USD をインポート。
 - ソルバ設定（PhysX または PhysX PBD ハイブリッド）、摩擦係数、接触マージンを設定。
2. ベクトル化環境とセンサ：
 - Isaac Sim の Python API を使用し 1 つの GPU 上に多数の並列シーンを作成。
 - レプリカごとにレイキャストセンサ、力センサ、カメラセンサをアタッチ。
3. 環境ラッパ：
 - Gym 風インタフェースを公開：reset()、step(action)、render()、close()。
 - 再現性のための決定的ランダムイゼーションシードを実装。
4. ドメインランダムイゼーションと雑音：
 - 質量分布、関節ダンピング、摩擦、アクチュエータ遅延をランダムイズ。
 - 訓練中にガウシアンモデルに従うセンサ雑音を注入し頑健性を向上。
5. 訓練ループ考慮事項：
 - トレーナに応じて非同期または同期ステップングを選択。
 - リアルタイムロギング使用時に再現可能物理を維持するためリアルタイム係数を同期。

最小 Isaac Sim 環境スケルトン（Python）

コードサンプル 126 人間型のための最小 Isaac Sim RL 環境。

```
import numpy as np
from typing import Tuple, Dict, Any
import omni.isaac.core.utils.prims as prim_utils
from omni.isaac.core.world import World
from omni.isaac.core.articulations import ArticulationView
from omni.isaac.core.utils.nucleus import get_assets_root_path
from omni.isaac.core.utils.stage import add_reference_to_stage

class HumanoidEnv:
    """Isaac-Sim上で人型アシムートをPD制御するRL環境."""

    def __init__(
        self,
        usd_path: str,
```

```

        dt: float = 1 / 60,
        device: str = "cuda:0",
    ) -> None:
        self._world = World(stage_units_in_meters=1.0, physics_dt=dt, rendering_dt=dt)
        self._device = device
        self.dt = dt

    # USDをステージに追加
    add_reference_to_stage(usd_path, "/World/Humanoid")
    # ArticulationViewで並列シミュレーション対応
    self.humanoid = ArticulationView(
        prim_paths_expr="/World/Humanoid.*", name="humanoid_view"
    )
    self._world.scene.add(self.humanoid)

    # 関節数の取得
    self.num_dof = self.humanoid.num_dof
    assert self.num_dof > 0, "有効な関節が見つかりません"

    # PDゲイン
    self.Kp = np.full(self.num_dof, 100.0, dtype=np.float32)
    self.Kd = np.full(self.num_dof, 2.0, dtype=np.float32)

    # 初期姿勢
    self.default_q = np.zeros(self.num_dof, dtype=np.float32)

    # シミュレーション待機
    self._world.reset()

def reset(self) -> np.ndarray:
    """環境を初期状態に戻す."""
    self._world.reset()

    # ルート位置・姿勢を初期化
    root_pos = np.array([0.0, 0.0, 0.9], dtype=np.float32)
    root_quat = np.array([1.0, 0.0, 0.0, 0.0], dtype=np.float32)
    self.humanoid.set_world_poses(root_pos.reshape(1, 3), root_quat.reshape(1, 4))

    # 関節位置・速度を初期化
    self.humanoid.set_joint_positions(self.default_q)

```

```

self.humanoid.set_joint_velocities(np.zeros(self.num_dof, dtype=np.float32))

# 1ステップ進めて物理を安定
self._world.step(render=False)
return self._get_obs()

def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, Dict[str, Any]]:
    """1ステップ進める."""
    action = np.clip(action, -np.pi, np.pi).astype(np.float32)

    # 現在状態取得
    q = self.humanoid.get_joint_positions()
    qd = self.humanoid.get_joint_velocities()

    # PDトルク計算
    tau = self.Kp * (action - q) - self.Kd * qd
    self.humanoid.set_joint_efforts(tau)

    # シミュレーション進行
    self._world.step(render=True)

    obs = self._get_obs()
    reward = self._compute_reward(obs, action)
    done = self._check_termination(obs)
    return obs, reward, done, {}

def _get_obs(self) -> np.ndarray:
    """観測を返す."""
    q = self.humanoid.get_joint_positions()
    qd = self.humanoid.get_joint_velocities()
    return np.concatenate([q, qd]).astype(np.float32)

def _compute_reward(self, obs: np.ndarray, action: np.ndarray) -> float:
    """報酬計算."""
    # 起立報酬（腰の高さ）
    base_z = self.humanoid.get_world_poses()[0][0, 2]
    upright_reward = base_z

    # エネルギー消費ペナルティ
    energy_penalty = 1e-3 * np.sum(np.square(action))

```

```

return upright_reward - energy_penalty

def _check_termination(self, obs: np.ndarray) -> bool:
    """転倒判定."""
    base_z = self.humanoid.get_world_poses()[0][0, 2]
    return base_z < 0.3 # 腰が0.3 m未満で終了

```

実践的な訓練推奨事項

- 人間型のベクトル化レプリカを使用し効率的にバッチを収集。Isaac Sim は最新 GPU 上で大きなバッチサイズをサポート。
- 制御周波数をハードウェアに合わせる；実機が 100Hz で動作する場合、同様の行動更新レートで訓練するか制御遅延を明示的にモデル化。
- カリキュラム学習を使用：擾乱を減らし小さなタスクから始め、難易度とランダムマイゼーションを増加。
- 接触、根軌跡、作動履歴をログしオフライン解析に利用。

エンジニアリングへの影響、トレードオフ、リスク

- 忠実度 vs スループット：最終ポリシー頑健性のため忠実度を優先。混合忠実度を使用：粗いポリシーを素早く訓練し、高忠実度シミュレーションで洗練。
- アクチュエータモデリング：単純化されたトルクまたは PD モデルは脆い転移のリスクがある。アクチュエータ動力学と遅延を含め sim-to-real ギャップを削減。
- 接触モデリング：摩擦または接触幾何の小さな誤差がバランスコントローラを不安定化。接触パラメータをハードウェア実験で検証。
- 運用上のリスク：狭いドメインランダムマイゼーション分布に過適合すると展開時に故障モードが生じる。環境とセンサ雑音プロファイルの多様性を維持。

これらの実践により、Isaac Sim 内で再現可能でスケーラブルな RL 環境が生成され、ハードウェアインザループ試験での慎重な検証と組み合わせることで、より安全で転移可能な人間型ポリシーが得られる。

37.2 学習済み挙動のハードウェアへの転送

前の小節では Isaac Sim における高忠実度 RL 環境の構築と、sim-to-real ギャップを縮小するシミュレーション手法について述べた。これらの実践は、仮想ヒューマノイドレプリカから物理ハードウェアへポリシーを安全かつ確実に移行するための構造化されたパイプラインの出発点となる。

問題定義。転送には、ダイナミクス、センシング、タイミング、コンタクトの違いを橋渡しする必要がある。ヒューマノイドロボットは自由度が高く、足部がアンダーアクチュエートされ、歩行中に繰り返し衝撃イベントが発生するため、これらの差異が増幅される。エンジニアリングの目標は、(1) ハードウェア制限を尊重し、(2) センサおよびモデルの不確実性に耐性があり、(3) 予期せぬ条件下でグレースフルに失敗するポリシーを提供することである。

技術解析。ヒューマノイドの剛体ダイナミクスは連続時間で

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J_c^\top(q)f_c, \quad (311)$$

に従う。ここで q は関節位置、 τ はアクチュエータトルク、 J_c はコンタクトヤコビアン、 f_c はコンタクト力である。シミュレータは M, C, g, J_c の推定値を提供するが、物理ロボットにはパラメータ mismatches とモデル化されていない柔軟性が存在する。転送誤差の 2 クラスが支配的である：

- パラメトリック mismatches：質量、慣性、摩擦、モータ定数がモデル値と異なる；
- モデル化されていない現象：関節コンプライアンス、センサバイアス、通信遅延、コンタクトマイクロダイナミクス。

頑健な学習目的は、モデルパラメータ θ を確率変数として扱い、期待性能損失を最小化する。 π_ϕ を ϕ でパラメータ化されたポリシーとし、 $R(\pi_\phi; \theta)$ をパラメータ θ 下でのリターンとする。頑健最適化は

$$\max_{\phi} \mathbb{E}_{\theta \sim p(\theta)} [R(\pi_\phi; \theta)], \quad (312)$$

となる。ここで $p(\theta)$ は質量、ダンピング、コンタクト摩擦、センサノイズ、遅延の不確実性範囲を捉えたドメインランダム化分布である。

実装パイプライン。以下の具体的な手順がエンジニアリングワークフローを形成する。

1. システム同定（システム ID）。

- 安全な低ゲインコントローラを使用してハードウェアで制御軌道を収集する。
- アクチュエータダイナミクス（モータトルク定数、時定数）と関節摩擦のパラメトリックモデルを当てはめる。
- 簡略化されたトルクマップを線形最小二乗で解く： $\tau \approx K_p(q_{\text{ref}} - q) + K_d(\dot{q}_{\text{ref}} - \dot{q}) + \tau_{\text{ff}}$ 。

2. シミュレーション拡張。

- 同定されたアクチュエータラグ、トルク制限、センサ遅延、現実的な IMU/ビジョンノイズを Isaac Sim に注入する。
- 同定されたパラメータ不確実性と妥当なモデル化されていない範囲にわたってドメインランダム化を適用する。
- 移動タスクのために地形コンプライアンスとコンタクト摩擦をランダム化する。

3. セーフティラッパーとアクションフィルタリング。

- ハードウェア上でポリシーをセーフティエンベロープ内で実行する。アクションクリッピング、レート制限、人間監視のデッドマンを強制する。
- 状態推定値が閾値を超えて逸脱した場合に引き継ぐフォールバックコントローラ、例えば低ゲイン PD スタビライザを実装する。

4. ポリシー条件付けとハイブリッド制御。

- 推定質量や遅延など、測定可能なシステムパラメータが利用可能な場合は、学習済みポリシーをこれらに条件付ける。
- 残差強化学習を使用する：モデルベースコントローラ π_{base} と学習済み残差 π_{res} を組み合わせる：

$$\tau = \pi_{\text{base}}(s) + \pi_{\text{res}}(s; \phi). \quad (313)$$

これはベースライン安定性を保ちながら学習による改善を可能にする。

5. 段階的展開とオンライン適応。

- テザーされた低エネルギーテストから始め、段階的に複雑性を増やす。
- ベイジアンオンライン推定による θ の推定や、セーフティ制約下での少数ショット更新による高速ファインチューニングなど、オンライン適応手法を使用する。

実践的マッピングと制約。正規化されたポリシー出力 $a \in [-1, 1]^n$ を、単調マッピングと変化率フィルタリングを用いてハードウェアトルクにマッピングする：

$$\tau_t = \text{clip}(\tau_{t-1} + \Delta_{\max} \cdot a, \tau_{\min}, \tau_{\max}), \quad (314)$$

ここで Δ_{\max} はアクチュエータスループレートから選択する。コンプライアントトランスミッションにおけるモデル化されていない共振を励起しないよう、ローパスフィルタリングを実装する。

コード例。以下の ROS 互換 Python ブリッジは、PyTorch ポリシーの出力を安全なハードウェアコマンドにマッピングする例を示す。基本的なセーフティチェックとウォッチドッグタイマを含む。

コードサンプル 127 Policy-to-hardware bridge with safety checks

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
from std_msgs.msg import Float64MultiArray
import torch
import numpy as np
from typing import List, Optional

class PolicyRunner(Node):
    def __init__(self) -> None:
        super().__init__('policy_runner')

        # QoS設定：リアルタイム制御に適したプロファイル
        ctrl_qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        self._pub = self.create_publisher(
            Float64MultiArray, '/cmd_torque', ctrl_qos
        )
```

```

self._sub = self.create_subscription(
    JointState, '/joint_states', self.joint_cb, ctrl_qos
)

# ポリシ読み込み
self.policy: torch.jit.ScriptModule = torch.jit.load(
    'policy.pt', map_location='cpu'
)
self.policy.eval()

# パラメータ
self.declare_parameter('num_joints', 7)
self.declare_parameter('watchdog_timeout', 0.5)
self.declare_parameter('torque_limits', [-30.0, 30.0])
self.declare_parameter('delta_max', 5.0)

self.num_joints: int = self.get_parameter('num_joints').value
self.watchdog_timeout: float = self.get_parameter('watchdog_timeout').value
self.tau_min: float = self.get_parameter('torque_limits').value[0]
self.tau_max: float = self.get_parameter('torque_limits').value[1]
self.delta_max: float = self.get_parameter('delta_max').value

self.prev_tau: np.ndarray = np.zeros(self.num_joints)
self.last_time: float = self.get_clock().now().nanoseconds * 1e-9

# ウォッチドッグタイマー
self._watchdog = self.create_timer(0.02, self.watchdog_cb) # 50 Hz

def joint_cb(self, msg: JointState) -> None:
    self.last_time = self.get_clock().now().nanoseconds * 1e-9

# 状態テンソル構築
q = np.array(msg.position, dtype=np.float32)
dq = np.array(msg.velocity, dtype=np.float32)
state = torch.from_numpy(np.concatenate([q, dq])).unsqueeze(0)

with torch.no_grad():
    a = self.policy(state).squeeze(0).numpy() # [-1, 1]

# トルク指令算出 & クリップ

```

```

        delta = np.clip(a * self.delta_max, -self.delta_max, self.delta_max)
        tau = np.clip(self.prev_tau + delta, self.tau_min, self.tau_max)
        self.prev_tau = tau

    out = Float64MultiArray()
    out.data = tau.tolist()
    self._pub.publish(out)

def watchdog_cb(self) -> None:
    if self.get_clock().now().nanoseconds * 1e-9 - self.last_time > self.watchdog_timeout:
        # 安全停止
        stop_msg = Float64MultiArray()
        stop_msg.data = [0.0] * self.num_joints
        self._pub.publish(stop_msg)

def main(args: Optional[List[str]] = None) -> None:
    rclpy.init(args=args)
    node = PolicyRunner()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

エンジニアリングへの影響とトレードオフ。ドメインランダム化は頑健性を高めるが、ハードウェア能力を十分に活用しない保守的な挙動を生む可能性がある。残差 RL は安全性を保つが、学習による改善の大きさを制限する。重度のフィルタリングとアクションクリッピングは振動を減らすが、応答を遅くする可能性がある。オンライン適応は性能を改善するが、制約なしでは不安定化のリスクを伴う複雑性を追加する。運用上のリスクには、意図しない高衝撃コンタクト、アクチュエータの過熱、センサバイアスの蓄積が含まれる。学習済みヒューマノイド挙動を展開する際は、段階的テスト、ハードウェアインザループシステム ID、頑健なウォッチドッグを優先する。

37.3 シミュレーションへの過学習を回避する

これらの技術を基に、本小節ではポリシーがシミュレータの人工産物を悪用し、実際のヒューマノイド展開で失敗することを防ぐためのエンジニアリング手法に焦点を当てる。

問題定義：シミュレートされた報酬のみを最大化するように訓練されたポリシーは、物理的不正確

さを悪用する脆い戦略を学習することが多い。このような過学習は、非現実的な接触シーケンス、完全なセンサへの依存、実際のノイズ下で破綻するタイミングに敏感な行動として現れる。ヒューマノイドの場合、摩擦が異なる際の不安定なバランス、アクチュエータ動特性の変化によるぎくしゃくした関節トルク、合成テクスチャによる視覚駆動の誤認識が失敗に含まれる。

技術的分析。実践的な影響の順に並べた過学習の源：

- 物理の誤指定：質量分布、接触剛性、地面コンプライアンスの不正確さ。
- センサモデルの不一致：過度にクリーンな視覚、理想的 IMU 出力、遅延の欠如。
- 制御・駆動の不一致：理想化されたトルク制御対実際のモータ電流制御と飽和。
- 報酬の悪用：タスクを達成せずに報酬を最大化する非物理的な近道。
- 視覚／ドメイン文脈におけるデータセットバイアス：展開を代表しないテクスチャ、物体配置、照明。

頑健な訓練目的を形式化する。 ϕ をシミュレータパラメータ（動特性、センサ、視覚）とする。分布 $p(\phi)$ 下での期待ポリシー報酬を定義する：

$$[H] \max_{\pi_\theta} \mathbb{E}_{\phi \sim p(\phi)} [J(\pi_\theta; \phi)], \quad (315)$$

ここで J はエピソード報酬である。リスク回避または最悪ケース定式化は min-max 目的を用いる：

$$[H] \max_{\pi_\theta} \min_{\phi \in \Phi} J(\pi_\theta; \phi), \quad (316)$$

Φ は妥当なパラメータの有界集合である。式 (1)–(2) はエンジニアリング選択を導く：期待値を近似するため ϕ をサンプリングするか、敵対的摂動下で性能が良いポリシーを最適化する。

Isaac Sim におけるヒューマノイド RL のための実装レシピ。目的は完璧な現実性ではなく運用上の頑健性である。以下のエンジニアリングパイプラインを用いる：

1. システム同定ベースライン
 - スクリプト化された動作を実行するヒューマノイドから実軌道を収集する。
 - 主要パラメータをフィット：リンク質量、重心オフセット、関節摩擦、アクチュエータ遅延統計。
2. パラメータ分布 $p(\phi)$ の設計
 - 同定値を中心に分布を配置する。
 - 製造ばらつきと摩耗をカバーするよう範囲を広げる。
 - 必要に応じて質量と慣性の相関摂動を含める。
3. 訓練中の動特性・センサのランダムマイゼーション
 - エピソードまたはロールアウトセグメントごとに物理特性をランダム化する。
 - 測定分布からサンプルしたセンサノイズと遅延を注入する。
4. 視覚ドメインのランダムマイゼーション
 - テクスチャ、照明、カメラ内部パラメータ、レンズ歪みをランダム化する。
 - 手続き的に生成された背景と障害物オブジェクトを用いる。
5. 正則化と検証
 - ポリシー Entropy または L_2 正則化を追加して過度の特化を防ぐ。
 - 検証シナリオを取っておき、定期的の実機試験を実行する。

6. 敵対的およびアンサンブル手法

- 状態入力への敵対的摂動、または ϕ を変更する敵対者とともに訓練する。
- 認識論的不確実性を推定するための動特性モデルのアンサンブルを維持する。

ヒューマノイド歩行のための具体的なランダム化セッション選択：

- 質量：四肢ごとに ± 10 – 20 。
- 関節ダンピングおよびクーロン摩擦： ± 30 – 50 。
- 地面摩擦係数： $[0.4, 1.2]$ で一様。
- 接触剛性／コンプライアンス：ゴム、コンクリート、カーペットをエミュレートするよう変化。
- アクション遅延： $[0, 3]$ 制御ティックで整数ステップ遅延をサンプル。
- IMU バイアスとノイズ：ハードウェアに一致するスペクトル特性を持つガウスバイアスドリフト。

実践的なコード例。スニペットは動特性パラメータをサンプルし、エピソードごとにシミュレータに適用する訓練ループを示す。プレースホルダを統合における Isaac Sim API 呼び出し（環境パラメータセッターまたは PhysX シーンオーバーライドなど）に置き換える。

```
import numpy as np
import torch
from typing import Dict, Any
import rclpy
from rclpy.node import Node
from omni.isaac.core.utils.prims import define_prim
from omni.isaac.core.simulation_context import SimulationContext
from stable_baselines3 import SAC
from stable_baselines3.common.buffer import ReplayBuffer
import yaml
import time
import logging
```

```
class DomainRandomizer:
    """Isaac Simの物理パラメータをランダムイズ"""
    def __init__(self, rng: np.random.Generator):
        self.rng = rng
        self.param_ranges = {
            "mass_scale": (0.9, 1.2),
            "friction": (0.4, 1.2),
            "joint_damping": (0.7, 1.3),
            "action_delay": (0, 3),
            "imu_noise_std": (0.01, 0.08),
```

```

    }

def sample(self) -> Dict[str, Any]:
    return {k: self.rng.uniform(*v) if k != "action_delay" else self.rng.integers(
        for k, v in self.param_ranges.items())}

```

```

class IsaacEnvWrapper:

```

```

    """Isaac Sim環境のラッパー"""

```

```

    def __init__(self, stage_path: str, robot_prim: str):
        self.sim = SimulationContext()
        define_prim(stage_path, "Xform")
        self.robot_prim = robot_prim
        self.dr = DomainRandomizer(np.random.default_rng())

```

```

    def apply_physics_params(self, params: Dict[str, Any]):

```

```

        """USDプロパティに物理パラメータを反映"""

```

```

        from pxr import UsdPhysics, PhysxSchema

```

```

        stage = self.sim.stage

```

```

        for prim in stage.Traverse():

```

```

            if prim.GetTypeName() == "PhysicsRigidBodyAPI":

```

```

                mass_api = UsdPhysics.MassAPI(prim)

```

```

                if mass_api:

```

```

                    base_mass = mass_api.GetMassAttr().Get()

```

```

                    mass_api.GetMassAttr().Set(base_mass * params["mass_scale"])

```

```

            if prim.GetTypeName() == "PhysicsMaterial":

```

```

                material = UsdPhysics.MaterialAPI(prim)

```

```

                material.GetStaticFrictionAttr().Set(params["friction"])

```

```

                material.GetDynamicFrictionAttr().Set(params["friction"])

```

```

    def reset(self):

```

```

        self.sim.reset()

```

```

        return self._get_obs()

```

```

    def step(self, action):

```

```

        self.sim.step()

```

```

        obs = self._get_obs()

```

```

        reward = self._compute_reward(obs, action)

```

```

        done = self._check_done(obs)

```

```

        return obs, reward, done, {}

```

```

def _get_obs(self):
    """実装に応じて観測を取得"""
    return np.zeros(42) # dummy

def _compute_reward(self, obs, action):
    return -np.sum(action**2)

def _check_done(self, obs):
    return False

class DelayedActionWrapper:
    """アクション遅延をエミュレート"""
    def __init__(self, delay_steps: int):
        self.delay_steps = delay_steps
        self.queue = []

    def __call__(self, action: np.ndarray) -> np.ndarray:
        self.queue.append(action.copy())
        if len(self.queue) > self.delay_steps:
            return self.queue.pop(0)
        return np.zeros_like(action)

def main():
    rclpy.init()
    node = Node("isaac_rl_train")

    # 設定読み込み
    with open("config/train_config.yaml", "r") as f:
        cfg = yaml.safe_load(f)

    env = IsaacEnvWrapper("/World/robot", "/World/robot/torso")
    replay_buffer = ReplayBuffer(
        cfg["buffer_size"],
        env.observation_space,
        env.action_space,
        device="cuda",
        n_envs=1
    )

```

```

policy = SAC("MlpPolicy", env, verbose=1, device="cuda")

for episode in range(cfg["num_episodes"]):
    params = env.dr.sample()
    env.apply_physics_params(params)

    obs = env.reset()
    delay_wrapper = DelayedActionWrapper(int(params["action_delay"]))

    done = False
    episode_reward = 0.0

    while not done:
        action, _ = policy.predict(obs, deterministic=False)
        action = delay_wrapper(action)

        obs, reward, done, info = env.step(action)
        replay_buffer.add(obs, action, reward, done, info)
        episode_reward += reward

        if replay_buffer.size() > cfg["learning_starts"]:
            policy.train(gradient_steps=cfg["gradient_steps"])

    node.get_logger().info(f"Episode_{episode}: reward={episode_reward:.2f}")

    policy.save("isaac_sac_policy")
    rclpy.shutdown()

if __name__ == "__main__":
    main()

```

検証と転移チェック。極端な摩擦、部分的センサドロップアウト、非一様質量シフトなどのシミュレーションホールドアウトを維持する。定期的な実機評価をスケジュールして失敗モードを収集する。これらの実機ロールアウトを用いて $p(\phi)$ を改良し、同定ループを閉じる。

エンジニアリングにおける意味とトレードオフ：

- ランダマイゼーションを広げると頑健性は増すが、訓練収束が遅く、ピークシミュレーション性能が劣化する可能性がある。
- 保守的なパラメータ境界は未経験の失敗の確率を下げるが、実際のばらつきに過不足なく適合することがある。
- アンサンブルおよび敵対的アプローチは汎化を改善するが、より多くの計算資源を必要とし、安

定性のための慎重な調整が必要である。

- 過度のランダム化は有用な勾配を隠蔽し、訓練エポックにわたってランダム性を増やすカリキュラムスケジュールを必要とする。

運用上のリスク：

- アクチュエータ飽和または熱効果をモデル化できないと、ハードウェア上で安全でない指令を生成する可能性がある。
- 視覚ギャップの過小評価は、雑然とした環境での知覚トリガ型の失速を引き起こす。
- 実機での不十分な検証は、壊滅的なバランス失敗の遅い発見というリスクがある。

反復サイクルを採用：同定、ランダム化、検証、改良。不確実性を定量化し、パラメータ範囲を文書化し、実機試験中は安全モニタを含める。これらの手法は、ヒューマノイドロボットのシミュレーション過学習リスクを実質的に減らし、運用信頼性を向上させる。

38 強化学習の応用

38.1 ヒューマノイドの歩行と走行

前述のシミュレーション訓練と sim-to-real 移行戦略を基に、本小節では強化学習を用いて頑健なヒューマノイド歩行・走行歩容を生成することに焦点を当てる。実用的な設計選択、報酬設計、訓練カリキュラム、およびハードウェアへ確実に移行する実装パターンを強調する。

問題定義. 多自由度ヒューマノイドに対してセンサ観測からアクチュエータ指令へ写像する制御方針を設計せよ。その方針は

- 指令水平速度および進行方向を追従し、
- 外乱と接触不確実性下でバランスを維持し、
- エネルギー効率が良くアクチュエータ制限を尊重し、
- シミュレーションからオンボードセンサおよび遅延アクチュエータへ一般化しなければならない。

鍵となる制約と困難の源は単脚支持相での非駆動、非スムーズ接触ダイナミクス、高次元状態・行動空間、限られたオンボードセンシングである。強化学習 (RL) はこれをマルコフ決定過程 (MDP) として扱う。定義：

- 状態 s ：関節角度、関節速度、IMU 姿勢および角速度、足部接触フラグ、推定 CoM および所望速度、
- 行動 a ：関節トルク指令または局所 PD コントローラへ渡す関節位置目標のいずれか、
- 報酬 $r(s, a)$ ：歩行・走行挙動を形成するための設計されたスカラー報酬。

報酬構造と安定性指標. 実用的な報酬関数は運動学的、動力学的、安全性の項を組み合わせる。一般的な複合報酬は：

$$[H]R(s, a) = w_v R_v + w_p R_p + w_e R_e + w_c R_c - w_{\text{pen}} P, \quad (317)$$

ただし

- $R_v = \exp(-\alpha_v \|v_{\text{com}} - v_{\text{target}}\|^2)$ は速度追従を促す,
- $R_p = \exp(-\alpha_p \|q - q_{\text{nominal}}\|^2)$ は公称姿勢を保持する,
- $R_e = \exp(-\alpha_e E)$ は消費エネルギー $E = \sum \tau_i \dot{q}_i$ をペナルティする,
- R_c は所望の足部クリアランスと対称的な足置きを報酬する,
- P は関節制限, 大きな足部滑り, 胴体転倒に対する安全性ペナルティの和である.

明示的な安定性指標を含める. キャプチャポイント計画には線形倒立振子の固有振動数 $\omega = \sqrt{g/h}$ を用いる. キャプチャポイント x_{cp} は目標足置き位置を提供する:

$$[H]x_{\text{cp}} = x + \frac{v}{\omega}, \quad (318)$$

ただし x は水平 CoM 位置, v はその速度である. x_{cp} へ歩行を誘導する足置き項を組み込む.

アルゴリズム選択とカリキュラム. 標本効率と安定性を両立した RL アルゴリズムを選択する. 安定更新を伴うオンポリシー学習には PPO (Proximal Policy Optimization) を用いる. リプレイを伴う標本効率には SAC (Soft Actor-Critic) を用いる. 典型的なエンジニアリング手順:

1. 高忠実度物理シミュレーション (Isaac Sim) で訓練し, 接触およびアクチュエータモデルを可能にする.
2. 低目標速度および制約付き外乱から開始する.
3. 段階的に v_{target} と地形難易度を増加させる (カリキュラム学習).
4. 慣性, 摩擦, センサ遅延, レイテンシにドメインランダムマイゼーションを適用する.

階層分解は学習負荷を軽減する. 所望ステップ位置とタイミングを出力する高レベル歩容プランナと, 局所状態を関節レベル指令へ写像する低レベル安定化方針を実装する. この分離は学習を加速し解釈性を向上させる.

状態・行動エンジニアリング. 良好な観測設計は部分観測問題を軽減する:

- 接触タイミングを方針が推論できるよう遅延関節履歴またはフィルタ速度を含める,
- 入力をロボット固有のスケール (質量, 脚長) で正規化し移行性を改善する,
- モーションキャプチャ歩行を模倣する際は参照位相信号を状態に付加する.

行動はハードウェア制限を尊重すべきだ. 二つの実用的選択:

- 行動 = PD 目標; 局所 PD は高周波トルク安定化を処理する,
- 行動 = 正規化トルクに安全クリッピング.

実装スケッチ. 以下のコードはパラメータ化速度指令を用いた PPO 訓練ループを概説する. 環境リセット, 報酬計算, 方針更新を実行する. 環境および方針を独自の Isaac Sim インタフェースおよびネットワークアーキテクチャへ置き換えること.

コードサンプル 129 速度条件付き歩容の最小 PPO 訓練ループ

```
import numpy as np
import torch
from typing import Tuple, Dict, Any
```

```

# 環境インターフェース（型ヒント付き）
class Env:
    def reset(self) -> np.ndarray: ...
    def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, Dict[str, Any]]: ...

# バッファ（Tensor化+GPU対応）
class RolloutBuffer:
    def __init__(self, device: torch.device):
        self.device = device
        self.clear()

    def store(self, obs: np.ndarray, act: np.ndarray, rew: float, done: bool):
        self.obs.append(torch.from_numpy(obs).float())
        self.actions.append(torch.from_numpy(act).float())
        self.rewards.append(rew)
        self.dones.append(done)

    def tensor(self) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]:
        return (torch.stack(self.obs).to(self.device),
                torch.stack(self.actions).to(self.device),
                torch.tensor(self.rewards, device=self.device),
                torch.tensor(self.dones, device=self.device))

    def clear(self):
        self.obs, self.actions, self.rewards, self.dones = [], [], [], []

# GAE計算（バッチ化）
@torch.no_grad()
def compute_gae(rewards: torch.Tensor,
                dones: torch.Tensor,
                values: torch.Tensor,
                gamma: float = 0.99,
                lam: float = 0.95) -> torch.Tensor:
    T = len(rewards)
    gae = 0
    adv = torch.zeros_like(rewards)
    for t in reversed(range(T)):
        delta = rewards[t] + gamma * values[t+1] * (1 - dones[t]) - values[t]
        gae = delta + gamma * lam * (1 - dones[t]) * gae
        adv[t] = gae

```

```

    return adv

# 本番コード
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
buffer = RolloutBuffer(device)

for episode in range(num_episodes):
    obs = env.reset()
    v_target = sample_velocity()
    policy.set_command(v_target)

    done = False
    while not done:
        action = policy.act(torch.from_numpy(obs).float().to(device))
        next_obs, r, done, info = env.step(action.cpu().numpy())
        buffer.store(obs, action.cpu().numpy(), r, done)
        obs = next_obs

    obs_t, act_t, rew_t, done_t = buffer.tensor()
    values = policy.critic(obs_t).squeeze()
    advantages = compute_gae(rew_t, done_t, torch.cat([values, values[-1:]]))
    policy.update(obs_t, act_t, advantages)
    buffer.clear()

```

sim-to-real 移行の実装上の考慮. sim-to-real ミスマッチを減らすため:

- 接触摩擦, リンク質量, センサノイズをランダム化する,
- アクチュエータ遅延と帯域制限をシミュレーションに追加する,
- 誤推定 CoM および状態推定器ドリフトに対して頑健な方針を訓練する,
- ハードウェアへ送信する前に行動に対して保守的安全フィルタを適用する.

評価指標とテスト. ハードウェア試験前に以下のバッテリーを用いる:

- 変動指令速度に対する速度追従誤差,
- 規定インパルス大きさの突きからの回復,
- 走行距離あたりのエネルギー,
- 足部滑り頻度および最大横偏差.

エンジニアリングへの影響, トレードオフ, リスク. RL は柔軟な歩容を生み出すが標本複雑性, シミュレーション忠実度要件, 評価中のハードウェア摩耗を増大させる. トレードオフ:

- 表現力の高い方針 vs. 検証可能性および安全性保証,
- 標本効率のための階層制御 vs. システム複雑性の追加.

運用上のリスクには予期せぬ接触ダイナミクス、ハードウェア故障を引き起こす高関節トルク、シミュレーションアーチファクトを悪用する方針が含まれる。これらは段階的テスト、保守的アクションクリッピング、学習済み方針とモデルベース安全モニタを組み合わせたハイブリッド制御アーキテクチャで軽減する。

38.2 物体操作タスク

動的歩行で学習されたバランスや全身協調などのスキルは、操作へ移行する際に有用な事前知識となる。物体操作では、より厳密な相互作用モデル、多モーダルセンシング、および歩行制御器を補完する接触を意識した方策が必要である。

ヒューマノイドロボットによる物体操作は、物体を掴み、移動し、組み立て、または使用するための接触を伴う相互作用を計画・実行するエンジニアリングタスクである。問題定義：観測 o_t を行動 a_t に写像し、実環境で信頼性が高く安全な把持および物体運動を生成する方策 π_θ （パラメータ θ でパラメータ化）を設計せよ。ここで観測は視覚、プロプライオセプション、触覚センシングを組み合わせる。行動は関節位置目標、関節トルク、または高レベルのエンドエフェクタ設定点となり得る。強化学習（RL）はこれを期待累積報酬の最大化として扱う：

$$[H]J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^T \gamma^t r_t \right], \quad (319)$$

ここで τ は軌道、 γ は割引率、 r_t はタスク成功、接触品質、安全性を符号化する。

技術的分析

- 観測設計：単眼／深度画像、関節角度、関節速度、触覚アレイを組み合わせる。視覚には知覚エンコーダを、プロプライオセプションには小さな MLP を用いて融合する。知覚または状態推定から得られる場合は相対物体姿勢を含める。
- 行動パラメータ化：低レベルトルク行動はコンプライアントな相互作用に対する精密な制御を提供するが、正確なダイナミクスを必要とする。位置またはインピーダンス目標は方策学習を単純化しサンプル複雑性を削減する。
- 接触モデリング：接触イベントはダイナミクスを不連続に変化させる。堅牢なソルバおよび摩擦モデルを備えた接触を意識したシミュレーションを用いる。観測に接触インジケータを組み込み、方策が滑りや衝撃に反応できるようにする。
- 報酬設計：報酬をタスク成分に分解する：
 1. r_{goal} ：物体姿勢近接度、
 2. r_{grasp} ：安定把持メトリクス（力閉鎖、レンチマージン）、
 3. r_{energy} ：アクチュエータ努力に対するペナルティ、
 4. r_{safety} ：関節リミットおよび衝突回避。

把持安定性の実用的な報酬項は、推定レンチマージン w_{margin} に比例する。調整された重みで線形に結合する。

制御統合 RL で使用される主要な方程式には、運動学および力の関係が含まれる。エンドエフェクタ空間速度は $v = J(q)\dot{q}$ に従い、ここで $J(q)$ はマニピュレータヤコビアンである。エンドエフェクタレンチ F から関節トルクへの写像は

$$[H]\tau = J(q)^\top F, \quad (320)$$

であり、これは指令レンチと関節空間制御器をブレンドするハイブリッドインピーダンス方策を情報提供する。

実装パターン

- カリキュラム学習：静止した胴体と近距離物体から始め、リーチング、複雑な把持、全身再配置を追加する。
- 階層 RL：サブゴール（把持姿勢、持ち上げ軌道）のための高レベルプランナーと、コンプライアントな実行のための低レベル制御器を用いる。
- ドメインランダムマイゼーションおよびシステム同定：Isaac Sim で物体質量、摩擦、センサノイズ、関節ダンピングをランダム化し、シムツールリアル転移を改善する。
- 安全性制約：リスクメトリクスを境界付けるために制約付き RL を適用する。以下として定式化する

$$[H] \max_{\theta} J(\theta) \quad \text{s.t.} \quad \mathbb{E}_{\tau \sim \pi_{\theta}} [C(\tau)] \leq C_{\max}, \quad (321)$$

ここで C は衝突、過剰な力、または不安定な圧力中心逸脱を測定する。

実用的なトレーニンググループ（Isaac Sim + RL フレームワーク）：

コードサンプル 130 Isaac Sim 観測と方策最適化器を統合した最小トレーニンググループ。

```
#!/usr/bin/env python3
import os
import random
from dataclasses import dataclass
from pathlib import Path
from typing import Dict, List, Optional, Tuple

import numpy as np
import torch
import torch.nn as nn
from torch.utils.tensorboard import SummaryWriter

# ROS 2
import rclpy
from rclpy.node import Node

# Isaac Sim
from omni.isaac.kit import SimulationApp
from omni.isaac.core.utils.extensions import enable_extension
from omni.isaac.core.simulation_context import SimulationContext
from omni.isaac.core.tasks import BaseTask
from omni.isaac.core.scenes.scene import Scene
from omni.isaac.core.prims import XFormPrimView
```

```

from omni.isaac.core.utils.prims import define_prim
from omni.isaac.core.utils.stage import open_stage
from omni.isaac.core.utils.nucleus import get_assets_root_path
from omni.isaac.core.utils.viewports import set_camera_view
from omni.isaac.sensor import TactileSensor
from omni.isaac.core.utils.types import ArticulationAction

# RL
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv, VecNormalize
from stable_baselines3.common.callbacks import EvalCallback, CheckpointCallback
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.utils import set_random_seed

# MPC (for safety filter)
from do_mpc.controller import MPC
from do_mpc.model import Model
from do_mpc.tools import structure

# 設定
@dataclass
class TrainConfig:
    exp_name: str = "isaac_ppo_tactile"
    num_episodes: int = 5000
    eval_interval: int = 50
    seed: int = 42
    device: str = "cuda" if torch.cuda.is_available() else "cpu"
    log_dir: str = "./runs"
    save_dir: str = "./checkpoints"
    isaac_usd_path: str = "/Isaac/Samples/Tasks/PickAndPlace/pick_and_place.usd"
    tactile_prim_path: str = "/World/robot/gripper/tactile"
    robot_prim_path: str = "/World/robot"
    camera_prim_path: str = "/World/camera"
    mpc_horizon: int = 10
    mpc_dt: float = 0.1

# Isaac Sim 環境
class IsaacSimEnv(Node):

```

```

def __init__(self, cfg: TrainConfig, headless: bool = True) -> None:
    Node.__init__(self, "isaac_env")
    self.cfg = cfg
    self.headless = headless
    self._init_sim()
    self._init_scene()
    self._init_sensors()
    self._init_spaces()

def _init_sim(self) -> None:
    # Isaac Sim起動
    self.simulation_app = SimulationApp({"headless": self.headless})
    enable_extension("omni.isaac.sensor")
    self.simulation_context = SimulationContext(
        physics_dt=1.0 / 60.0, rendering_dt=1.0 / 30.0, stage_units_in_meters=1.0
    )

def _init_scene(self) -> None:
    assets_root_path = get_assets_root_path()
    usd_path = assets_root_path + self.cfg.isaac_usd_path
    open_stage(usd_path)
    self.scene = Scene()
    self.scene.add_default_ground_plane()
    set_camera_view(eye=[2, 2, 2], target=[0, 0, 0])

    # ロボット読み込み
    define_prim(self.cfg.robot_prim_path, "Xform")
    self.robot = XFormPrimView(self.cfg.robot_prim_path)

def _init_sensors(self) -> None:
    # 触覚センサ
    self.tactile = TactileSensor(
        prim_path=self.cfg.tactile_prim_path,
        name="tactile",
        translation=np.array([0, 0, 0.01]),
    )
    self.tactile.initialize()

def _init_spaces(self) -> None:
    # 観測・行動空間定義

```

```

self.obs_dim = 3 + 3 + 48 # 画像特徴 + 関節 + 触覚
self.act_dim = 6 # エンドエフェクタ速度
self.observation_space = gym.spaces.Box(
    low=-np.inf, high=np.inf, shape=(self.obs_dim,), dtype=np.float32
)
self.action_space = gym.spaces.Box(
    low=-1.0, high=1.0, shape=(self.act_dim,), dtype=np.float32
)

def reset(self) -> np.ndarray:
    self.simulation_context.reset()
    self.tactile.reset()
    obs = self._get_obs()
    return obs

def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, Dict]:
    # 安全フィルタ適用 (MPC)
    safe_action = self._safety_filter(action)
    self.robot.apply_action(
        ArticulationAction(joint_velocities=safe_action)
    )
    self.simulation_context.step(render=not self.headless)
    obs = self._get_obs()
    reward = self._compute_reward()
    done = self._check_done()
    info = {}
    return obs, reward, done, info

def _get_obs(self) -> np.ndarray:
    # 画像特徴抽出 (簡易CNN)
    img = self._get_camera_image()
    img_feat = self._encode_image(img)
    joints = self.robot.get_joint_positions()
    tactile_vals = self.tactile.get_sensor_values()
    obs = np.concatenate([img_feat, joints, tactile_vals])
    return obs.astype(np.float32)

def _encode_image(self, img: np.ndarray) -> np.ndarray:
    # 軽量CNN (推論用)
    if not hasattr(self, "_img_encoder"):

```

```

        self._img_encoder = (
            nn.Sequential(
                nn.Conv2d(3, 16, 3, 2),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(16 * 32 * 32, 3),
            )
            .to(self.cfg.device)
            .eval()
        )
    with torch.no_grad():
        x = torch.from_numpy(img).unsqueeze(0).to(self.cfg.device)
        feat = self._img_encoder(x).squeeze(0).cpu().numpy()
    return feat

def _get_camera_image(self) -> np.ndarray:
    # カメラ画像取得 (256x256x3)
    import omni.isaac.synthetic_utils as su
    viewport = su.get_viewport_from_camera(self.cfg.camera_prim_path)
    return su.get_rgb(viewport)

def _compute_reward(self) -> float:
    # 報酬はタスク依存 (簡易)
    return 1.0

def _check_done(self) -> bool:
    # 終了条件
    return False

def _safety_filter(self, action: np.ndarray) -> np.ndarray:
    # MPCベース安全フィルタ
    if not hasattr(self, "_mpc"):
        self._setup_mpc()
    x0 = self._get_state()
    self.mpc.x0 = x0
    self.mpc.set_initial_guess()
    u = self.mpc.make_step(x0)
    return u.flatten()

def _setup_mpc(self) -> None:

```

```

model_type = "continuous"
model = Model(model_type)
# 状態: 関節位置・速度
model.set_variable("_x", "q", (6, 1))
model.set_variable("_x", "dq", (6, 1))
# 入力: 関節速度
model.set_variable("_u", "u", (6, 1))
# ダイナミクス
model.set_rhs("q", model.x["dq"])
model.set_rhs("dq", model.u["u"])
model.setup()
self.mpc = MPC(model)
setup_mpc = {
    "n_horizon": self.cfg.mpc_horizon,
    "t_step": self.cfg.mpc_dt,
    "n_robust": 0,
    "store_full_solution": False,
}
self.mpc.set_param(**setup_mpc)
# 制約
self.mpc.set_rterm(u=1e-2)
self.mpc.setup()

def _get_state(self) -> np.ndarray:
    q = self.robot.get_joint_positions()
    dq = self.robot.get_joint_velocities()
    return np.concatenate([q, dq])

# 評価用コールバック
class IsaacEvalCallback(EvalCallback):
    def __init__(self, eval_env, **kwargs):
        super().__init__(eval_env, **kwargs)

    def _on_step(self) -> bool:
        # 安全メトリクス記録
        return super()._on_step()

# メイン

```

```

def main():
    cfg = TrainConfig()
    set_random_seed(cfg.seed)

    # ログ・保存ディレクトリ
    Path(cfg.log_dir).mkdir(exist_ok=True)
    Path(cfg.save_dir).mkdir(exist_ok=True)

    # 環境生成
    rclpy.init()
    env = IsaacSimEnv(cfg, headless=True)
    check_env(env)
    env = Monitor(env)
    env = DummyVecEnv([lambda: env])
    env = VecNormalize(env, norm_obs=True, norm_reward=True)

    # 評価環境
    eval_env = IsaacSimEnv(cfg, headless=True)
    eval_env = Monitor(eval_env)
    eval_env = DummyVecEnv([lambda: eval_env])
    eval_env = VecNormalize(eval_env, norm_obs=True, norm_reward=False)

    # PPOモデル
    policy_kwargs = dict(
        activation_fn=nn.Tanh,
        net_arch=[dict(pi=[256, 256], vf=[256, 256])],
    )
    model = PPO(
        "MlpPolicy",
        env,
        policy_kwargs=policy_kwargs,
        verbose=1,
        tensorboard_log=cfg.log_dir,
        device=cfg.device,
        n_steps=2048,
        batch_size=64,
        n_epochs=10,
        learning_rate=3e-4,
        clip_range=0.2,
        gamma=0.99,

```

```

        gae_lambda=0.95,
        max_grad_norm=0.5,
        ent_coef=0.0,
        vf_coef=0.5,
    )

    # コールバック
    eval_callback = IsaacEvalCallback(
        eval_env,
        best_model_save_path=cfg.save_dir,
        log_path=cfg.log_dir,
        eval_freq=cfg.eval_interval * env.num_envs * 2048,
        deterministic=True,
        render=False,
    )
    ckpt_callback = CheckpointCallback(
        save_freq=100 * env.num_envs * 2048,
        save_path=cfg.save_dir,
        name_prefix="model",
    )

    # 学習
    model.learn(
        total_timesteps=cfg.num_episodes * 2048,
        callback=[eval_callback, ckpt_callback],
    )

    # 保存
    model.save(os.path.join(cfg.save_dir, "final_model"))
    env.save(os.path.join(cfg.save_dir, "vec_normalize.pkl"))

    # 終了
    env.close()
    eval_env.close()
    rclpy.shutdown()
    env.simulation_app.close()

if __name__ == "__main__":
    main()

```

エンジニアリングへの影響、トレードオフ、運用上のリスク

- サンプル効率 vs 忠実度：トルクレベル方策はより多くのサンプルと正確なダイナミクスを必要とする。位置／インピーダンス行動を用いてサンプル需要を下げる。
- 知覚遅延と遮蔽：劣った物体姿勢推定は方策性能を損なう。時間フィルタおよび多視点融合を備えた知覚パイプラインを設計する。
- 安全性とハードウェア摩耗：接触を伴う学習は高衝撃力を生じ得る。保護制約、シミュレーション優先トレーニング、保守的な探索を用いる。
- 転移リスク：十分なランダムマイゼーションまたはダイナミクスキャリブレーションなしでは、方策はシミュレーションアーティファクトを悪用する。制約付き低力タスクから段階的にハードウェアで検証する。
- 計算およびセンサ要件：触覚アレイおよび高レートフィードバックは安定性を改善するが、システム複雑性および処理遅延を増大させる。

設計トレードオフは安全性と保守性を優先すべきである。実用ヒューマノイドシステムでは、階層制御器および制約付き RL を用いて運用リスクを境界付けつつ、適応把持および複雑な相互作用のために RL を活用する。

38.3 強化学習を用いた協調ロボティクス

操縦と移動の例に続き、協調タスクでは全身スキルを複数のヒューマノイドにわたる調整された意思決定と組み合わせる必要がある。単体エージェントのバランスおよび把持の技術は、エージェントが荷重を共有し、軌道を調整し、ロボット間の安全制約を尊重しなければならない場合に異なるスケリング特性を示す。

協調問題の定式化と運用上の意義. 工業および災害対応の現場では、2 台以上のヒューマノイドが大型のペイロードを輸送したり、部品を組み立てたり、大型物体を共同で操作する必要がある。工学上の課題は、以下を満たす方策を合成することである：

- 高自由度を持つロボット間で滑らかで力整合性のある動作を生成すること；
- 部分的観測とノイジーなセンシング（IMU、力・トルクセンサ、ビジョン）下で動作すること；
- 通信遅延と断続的なパケット損失に耐えること；および
- ロボット間の衝突を回避しペイロードの安定性を維持することで安全性を保証すること。

技術解析. マルチエージェント強化学習（MARL）はシステムを N 個のエージェントとして扱い、各エージェントは局所観測 o_i 、行動 a_i 、そして共有される環境状態 s を持つ。結合報酬は

$$[H]G_t = \sum_{k=t}^{\infty} \gamma^{k-t} \sum_{i=1}^N r_i(s_k, a_{1:k}, \dots, a_{N:k}), \quad (322)$$

であり、 γ は割引率、 r_i は各エージェントの報酬寄与である。訓練時集中・実行時分散（CTDE）はヒューマノイドにとって実用的な工学選択である。CTDE は訓練中に集中クリティック $Q_{\phi}(s, a_{1:N})$ を用いてマルチエージェントの信用割当を解決する。各エージェントはリアルタイム制御のために分散方策 $\pi_{\theta_i}(a_i|o_i)$ を保持する。

CTDE 下でのエージェント i に対する一般的なアクタ・クリティック勾配は

$$[H]\nabla_{\theta_i} J(\theta_i) = \mathbb{E} \left[\nabla_{\theta_i} \log \pi_{\theta_i}(a_i|o_i) A_i(s, a_{1:N}) \right], \quad (323)$$

であり、アドバンテージ $A_i(s, a_{1:N}) = Q_\phi(s, a_{1:N}) - V_\psi(s)$ で、 V_ψ は集中的価値ベースラインである。この定式化は分散を減らし、 Q_ϕ を通じて明示的な協調手がかりを提供する。

設計上の考慮と報酬エンジニアリング. 実用的な報酬設計は協調目標と安全性をバランスさせる。荷重分担には密なシェーピング報酬（例：力不均衡ペナルティ）を用い、経路成功には疎なタスク完了報酬を用いる。典型的な構成要素：

- ペイロード安定性：ペイロードの角速度と並進偏差をペナルティ；
- 力配分：エージェント間のエンドエフェクタ力測定値の大きな差をペナルティ；
- 衝突回避：許容面外のロボット間接触に対する硬いペナルティ；
- エネルギー・コスト：激しい動作を避けるためのトルク正則化。

カリキュラム学習はペイロード質量、環境障害物、通信遅延を段階的に増やし、ハードウェアへの転移を改善する。

実装レシピ. シミュレーション (Isaac Sim) で接触可能エンドエフェクタを持つ複数のヒューマノイドアクタをインスタンス化する。摩擦、質量、センサノイズにドメイン・ランダムイゼーションを適用。連続制御におすすめの MARL アルゴリズムは MAPPO (マルチエージェント PPO) と MADDPG (マルチエージェント深層決定の方策勾配) である。エージェントが同質の場合、パラメータ共有が訓練を加速させることが多い。

PyTorch 風擬似コードにおける集中クリティック更新と分散アクタステップの例. このコードは連続行動と共有クリティックを仮定し、コメントは簡潔かつ具体的にすること。

コードサンプル 131 集中クリティック更新と分散アクタ (PyTorch 風)

```
import torch
import torch.nn.functional as F
from typing import List, Tuple

def update_critics_and_policies(
    obs: List[torch.Tensor],
    actions: List[torch.Tensor],
    rewards: torch.Tensor,
    next_obs: List[torch.Tensor],
    dones: torch.Tensor,
    centralized_state: torch.Tensor,
    centralized_next_state: torch.Tensor,
    policies: List[torch.nn.Module],
    critic: torch.nn.Module,
    critic_optimizer: torch.optim.Optimizer,
    pi_optimizers: List[torch.optim.Optimizer],
    gamma: float,
) -> Tuple[torch.Tensor, List[torch.Tensor]]:
    """
```

```

##### マルチエージェント集中クリティック・分散アクタ更新
##### 返り値: (critic_loss, actor_losses)
##### """
    device = centralized_state.device
    batch_size = centralized_state.size(0)
    n_agents = len(policies)

    # --- クリティックターゲット計算 (TD(0)) ---
    with torch.no_grad():
        next_actions = []
        log_probs = []
        for i, pi in enumerate(policies):
            dist = pi(next_obs[i])
            act = dist.sample() # 再パラメータ化トリック
            next_actions.append(act)
            log_probs.append(dist.log_prob(act).sum(dim=-1, keepdim=True))

        q_next = critic(centralized_next_state, torch.cat(next_actions, dim=-1))
        target_q = rewards.sum(dim=1, keepdim=True) + gamma * (1.0 - done.float()) *

    # --- クリティック更新 ---
    q_pred = critic(centralized_state, torch.cat(actions, dim=-1))
    critic_loss = F.mse_loss(q_pred, target_q)
    critic_optimizer.zero_grad()
    critic_loss.backward()
    # 勾配クリッピング
    torch.nn.utils.clip_grad_norm_(critic.parameters(), max_norm=1.0)
    critic_optimizer.step()

    # --- アクタ更新 (分散) ---
    actor_losses = []
    for i, pi in enumerate(policies):
        dist = pi(obs[i])
        sampled_action = dist.sample()
        log_prob = dist.log_prob(sampled_action).sum(dim=-1, keepdim=True)

        all_actions = actions.copy()
        all_actions[i] = sampled_action
        q_val = critic(centralized_state, torch.cat(all_actions, dim=-1))

```

```

# エントロピー正則化付き目的
entropy = -log_prob
actor_loss = -(q_val + 0.01 * entropy).mean()

pi_optimizers[i].zero_grad()
actor_loss.backward()
torch.nn.utils.clip_grad_norm_(pi.parameters(), max_norm=1.0)
pi_optimizers[i].step()
actor_losses.append(actor_loss.detach())

return critic_loss.detach(), actor_losses

```

実用的なセンサおよび通信エンジニアリング。手首に力・トルクセンシングを用い、コンプライアント制御でミスマッチを吸収する。状態ブロードキャストには低遅延 UDP または ROS2 DDS と時刻同期を実装する。制御スタックを設計し、危険事象では局所安全反射が学習コマンドを上書きするようにする。

評価とハードウェアへの転移。ランダム化された遅延とセンサノイズをシミュレーションで検証する。シミュレートされたエピソードをリアルタイムで再生し、駆動限界をテストする。転移ではドメイン・ランダム化セッションを段階的に減らし、フル質量テスト前にスケール済みペイロードで検証する。接触力と関節トルクマージンを継続的に監視する。

工学上の影響、トレードオフ、リスク：

1. 集中クリティックは学習を簡素化するが、訓練の複雑性を増やし、訓練中にフル状態へのアクセスを必要とする。
2. パラメータ共有は同質チームのサンプル複雑性を減らす；異質役割への特殊化を制限する。
3. 攻撃的な報酬シェーピングは脆い方策を生むことがある；段階的カリキュラムと安全オーバーライドを優先する。
4. 通信損失と遅延は運用上のリスクを生む；パケットドロップ下で制御が優雅に劣化するように設計する。

注意深い報酬設計、頑健なセンシング、段階的ハードウェアテストが協調ヒューマノイド展開における故障モードを減らす。

38.4 ロボティクスにおける強化学習の将来動向

物体操作およびマルチエージェント協調を基盤に、今後 10 年間に於いて人間型ロボット的能力を形作る強化学習の動向を以下に示す。これらの動向は、サンプル効率、安全性、転移、および全身制御に必要な計算・ハードウェア共設計に対処する。

モデルベースおよびハイブリッド手法は、訓練データ要件を削減しながら頑健性を維持する。モデルベース強化学習 (MBRL) は学習済みダイナミクスモデル $p_\phi(s_{t+1}|s_t, a_t)$ を構築し、計画または想像に基づく更新に利用する。工学的重要性：ダイナミクスモデルは、全身接触学習の高価な現実世界試

行を償却できる．実用的な目的関数は方策学習とモデル適合を結合する：

$$[H]J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^T \gamma^t r(s_t, a_t) \right], \quad \mathcal{L}_{\text{model}}(\phi) = \mathbb{E}_{(s,a,s') \sim \mathcal{D}} [\|s' - \hat{s}'_\phi(s,a)\|^2]. \quad (324)$$

設計上のトレードオフ：モデルバイアスは収束を高速化するが，学習モデルが接触遷移付近で誤っている場合に破局的なエラーのリスクがある．

メタ学習および終身強化学習は，新しいタスクや変化するダイナミクスへの迅速な適応を可能にする．人間型ロボットにとって，少数ショット適応はハードウェア改修またはペイロード変更後のダウンタイムを削減する．メタ強化学習コントローラは，適応コストを最小化する初期化 θ_0 を学習する：

$$\min_{\theta_0} \mathbb{E}_{\tau \sim p(\mathcal{T})} [\mathcal{L}_\tau(\text{Adapt}(\theta_0, \mathcal{D}_\tau))].$$

運用上の意味：メタ学習済み方策は，限局的本体感覚データを用いてオンボードで適応でき，フィールド修理および再構成を可能にする．

階層強化学習およびモジュラー方策は，振る舞いの複雑さをスケールリングする．制御を高レベルタスク選択と低レベル安定化コントローラに分割する．例えば：

- 高レベルプランナはタスク空間で目標を発行する（到達，歩行，把持）．
- ミドルレベルは関節空間軌道と接触スケジュールを割り当てる．
- 低レベルトルクコントローラは安定性と安全性を強制する．

このモジュール性により，バランスコントローラを操作タスク間で再利用でき，安全上重要なコンポーネントの認証を簡素化する．

対比および表現学習は，知覚から制御までのパイプラインを改善する．人間型ロボットにとって，マルチモーダルセンサからコンパクトな潜在状態 $z = f_\psi(o)$ を学習することで，方策入力次元を削減し，センサドロップアウトに対する頑健性を向上させる．対比目的は，タスク関連不変性を保持する表現を生み，シミュレーションとハードウェア間の切り替えに有利である．

安全および制約付き強化学習は，人間型ロボットの展開において標準となる．安全強化学習は，ラグランジュ手法または射影層を通じて制約 $g_i(s_t, a_t) \leq 0$ を強制する．実用的なアプローチ：

- 接触力および関節限界に対する安全クリティックを用いた制約付き方策最適化．
- 高リスク時にモデル予測コントローラへ切り替えるランタイムスーパーバイザ．

工学上のリスク：純粋に学習された安全層は，誤った自信を生む可能性がある．解析的バウンドを用いたハイブリッド検証が推奨される．

シミュレーション忠実度，微分可能物理，およびリアルツーシム共設計は，sim2real ギャップを閉じる．微分可能シミュレータは，現実世界損失信号を用いた勾配ベースシステム同定および方策微調整を可能にする．ドメインランダムマイゼーションは依然として有用であるが，アクチュエータダイナミクスに対する標的システム同定と組み合わせるべきである．

大規模事前学習制御モデルおよび他ドメインからの転移が登場する．事前学習済みビジョンモデルに類似し，多様なシミュレーションタスクで訓練された制御モデルを，特定の間型ロボット目的に微調整できる．これはタスクあたり計算コストを削減し，開発サイクルを加速する．

分散およびクラウドエッジ訓練パイプラインは，イテレーションを加速する．現実の間型ロボットタスクにとって，オフラインデータ収集およびオンデバイス適応は，レイテンシと帯域幅を balan

スさせる必要がある。典型的なパイプライン：

1. ドメインランダムマイゼーションを用いてシミュレーションで軌道を収集する。
2. GPU クラスタで方策を訓練し、並列シミュレーションシードで評価する。
3. 蒸留済み方策をエッジハードウェアに展開し、短時間のロボット上適応を実行する。

計算上の意味：訓練コストは環境忠実度とともに上昇し、リソース計画およびハードウェア認識モデル圧縮を必要とする。

残差および安全シェイピング制御は、古典的コントローラと強化学習の橋渡しをする。実証済み逆ダイナミクスコントローラの上で、残差トルクまたは補正項を学習するために強化学習を用いる。残差強化学習は探索リスクを削減し、既存の安定性保証を活用する。

ヒューマンインザループおよびデモンストレーションからの学習は、複雑なタスクおよび社会的文脈にとって依然として重要である。デモンストレーションは報酬学習をシードし、探索を制約する。人間型ロボット HRI にとって、人間の監視を伴う対話的報酬シェイピングは、安全性およびユーザ受容性を改善する。

実装スケッチ：Isaac Sim における分散 PPO 訓練とカリキュラムおよびドメインランダムマイゼーション。

コードサンプル 132 Isaac Sim および PPO を用いた分散訓練ループ

```
import os
import torch
import torch.nn as nn
from torch.distributions.normal import Normal
import rclpy
from rclpy.node import Node
from isaacgym import gymtorch, gymapi
from isaacgymenvs.tasks.base.vec_task import VecTask
from omegaconf import DictConfig
import hydra
from typing import Dict, List, Tuple
import numpy as np
from datetime import datetime

class PolicyNetwork(nn.Module):
    """アクタ・クリティック方策ネットワーク"""
    def __init__(self, obs_dim: int, act_dim: int, hidden: int = 256):
        super().__init__()
        self.backbone = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.Tanh(),
            nn.Linear(hidden, hidden), nn.Tanh())
        self.mean = nn.Linear(hidden, act_dim)
```

```

        self.log_std = nn.Parameter(torch.zeros(act_dim))
        self.value = nn.Linear(hidden, 1)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            nn.init.orthogonal_(m.weight, gain=0.01)
            nn.init.zeros_(m.bias)

    def forward(self, x):
        feat = self.backbone(x)
        mean = self.mean(feat)
        std = self.log_std.exp().expand_as(mean)
        value = self.value(feat).squeeze(-1)
        return mean, std, value

class RolloutBuffer:
    """ ロールアウトデータ格納 """
    def __init__(self):
        self.obs, self.act, self.logp, self.val, self.rew, self.ret = [], [], [], [], [],

    def push(self, obs, act, logp, val, rew):
        self.obs.append(obs)
        self.act.append(act)
        self.logp.append(logp)
        self.val.append(val)
        self.rew.append(rew)

    def compute_returns(self, gamma: float = 0.99, lam: float = 0.95, last_val: torch.
        """ GAE-Lambda でリターン計算 """
        rewards = torch.cat(self.rew)
        values = torch.cat(self.val + [last_val])
        gae = 0
        returns = []
        for step in reversed(range(len(self.rew))):
            delta = rewards[step] + gamma * values[step + 1] - values[step]
            gae = delta + gamma * lam * gae
            returns.insert(0, gae + values[step])
        self.ret = returns

```

```

class PPOAgent(Node):
    def __init__(self, cfg: DictConfig):
        super().__init__('ppo_agent')
        self.cfg = cfg
        self.device = torch.device(cfg.device)
        self.envs = hydra.utils.instantiate(cfg.task)
        self.policy = PolicyNetwork(
            self.envs.observation_space.shape[0],
            self.envs.action_space.shape[0]).to(self.device)
        self.optimizer = torch.optim.Adam(self.policy.parameters(), lr=cfg.lr, eps=1e-
        self.buffer = RolloutBuffer()
        self.epoch = 0
        self.eval_every = cfg.eval_every
        self.checkpoint_dir = cfg.checkpoint_dir
        os.makedirs(self.checkpoint_dir, exist_ok=True)

    @torch.no_grad()
    def collect_rollouts(self) -> Dict[str, torch.Tensor]:
        """並列シムでバッチロールアウト収集"""
        obs = self.envs.reset()
        self.buffer = RolloutBuffer()
        for _ in range(self.cfg.steps_per_env):
            mean, std, val = self.policy(torch.from_numpy(obs).to(self.device))
            dist = Normal(mean, std)
            act = dist.sample()
            logp = dist.log_prob(act).sum(-1)
            next_obs, rew, done, info = self.envs.step(act.cpu().numpy())
            self.buffer.push(
                torch.from_numpy(obs).to(self.device),
                act,
                logp,
                val,
                torch.from_numpy(rew).to(self.device))
            obs = next_obs
        _, _, last_val = self.policy(torch.from_numpy(obs).to(self.device))
        self.buffer.compute_returns(self.cfg.gamma, self.cfg.lam, last_val)
        return {
            'obs': torch.cat(self.buffer.obs),
            'act': torch.cat(self.buffer.act),
            'logp': torch.cat(self.buffer.logp),

```

```

        'ret': torch.cat(self.buffer.ret),
        'val': torch.cat(self.buffer.val)}

def ppo_loss(self, batch: Dict[str, torch.Tensor]) -> torch.Tensor:
    """PPOクリップ損失"""
    mean, std, val = self.policy(batch['obs'])
    dist = Normal(mean, std)
    new_logp = dist.log_prob(batch['act']).sum(-1)
    ratio = (new_logp - batch['logp']).exp()
    adv = (batch['ret'] - batch['val']).detach()
    adv = (adv - adv.mean()) / (adv.std() + 1e-8)
    surr1 = ratio * adv
    surr2 = torch.clamp(ratio, 1 - self.cfg.clip, 1 + self.cfg.clip) * adv
    actor_loss = -torch.min(surr1, surr2).mean()
    critic_loss = 0.5 * (batch['ret'] - val).pow(2).mean()
    entropy = dist.entropy().sum(-1).mean()
    return actor_loss + self.cfg.vf_coef * critic_loss - self.cfg.ent_coef * entropy

def train_epoch(self):
    batch = self.collect_rollouts()
    for _ in range(self.cfg.epochs_per_update):
        idx = torch.randperm(len(batch['obs']))
        for start in range(0, len(batch['obs']), self.cfg.mini_batch):
            mb = {k: v[idx[start:start + self.cfg.mini_batch]] for k, v in batch.items()}
            loss = self.ppo_loss(mb)
            self.optimizer.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(self.policy.parameters(), self.cfg.max_grad_norm)
            self.optimizer.step()
    self.epoch += 1

def evaluate_on_robot(self):
    """短時間ハードウェアテスト (ROS2サービス経由) """
    client = self.create_client(Trigger, '/hardware/evaluate')
    if not client.wait_for_service(timeout_sec=1.0):
        self.get_logger().warn('ハードウェア評価サービス未接続')
        return
    req = Trigger.Request()
    future = client.call_async(req)
    rclpy.spin_until_future_complete(self, future)

```

```

self.get_logger().info(f'評価結果:{future.result()}')

def save_checkpoint(self):
    ckpt = {
        'epoch': self.epoch,
        'model': self.policy.state_dict(),
        'optim': self.optimizer.state_dict()}
    path = os.path.join(self.checkpoint_dir, f'ckpt_{self.epoch:06d}.pt')
    torch.save(ckpt, path)
    torch.save(ckpt, os.path.join(self.checkpoint_dir, 'latest.pt'))

@hydra.main(version_base=None, config_path='conf', config_name='config')
def main(cfg: DictConfig):
    rclpy.init()
    agent = PPOAgent(cfg)
    for _ in range(cfg.total_epochs):
        agent.train_epoch()
        if agent.epoch % agent.eval_every == 0:
            agent.evaluate_on_robot()
            agent.save_checkpoint()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

具体的な工学上の意味, トレードオフ, および運用上のリスク:

- サンプル効率対モデルバイアス: モデルベースの利得は物理的摩耗を削減するが, 頑健なモデル検証を必要とする.
- 計算対レイテンシ: 高忠実度訓練はクラウド GPU を要求し, 推論は組み込み CPU またはアクセラレータに収まる必要がある.
- 安全性対自律性: より強い自律性は能力を高めるが, 認証および責任の複雑さを上げる.
- メンテナンスおよび再現性: 大規模強化学習システムは, バージョン管理されたシミュレータ, データセット, および決定論的シードをフィールド信頼性のために必要とする.

検証済み低レベルコントローラと学習済み高レベル方策を組み合わせたハイブリッドアーキテクチャを採用する. 安全性検証, アクチュエータ認識モデル, および軽量オンデバイス適応を優先し, 人間型ロボットシステムに強化学習を運用化する.

協働ロボティクス

39 マルチロボットシステムの導入

39.1 協働の利点

これまでに開発してきた学習・知覚・運動計画のプリミティブを基に、複数のヒューマノイドが協働することで単独ユニットを上回る理由を現実的なタスクにおいて定量化する。以下の解析では、制約と通信制約を測定可能な運用上の利点に結びつける。

協働は、産業・災害対応・医療の現場でヒューマノイドチームが直面する特定の運用上の問題——タスク完了時間の長さ、部分的観測可能性、単一障害点、限られたアクチュエータ出力——に対処する。問題定義：空間的・時間的に結合したサブタスクに分解可能なミッションが与えられたとき、現実的な通信オーバーヘッドのもとで協働が完了時間、成功確率、資源消費に与える影響を決定する。

技術的解析：

・エンジニアリングへの含意を伴う主要な利点：

1. 並列性と特化による完了時間の短縮。機械的または知覚的に最も適したエージェントにサブタスクを割り当てることで、タスクごとの遅延を削減する。
2. 冗長性による堅牢性とグレースフルデグラデーションの向上。冗長化により、アクチュエータまたはセンサ故障に対するミッション失敗確率を下げる。
3. 分散センシングによる状況認識の向上で、知覚不確実性を低減する。
4. エネルギーと作業負荷の平準化により、チームの運用持続時間を延長する。
5. 新たな能力の創出、例：2 台のヒューマノイドが協調して単独では扱えない重いペイロードを持ち上げる。

・性能とリスクのための定量的モデル：

1. 通信オーバーヘッドを含むスピードアップ。ベースラインの直列実行時間 T_1 のミッションをモデル化。 N 台のエージェントによる理想的並列時間は T_1/N 。現実的な同期と通信はオーバーヘッド $T_{\text{comm}}(N)$ を加える。単純なモデル：

$$[H]T_N = \frac{T_1}{N} + T_{\text{sync}} + T_{\text{comm}}(N), \quad (325)$$

ここで $T_{\text{comm}}(N) = \alpha N^\beta$ は増加する調整コストを表す。スピードアップ $S(N) = T_1/T_N$ は $\beta > 0$ のとき N が大きくなるにつれて収束する。

2. 冗長性を用いたミッション成功確率。各エージェントが独立確率 p でクリティカルサブタスクに成功し、 k 台の冗長エージェントがそのサブタスクを試行する場合：

$$[H]P_{\text{success}} = 1 - (1 - p)^k. \quad (326)$$

密結合タスクでは独立性は楽観的；設計検証で条件付き確率を用いて依存性をモデル化する。

3. 通信遅延とのトレードオフ。周期 T_c の制御ループに対し、 $T_{\text{comm}}(N) < \gamma T_c$ を要求して不安定化を回避し、設計係数 $\gamma \in (0, 1)$ を用いる。この制約がネットワーク設計と分散制御選択を促す。

実装パターンとアルゴリズム：

- アーキテクチャ：
 1. 集中型タスクアロケータ：アルゴリズム複雑度は低いが単一障害点，高容量基地局が利用可能な場合に有用.
 2. 市場ベース（オークション）割当：スケーラブルで能力とコストを自然に均衡化し，ノード損失に対して堅牢.
 3. 合意ベース分散制御：低遅延の局所決定，断続的リンク下で堅牢.
- 実用的タスク割当選択は以下に依存：
 - タスク分解粒度.
 - 通信信頼性と帯域.
 - 運動制御サブタスクのリアルタイム制約.

例：軽量市場ベースタスクアロケータを ROS2 スタイル Python ノードとして実装. ノードは入札をブロードキャストし，最高入札がタスクを獲得する. これはエンジニアリングの出発点であり，本番スケジューラではない.

コードサンプル 133 Simple auction-based bid broadcast (ROS2-like pseudocode)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from rclpy.timer import Timer
from rclpy.time import Time
from auction_interfaces.msg import Task, Bid, Winner
from builtin_interfaces.msg import Time as RosTime
import threading
from typing import Dict, List, Optional
import time
import uuid

class AuctionBidder(Node):
    def __init__(self) -> None:
        super().__init__('auction_bidder')
        self.declare_parameter('robot_id', str(uuid.uuid4()))
        self.robot_id: str = self.get_parameter('robot_id').value

        self.declare_parameter('manip_strength', 1.0)
        self.manip_strength: float = self.get_parameter('manip_strength').value
```

```

qos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    history=HistoryPolicy.KEEP_LAST,
    depth=10
)

self.task_sub = self.create_subscription(
    Task, '/task_auction', self.task_callback, qos)

self.bid_pub = self.create_publisher(Bid, '/auction_bids', qos)

self.winner_sub = self.create_subscription(
    Winner, '/auction_winner', self.winner_callback, qos)

self.pending_tasks: Dict[str, Task] = {}
self.lock = threading.Lock()

self.bid_window_sec = 0.5
self.bid_timer: Optional[Timer] = None

def task_callback(self, msg: Task) -> None:
    with self.lock:
        self.pending_tasks[msg.task_id] = msg
    # 入札ウィンドウタイマー開始
    if self.bid_timer is None:
        self.bid_timer = self.create_timer(self.bid_window_sec, self.bid_all_pending)

def bid_all_pending(self) -> None:
    with self.lock:
        tasks = list(self.pending_tasks.values())
        self.pending_tasks.clear()
    for task in tasks:
        bid = self.compute_bid(task)
        self.bid_pub.publish(bid)
    # タイマー停止
    if self.bid_timer:
        self.destroy_timer(self.bid_timer)
        self.bid_timer = None

```

```

def compute_bid(self, task: Task) -> Bid:
    # 推定作業時間を能力で割り、逆数を入札値とする
    est_time = task.workload / (self.manip_strength + 1e-6)
    bid_value = 1.0 / est_time
    bid = Bid()
    bid.task_id = task.task_id
    bid.robot_id = self.robot_id
    bid.bid_value = bid_value
    bid.stamp = self.get_clock().now().to_msg()
    return bid

def winner_callback(self, msg: Winner) -> None:
    # 自身が落札した場合の処理を追加可能
    pass

def main(args=None) -> None:
    rclpy.init(args=args)
    node = AuctionBidder()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

評価指標と試験プロトコル：

- 完了時間，タスクあたりエネルギー，タスク成功率，通信負荷を測定する．
- アブレーションを実施： N ，ネットワーク遅延，エージェントごとの故障率を変化させる．
- ハードウェア試験の前に，シミュレーション（Isaac Sim）を用いて衝突，同期失敗，消費電力を計測する．

エンジニアリングへの含意，トレードオフ，運用上のリスク：

- トレードオフ：
 1. スケーラビリティ vs. 調整コスト：ヒューマノイドを増やすと冗長性が増すが遅延とデッドロックリスクも増大．
 2. 集中型の単純さ vs. 回復力：集中制御は最適化を単純化するがミッションクリティカルノー

ドを生む。

3. 特化 vs. 柔軟性：高度に特化したエージェントは指定サブタスク時間を短縮するが、劣化モードでの互換性を低下させる。

• 運用上のリスク：

1. 通信障害は分散システムを単独ロボットの集合に変え、部分的ミッション失敗を引き起こす。
2. 創発的な安全でない相互作用：協調マニピュレーションにはロボット間衝突回避と同期インピーダンス制御を含める必要がある。
3. セキュリティ：保護されていない通信は入札や状態のスプーフィングを許し、安全でない割当を引き起こす。

• 推奨設計管理：

1. QoS と局所フォールバックにより $T_{\text{comm}}(N)$ を制御周期の割合以下に制限する。
2. 安全上クリティカルな協調動作には形式検証を用いる。
3. 段階的冗長性を実装：フェイルオーバー動作はグレースフルに劣化する。

具体的なエンジニアリング・トレードオフがシステムアーキテクチャ決定を導く。(325) を用いて期待スピードアップを定量化し、(326) で冗長性仮定を検証する。通信信頼性、セキュリティ、検証済み調整ポリシーを最優先し、協働ヒューマノイドチームを安全に実用的に展開する。

39.2 通信プロトコルの設計

多人数型ヒューマノイド協働の定量化された利点を踏まえ、通信プロトコル設計はそれらの利点を信頼できる動作に変換する。効果的なプロトコルは共有意図を協調動作に変換しながら、安全性、遅延上限、障害時のグレースフルデグラデーションを維持する。

問題定義. 複数のヒューマノイドエージェントは、協調マニピュレーション、人間誘導護送、分散点検などのタスクを調整するために、状態、目標、センサデータを交換しなければならない。プロトコルは競合する要求を満たさねばならない：

- 制御に関連する状態の低遅延。
- 安全上重要なコマンドの高信頼性。
- 歩行位相等の動作同期のためのバウンドジッタ。
- 高密度センサストリーム（ポイントクラウド、画像）の帯域効率。
- ノード障害とネットワーク分断への耐性。
- 不正コマンドを防ぐためのセキュリティ。

技術分析. プロトコル設計は階層化された責任に関心を分離する：

1. データ分類. メッセージをタイムリネスとクリティカルリティで分割：
 - 制御ハートビート：バランスとポーズのための小さく高レートの状態ベクトル。
 - コマンドとネゴシエーション：権威あるタスク割り当てと確認応答。
 - バルクセンサ交換：協調知覚のための圧縮知覚データ。
2. トランスポート選択. メッセージクラスをトランスポートセマンティクスにマッチ：
 - 低遅延ハートビートとストリーミングのためのカスタムシーケンシング付き UDP。

- クリティカルコマンドのための信頼できるトランスポート（TCP、DDS 信頼 QoS）。
 - ネットワークがサポートする場合、一対多アナウンスのためのマルチキャスト。
3. サービス品質（QoS）パラメータ. 明示的に設計する：
- 各メッセージクラスごとの遅延予算 L_{\max} とジッタ予算 J_{\max} 。
 - 信頼性ターゲット R (L_{\max} 以内の到達確率)。
 - 制御ループ向けの最大許容メッセージ損失 p_{target} 。

単純な信頼性モデルは再送戦略を導く。各パケット損失確率が p で k 回の再送を許可する場合、成功確率 P_{success} は

$$[H]P_{\text{success}} = 1 - p^{k+1}. \quad (327)$$

目標信頼性 R_{target} を達成する最小再送回数 k を解くと

$$[H]k \geq \left\lceil \frac{\log(1 - R_{\text{target}})}{\log p} \right\rceil - 1. \quad (328)$$

これらの式を用いてコマンドメッセージの再送予算を次元決定する。再送が遅延を違反する制御ループでは、前方誤り訂正または冗長パスを優先する。

時刻同期はセンサフュージョンと協調動作にとってクリティカルである。Precision Time Protocol (PTP) またはネットワーク時刻同期を用いて、時刻を Δt 秒以内に整合させる。エージェント間で状態をフュージョンする際には、因果関係チェックと順序外訂正のためにタイムスタンプ t_i と単調増加シーケンス番号 s を含める。

コンセンサスとリーダー選出. タスク割り当てには軽量コンセンサスまたはトークンベースリーダー選出を実装する。小規模協調チーム ($N < 10$) には高速 Paxos ライクまたは Raft-lite ヒューリスティクスを用いる。コンセンサスの遅延はしばしばネットワーク直径 D とメッセージ遅延 τ に比例し、収束時間は $T_{\text{cons}} \approx O(D \cdot \tau)$ と近似される。

セキュリティと安全性. 対称鍵または TLS ライクなチャネルセキュリティを用いて制御メッセージを認証する。クリティカルコマンドに署名して出所を保証する。安全性を強制する：コマンドはローカル妥当性チェックとハートビート消失時の短いウォッチドッグタイムアウトの後のみ受け入れられねばならない。

メッセージスキーマと状態設計. 高レートメッセージには簡潔なバイナリエンコーディングを用いる。メッセージフィールドを定義：

- seq (uint32), timestamp (uint64), sender_id (uint16), msg_type (uint8), payload_length (uint16), payload (bytes), checksum (uint32).

実装例. 以下の Python スニペットは、シーケンス番号、タイムスタンプ、CRC32 チェックサムを備えた最小 UDP ハートビート送信者を示す。ヒューマノイド状態ブロードキャストのための低遅延トランスポートを体現する。

コードサンプル 134 UDP heartbeat with sequence, timestamp and CRC32

```
#!/usr/bin/env python3
import socket
import struct
```

```

import time
import zlib
import logging
from typing import Tuple

# ロギング設定
logging.basicConfig(level=logging.INFO, format='%(asctime)s-%(levelname)s-%(message)s')
logger = logging.getLogger(__name__)

class StateBroadcaster:
    def __init__(self, sender_id: int, addr: Tuple[str, int], hz: int = 100):
        self.sender_id = sender_id
        self.addr = addr
        self.hz = hz
        self.seq = 0
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
        self.sock.settimeout(0.01) # 送信タイムアウト
        logger.info(f"Broadcaster initialized: id={sender_id}, addr={addr}, hz={hz}")

    def pack_state(self, pose: Tuple[float, float, float], joint_vels: Tuple[float, ...]):
        ts = int(time.time_ns() // 1000) # マイクロ秒タイムスタンプ
        payload = struct.pack('fff', *pose)
        payload += struct.pack(f'{len(joint_vels)}f', *joint_vels[:3])

# 最大3軸送信
        header = struct.pack('!LQH_BH', self.seq, ts, self.sender_id, 1, len(payload))
        checksum = zlib.crc32(header + payload) & 0xffffffff
        self.seq = (self.seq + 1) & 0xffffffff
        return header + payload + struct.pack('!I', checksum)

    def send(self, pose: Tuple[float, float, float], joint_vels: Tuple[float, ...]) -> bool:
        try:
            msg = self.pack_state(pose, joint_vels)
            self.sock.sendto(msg, self.addr)
            return True
        except socket.error as e:
            logger.error(f"Send failed: {e}")
            return False

    def close(self):

```

```

        self.sock.close()
        logger.info("Socket closed")

def main():
    broadcaster = StateBroadcaster(sender_id=3, addr=('192.168.1.255', 9000))
    period = 1.0 / 100
    try:
        while True:
            pose = (0.1, 1.2, 0.0)
            joint_vels = (0.05, 0.0, 0.0)
            broadcaster.send(pose, joint_vels)
            time.sleep(period)
    except KeyboardInterrupt:
        logger.info("Interrupted by user")
    finally:
        broadcaster.close()

if __name__ == '__main__':
    main()

```

設計ルールとトレードオフ：

- 周波数分割を用いる：高レートハートビートを 50–200 Hz で実行し、コマンドは低レートで。
- バルク知覚データを圧縮し、デルタまたはオブジェクトレベルサマリーのみ送信。
- 高速ディスカバリーのためのステートレスブロードキャストを優先し、コマンドには確認応答を要求。
- CRC および暗号演算のための CPU サイクルを割り当てる；ハードウェア加速は遅延を削減。

運用上の影響とリスク：

- 高レートセンサストリームで無線チャンネルを過負荷すると、ジッタと制御メッセージのドロップを引き起こす。制御トラフィックをリンクで優先し、またはネットワークを分離する。
- 過剰な再送は遅延を増大させる；信頼性とタイムリネスを（1）および（2）のようなモデルでバランスさせる。
- 不完全な時刻同期は不整合なワールドモデルを生む。プロトコル設計で時計ドリフトを検出し補償する。
- セキュリティ対策は遅延と CPU コストを追加する；リアルタイム保証への影響を評価する。

注意深く設計されたプロトコルは協調障害を削減し、安全でない動作を最小化し、多人数型ヒューマノイドチームを実環境で予測可能にする。

39.3 タスク割り当て戦略

先行する小節では、信頼性の高い通信と定義されたプロトコルがチーム行動の前提条件である理由、および協調がスループットとフォールトトレランスをいかに増幅するかを確立した。タスク割り当ては、これらのプロトコルと定量化された協調の利得を直接基に、現実世界の制約下で人型ロボットの能力にタスクをマッピングすることで構築される。

問題定義. 複数人型チームにおいて、タスク集合 T をロボット集合 R に割り当て、運動学的到達可能性、バランス安全性、タイミング、バッテリー制限を尊重しながらミッション効用を最大化する。典型的なタスクには、重い物体の搬送、協調組立、高所設備の点検、災害対応での負傷者搬出が含まれる。各タスクは複数エージェントの同時動作、時間的順序、または専用エンドエフェクタを要求する場合がある。

技術解析. ロボット r をタスク t に割り当てるためのコスト指標を定義する。コストは推定実行時間 t_{rt} 、エネルギー消費 e_{rt} 、実行中のバランス喪失確率を表す安定性リスク s_{rt} を組み合わせる。ミッション優先度を反映するために調整可能な重み w_t, w_e, w_s を用いる：

$$[H]C_{rt} = w_t t_{rt} + w_e e_{rt} + w_s s_{rt}. \quad (329)$$

能力実現可能性を $A_{rt} \in \{0, 1\}$ で表し、ロボットがタスク t に必要な到達範囲、ペイロード、エンドエフェクタを持つ場合のみ $A_{rt} = 1$ とする。

タスクが単一ロボットかつ排他的であれば、割り当ては二値最適化となる：

$$\begin{aligned} [H] \quad & \min_{x_{rt} \in \{0,1\}} \sum_{r \in R} \sum_{t \in T} C_{rt} x_{rt} \\ & \text{s.t.} \quad \sum_{r \in R} x_{rt} = 1 \quad \forall t \in T \quad (\text{各タスクが割り当てられる}) \\ & \quad \sum_{t \in T} x_{rt} \leq 1 \quad \forall r \in R \quad (\text{ロボットごとに1タスク}) \\ & \quad x_{rt} \leq A_{rt} \quad \forall r, t. \end{aligned} \quad (330)$$

複数エージェントタスクには、割り当てタプルと先行制約を導入し、問題を整数計画 (IP) または混合整数計画 (MIP) に変換する。

アルゴリズムアプローチとエンジニアリングトレードオフ.

- 集中型最適ソルバ（純粹割り当てにはハンガリアン、制約付きには商用 MIP）は大域最適割り当てを生成するが、信頼性の高い低遅延リンクと信頼できる中央プランナを必要とする。時間的・組合せ制約に対してスケールしにくい。
- 分散型市場ベースオークションはスケールしやすく、通信損失に耐性があり、局所自律性を持つ人型チームに自然にマッピングされる。局所コスト計算と単純な入札プロトコルを用い、単一障害点リスクを軽減する。
- コンセンサスと分散最適化（ADMM、コンセンサスベースタスク割り当て）は最適性と頑健性をバランスさせるが、同期と有界通信遅延を要求する。

運用上関連のコストモデリング. t_{rt} と e_{rt} を推定するには以下を統合する：

- 現在の状態と環境モデルを用いたモーションプランニング時間。

- アクチュエータ電力モデルと予測歩行相エネルギー。
- 支持多角形に対する投影重心 (CoM) から算出される安定性余裕。要求到達範囲とペイロードを用いて余裕-失敗確率のロジスティック写像として s_{rt} を定量化する。

実装：異種人型向けオークションベース割り当て。以下の分散 Python 例は単一ラウンド第一価格オークションを示す。各ロボットは局所コストを計算し、先行小節で提供されたプロトコル経由で近隣エージェントに bids を送信する。勝者は受諾し、他は撤回する。このパターンは人間が介入する製造床タスクでの迅速な再割当てに適している。

コードサンプル 135 タスク割り当て向け分散第一価格オークション

```
import time
import logging
from typing import Dict, List, Tuple, Any, Optional
import heapq
import threading
from dataclasses import dataclass
from collections import defaultdict

# 入札メッセージ構造
@dataclass(frozen=True)
class Bid:
    robot: str
    task: str
    cost: float
    ts: float = time.time()

# 割当結果
@dataclass
class Assignment:
    task: str
    robot: str
    cost: float

class Auctioneer:
    """
    分散型タスク割当オークションノード。
    同一ロボットが複数回呼ばれることもあるため、スレッドセーフに実装。
    """
    def __init__(self, robot_id: str, comm, cap_check=None, cost_fn=None):
        self.robot_id = robot_id
        self.comm = comm
```

```

self.cap_check = cap_check or (lambda r, t: True)
self.cost_fn = cost_fn or (lambda r, t: 0.0)
self._lock = threading.Lock()
self._bids: Dict[str, List[Bid]] = defaultdict(list)
# task -> [Bid, ...]
self.logger = logging.getLogger(f"Auctioneer[{robot_id}]")

# 1ラウンドの入札・割当
def auction_round(self, tasks: List[str], timeout: float = 0.2) -> List[Assignment]:
    # 自身の入札を送信
    my_bids = []
    for t in tasks:
        if not self.cap_check(self.robot_id, t):
            continue
        c = self.cost_fn(self.robot_id, t)
        bid = Bid(self.robot_id, t, c)
        my_bids.append(bid)
        self.comm.broadcast(bid.__dict__) # 非ブロッキング

    # 受信スレッドで他ロボットの入札を収集
    stop_at = time.time() + timeout
    while time.time() < stop_at:
        msg = self.comm.recv(timeout=stop_at - time.time())
        if msg is None:
            continue
        if msg.get("type") != "bid":
            continue
        b = Bid(msg["robot"], msg["task"], msg["cost"], msg.get("ts", time.time()))
        with self._lock:
            heapq.heappush(self._bids[b.task], b)

    # 各タスクの最低コスト入札を選択
    winners: List[Assignment] = []
    with self._lock:
        for t, bids in self._bids.items():
            if not bids:
                continue
            best = min(bids, key=lambda b: b.cost)
            winners.append(Assignment(t, best.robot, best.cost))
    # 勝者通知

```

```

        self.comm.broadcast({"type": "assign", "task": t, "robot": best.robot})
    self._bids.clear()
    return winners

```

設計上の考慮事項と制約：

1. 遅延とパケット損失：オークションは不整合割当てを防ぐために有界入札ウィンドウを要求する。
2. タスク結合：協調持ち上げには、モーションプランニング前にパートナーを予約する編隊メカニズムが必要。
3. 安全臨界タスク：実行前に割当てを検証する中央検証または冗長チェックが必要。
4. 動的再割当て：バッテリー状態、予期しない障害物、または人間上書きに基づくタスクプリエンプシオン方針を維持する。

性能評価指標. 割当て戦略を以下で評価する：

- スループット（タスク/時）。
- エネルギー効率（消費総ジュール）。
- 安全インシデント（ミッション時あたりの転倒ニアミス回数）。
- 頑健性（通信損失にもかかわらず完了したタスクの割合）。

エンジニアリング含意、トレードオフ、運用上リスク.

- 集中最適性対分散頑健性：ネットワーク信頼性とミッションクリティカリティに基づいて選択する。
- コストモデル忠実度は安全性と効率の両方に影響；過度に楽観的なモデルは転倒リスクを生む。
- セキュリティ：オークションとコンセンサスは、混在ヒューマンロボットワークスペースでスプーフィングを防ぐために入札を認証しなければならない。
- リアルタイム制約：プランナはバランスコントローラの安全でないプリエンプシオンを防ぐために決定遅延を有界にしなければならない。

設計者はミッション制約、利用可能通信、安全余裕に合致する割当て手法を選択すべきである。コストモデルをログアクチュエータ電流と CoM テレメトリに対して検証し、割当て起因の不安定性を削減する。

40 群知能

40.1 スワーム・ヒューマノイド・ロボティクスの基礎

これまでに導入したタスク割り当てと通信プロトコルの概念を踏まえ、本小節ではスワーム知能の原理がヒューマノイド・ロボットにどう適応するかを考察する。焦点は、分散協調、動力学の物理的結合、複数ヒューマノイド展開に必要な安全重視の挙動である。

問題定義： N 台のヒューマノイド・エージェントが局所センシングと限定的通信を用いて協調タスクを遂行できるようにする。主要な運用要件には、安定した歩行協調、衝突のない軌道交渉、大型ペ

イロードの同期操作、通信喪失下でのグレースフル・デグラデーションが含まれる。

技術解析：

- 共有変数のための分散コンセンサス。スカラー量 x （例：形成重心推定値や歩行位相オフセット）を協調するために、離散時間コンセンサス更新は：

$$[H]x_i(k+1) = x_i(k) + \varepsilon \sum_{j \in \mathcal{N}_i} w_{ij}(x_j(k) - x_i(k)), \quad (331)$$

ここで \mathcal{N}_i は隣接ノード、 w_{ij} は正規化重み、 ε は収束のためのステップ幅である。ヒューマノイドでは、アクチュエータ遅延とセンサ更新レートを考慮して ε を保守的に選ぶ。

- 連続時間形成制御と安定性。位置または重心（CoM）軌道を協調する際、各ヒューマノイドの簡略化動力学をプランニング用の一次運動学的積分器としてモデルし、これらの参照を追跡するために全身コントローラを用いる。典型的な分散コントローラはポテンシャルに対する勾配降下を用いる：

$$[H]u_i = -\nabla_{p_i} \left(\sum_{j \neq i} U_{\text{rep}}(\|p_i - p_j\|) + \sum_{k \in \mathcal{G}} U_{\text{attr}}(\|p_i - p_k^*\|) \right), \quad (332)$$

ここで p_i はプランニング用のエージェント i の位置、 U_{rep} は安全のための斥力項、 U_{attr} はチームを目標位置 p_k^* に固定する。

- 衝突回避と運動学的制約。ヒューマノイド・エージェントはバランスを維持しなければならない。階層制御アーキテクチャを実装する：
 1. 上位分散プランナーが安全な CoM と足取り計画を出力。
 2. 中位歩行スケジューラが ZMP（ゼロ・モーメント・ポイント）制約を課す。
 3. 下位全身逆動力学が関節限界とトルク飽和を課す。
- 大規模アンサンブルのための平均場近似。 N が大きい場合、密度ベース制御を用いる。エージェント密度 $\rho(x, t)$ を表現し、連続の式を介して形状制御を行う：

$$[H]\partial_t \rho + \nabla \cdot (\rho v) = 0, \quad (333)$$

ポテンシャルから速度場 $v(x, t)$ を設計し、目標密度 $\rho^*(x)$ に収束させる。

実装ノート（実用的制約と選択）：

- 通信：位相、局所 CoM 推定値、圧縮タスク信頼度のようなコンパクトな状態要約のブロードキャストを優先する。帯域削減のためイベントトリガ通信を用いる。
- センシング：隣接ローカリゼーションのために局所 IMU、脚オドメトリ、短距離 LiDAR を融合する。相対位姿精度は形成の密接さに直接影響する。
- 頑健性：断続的接続のためホールド・アンド・ウェイト挙動を組み込む。各ヒューマノイドはネットワーク入力に失敗しても局所安定性と安全性を自律的に維持すべきである。

アルゴリズムレシピ（歩行位相同期と安全チェックのための簡単な分散コンセンサス）：

1. 各ヒューマノイドがその位相 $\phi_i \in [0, 1)$ と CoM 推定値を公開する。
2. 隣接ノードが位相を受信し、(343) を用いて更新を計算する。
3. 更新された位相が歩行タイミングを変更する；安全レイヤーが足クリアランスや ZMP 限界に違反するステップを先取りする。

実装例：歩行位相に対する隣接ベース・コンセンサスを実装し、安全オーバーライドを課す軽量 ROS2 ノード。ノードは局所位相と CoM を送信し、隣接状態をリッスンする。

コードサンプル 136 ROS2 node for decentralized gait-phase consensus with safety override

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from std_msgs.msg import Float32, Header
from geometry_msgs.msg import Point, PointStamped
from typing import Dict, Tuple
import math
import threading

class GaitConsensusNode(Node):
    def __init__(self) -> None:
        super().__init__('gait_consensus')

        # --- パラメータ ---
        self.declare_parameter('robot_id', 'robot0')
        self.declare_parameter('eps', 0.1)
        self.declare_parameter('safety_radius', 0.5)
        self.declare_parameter('com_frame', 'base_link')
        self.declare_parameter('neighbor_timeout', 0.5)

        self.robot_id: str = self.get_parameter('robot_id').value
        self.eps: float = self.get_parameter('eps').value
        self.safety_radius: float = self.get_parameter('safety_radius').value
        self.com_frame: str = self.get_parameter('com_frame').value
        self.neighbor_timeout: float = self.get_parameter('neighbor_timeout').value

        # --- 状態 ---
        self.phase: float = 0.0 # [0,1)
        self.coM: PointStamped = PointStamped()
        self.coM.header.frame_id = self.com_frame
        self.neighbors: Dict[str, Tuple[float, PointStamped, float]] = {}

# id -> (phase, com, stamp)

        self.lock = threading.Lock()

        # --- QoS ---
```

```

qos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    durability=DurabilityPolicy.VOLATILE,
    depth=10
)

# --- 通信 ---
self.phase_pub = self.create_publisher(Float32, f'{self.robot_id}/gait_phase',
self.com_pub = self.create_publisher(PointStamped, f'{self.robot_id}/com_est',

self.create_subscription(
    Float32, 'gait_phase_in', self.on_phase, qos)
self.create_subscription(
    PointStamped, 'com_est_in', self.on_com, qos)

self.create_timer(0.05, self.step) # 20 Hz
self.create_timer(0.1, self.purge_neighbors) # タイムアウト管理

def on_phase(self, msg: Float32) -> None:
    with self.lock:
        self.neighbors[msg.header.frame_id] = (
            msg.data,
            self.neighbors.get(msg.header.frame_id, (0.0, PointStamped(), 0.0))[1]
            self.get_clock().now().nanoseconds * 1e-9
        )

def on_com(self, msg: PointStamped) -> None:
    with self.lock:
        self.neighbors[msg.header.frame_id] = (
            self.neighbors.get(msg.header.frame_id, (0.0, PointStamped(), 0.0))[0]
            msg,
            self.get_clock().now().nanoseconds * 1e-9
        )

def step(self) -> None:
    now = self.get_clock().now().nanoseconds * 1e-9
    with self.lock:
        # 共有可能隣人のみ抽出
        valid = [(p, com) for rid, (p, com, t) in self.neighbors.items()
                    if now - t < self.neighbor_timeout]

```

```

# 位相コンセンサス
if valid:
    mean_diff = sum((p - self.phase) for p, _ in valid)
    self.phase += self.eps * mean_diff / len(valid)
    self.phase = self.phase % 1.0

# 衝突回避
for _, com in valid:
    if self.too_close(com):
        self.get_logger().warn('Safety_pause')
        return

# 公開
self.phase_pub.publish(Float32(data=float(self.phase)))
self.coM.header.stamp = self.get_clock().now().to_msg()
self.com_pub.publish(self.coM)

def too_close(self, other: PointStamped) -> bool:
    dx = self.coM.point.x - other.point.x
    dy = self.coM.point.y - other.point.y
    dz = self.coM.point.z - other.point.z
    return (dx*dx + dy*dy + dz*dz) < self.safety_radius**2

def purge_neighbors(self) -> None:
    now = self.get_clock().now().nanoseconds * 1e-9
    with self.lock:
        expired = [rid for rid, (_, _, t) in self.neighbors.items()
                    if now - t > self.neighbor_timeout]
        for rid in expired:
            del self.neighbors[rid]

def main(args=None):
    rclpy.init(args=args)
    node = GaitConsensusNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:

```

```
node.destroy_node()
rclpy.shutdown()
```

運用上の意味、トレードオフ、リスク：

- 密接な形成は操作精度を高めるが、衝突リスクを悪化させ、より高いセンシング精度を要求する。
- 分散制御は N に対してスケールし、単一障害点に耐性があるが、通信が疎かまたはアクチュエータ遅延が大きいと収束速度が低下する。
- 安全性はオンボードで実施されなければならない；ネットワーク指令は助言的である。ZMP およびトルク安全マージンをそれに応じて設計する。
- ヒューマノイド間のハードウェアばらつきは、異なる最大加速度を考慮するためコンセンサス則の適応重み w_{ij} を要求する。

設計者はオンボード安定性保証、最小限必須通信、保守的コンセンサス利得を優先すべきである。トレードオフには応答性対頑健性、形成の密接さ対安全バッファ・サイズが含まれる。

40.2 分散制御の実装

前の小節では、スワーム・ロボティクス of 主要なプリミティブ、すなわち局所センシング、局所相互作用ルール、単純なエージェントから生じる創発的行動を確立した。ここでは、これらのプリミティブを、共有物理タスクで動作するヒューマノイド・ロボットのチームのために実装可能な分散制御スタックに変換する。

問題定義と運用上の制約。ヒューマノイド・チームは、中央の司令官なしに組立、物体搬送、または救助タスクを調整しなければならない。各エージェントには限定的な直接センシング範囲、非ゼロ通信遅延、断続的なパケット損失、異種の移動能力がある。分散制御の目的は、バランス、ロボット間安全、リアルタイム応答性を保ちながら、タスクレベルでの収束を保証することである。

技術的分析。分散制御は、相互に作用する 3 つの層に分離される：

1. 近隣発見と状態共有。各ロボットは、姿勢、速度、小さなタスク状態記述子を含むコンパクトな状態メッセージを周期的にブロードキャストする。通信は優先順位付きでレート制限され、帯域幅を保護すべきである。
2. 分散推定と合意。ロボットは、近隣メッセージを局所センサと融合して局所世界モデルを形成する。合意アルゴリズムは、タスク割当重み、形成オフセット、またはグローバルタイミングのような共有変数を整列させる。
3. 安全フィルタ付き局所制御。各ロボットは、移動と操縦のための閉ループ・コントローラを実行する。安全フィルタは、局所最適化またはバリア関数を介して衝突回避と安定性制約を実施する。

グラフ理論的基礎。通信近隣を、隣接行列 A と次数行列 D を持つ無向グラフ $G(V, E)$ としてモデル化する。ラプラシアン L は、合意で使用される拡散ダイナミクスを定義する。スカラー状態 x_i に対する連続時間合意は、エージェント i 上で

$$[H]\dot{x}_i = - \sum_{j \in \mathcal{N}_i} (x_i - x_j) = - \sum_j L_{ij} x_j, \quad (334)$$

であり、実装で使用する離散時間更新は

$$[H]x_i[k+1] = x_i[k] + \epsilon \sum_{j \in \mathcal{N}_i} (x_j[k] - x_i[k]), \quad (335)$$

であり、ステップサイズ ϵ はネットワークスペクトル半径安定性のために選択される。

ヒューマノイド・チームのための設計レシピ。以下のステップを工学詳細とともに実装する：

- コンパクト状態ベクトル：状態を $s = (p, \dot{p}, h)$ として表現し、ここで p は 2 次元位置、 \dot{p} は平面速度、 h は小さなタスク記述子である。メッセージを 200 バイト未満に保つ。
- 近隣ポリシー：センシング半径 r_s 以内、または最終検出時刻ウィンドウ T_{\max} 内の近隣を考慮する。タイムスタンプとシーケンス番号を使用して古いパケットを検出する。
- 合意変数：合意を、割当役割インデックスまたは形成重心のような低レート、低次元変数に制限する。ジョイントエンコーダのような連続高レート変数は局所のままにする。
- 安全層：ロボット間最小距離 d_{\min} を、局所ポテンシャル $U(r) = \kappa(d_{\min} - r)^2$ ($r < d_{\min}$) を使用して実施する。過大な加速度を防ぐために速度バリアを追加する。

参考実装 (Python 風擬似コード)。以下のスニペットは、安全ポテンシャルと非ブロッキング通信を備えた最小 2 次元合意ループを示す。50 Hz 制御ループと状態共有用 UDP ブロードキャストを仮定する。

```
#!/usr/bin/env python3
import math
import time
import socket
import threading
import struct
from dataclasses import dataclass
from typing import List, Tuple, Optional

import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist, PoseStamped
from std_msgs.msg import Header

# ----- パラメータ -----
D_MIN: float = 0.5      # ロボット間最小距離 [m]
K_POT: float = 2.0      # ポテンシャルゲイン
EPS: float = 0.1        # 合意ステップサイズ
LOOP_HZ: float = 50.0   # 制御周波数
UDP_PORT_BASE: int = 50000
MAX_NEIGHBOR_AGE: float = 0.2 # 隣接情報破棄閾値 [s]
```

```

# ----- データ構造 -----
@dataclass
class Neighbor:
    role: float
    pos: Tuple[float, float]
    vel: Tuple[float, float]
    stamp: float # 受信時刻

# ----- ROS 2 ノード -----
class SwarmController(Node):
    def __init__(self, robot_id: int) -> None:
        super().__init__(f'swarm_controller_{robot_id}')
        self.robot_id = robot_id

    # 状態
    self.role: float = 0.0
    self.pos: Tuple[float, float] = (0.0, 0.0)
    self.vel: Tuple[float, float] = (0.0, 0.0)

    # 通信
    self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    self.sock.bind(('', UDP_PORT_BASE + self.robot_id))
    self.sock.setblocking(False)

    # 隣接リスト
    self.neighbors: dict[int, Neighbor] = {}
    self.lock = threading.Lock()

    # ROS インターフェース
    self.cmd_pub = self.create_publisher(Twist, 'cmd_vel', 1)
    self.pose_sub = self.create_subscription(
        PoseStamped, 'pose', self.pose_cb, 1)

    # タイマー
    self.timer = self.create_timer(1.0 / LOOP_HZ, self.control_loop)

    # 受信スレッド
    self.recv_thread = threading.Thread(target=self.recv_loop, daemon=True)
    self.recv_thread.start()

```

```

# ----- コールバック -----
def pose_cb(self, msg: PoseStamped) -> None:
    self.pos = (msg.pose.position.x, msg.pose.position.y)

# ----- 通信 -----
def broadcast_state(self) -> None:
    """非ブロッキング送信: role, pos, vel"""
    data = struct.pack('!Bddd', self.robot_id, self.role,
                       self.pos[0], self.pos[1])
    for i in range(256): # 簡易ブロードキャスト
        addr = ('255.255.255.255', UDP_PORT_BASE + i)
        try:
            self.sock.sendto(data, addr)
        except BlockingIOError:
            pass

def recv_loop(self) -> None:
    while rclpy.ok():
        try:
            data, _ = self.sock.recvfrom(32)
            rid, role, x, y = struct.unpack('!Bddd', data)
            if rid == self.robot_id:
                continue
            with self.lock:
                self.neighbors[rid] = Neighbor(
                    role=role, pos=(x, y), vel=(0.0, 0.0),
                    stamp=time.time())
        except BlockingIOError:
            time.sleep(0.001)

# ----- 制御 -----
def consensus_update(self, neighbors: List[Neighbor]) -> float:
    if not neighbors:
        return self.role
    diff = sum(n.role - self.role for n in neighbors)
    return self.role + EPS * diff / len(neighbors)

def potential_avoidance(self, neighbors: List[Neighbor]) -> Tuple[float, float]:
    vx, vy = 0.0, 0.0

```

```

    for n in neighbors:
        dx = n.pos[0] - self.pos[0]
        dy = n.pos[1] - self.pos[1]
        r = math.hypot(dx, dy)
        if 1e-3 < r < D_MIN:
            grad = K_POT * (D_MIN - r) / r
            vx += grad * dx
            vy += grad * dy
    return (vx, vy)

def compute_task_velocity(self, role: float) -> Tuple[float, float]:
    # 例: 役割に応じた速度を生成
    return (0.5 * math.cos(role), 0.5 * math.sin(role))

def control_loop(self) -> None:
    now = time.time()
    with self.lock:
        # 古い隣接を削除
        self.neighbors = {
            k: v for k, v in self.neighbors.items()
            if now - v.stamp < MAX_NEIGHBOR_AGE}
        neighbors = list(self.neighbors.values())

    # 状態送信
    self.broadcast_state()

    # 役割合意
    self.role = self.consensus_update(neighbors)

    # 速度計算
    rep_v = self.potential_avoidance(neighbors)
    des_v = self.compute_task_velocity(self.role)
    cmd_v = (des_v[0] + rep_v[0], des_v[1] + rep_v[1])

    # 指令送信
    twist = Twist()
    twist.linear.x, twist.linear.y = cmd_v
    self.cmd_pub.publish(twist)

# ----- メイン -----

```

```
def main(args=None):
    rclpy.init(args=args)
    node = SwarmController(robot_id=42) # IDは起動時引数で変更可
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

ヒューマノイド歩容および操縦との統合。合意出力を直接使用してジョイント軌道を指令しないでください。代わりに、合意および安全出力を高レベルの足跡ターゲット、重心（CoM）速度参照、または手ターゲットオフセットにマッピングする。ゼロ・モーメント・ポイント（ZMP）安定性およびジョイント限界を実施する、局所で実行される全身コントローラまたはモデル予測コントローラを使用する。

フォールト・トレランスと推定。近隣リストでタイムアウトと外れ値除去を使用する。位置不確実性については、局所オドメトリと近隣相対観測を組合せた分散推定器を実行する。エージェントが互いに光学的に観測する場合、合意ベースの分散カルマン・フィルタがドリフトを低減できる。

工学への影響とトレードオフ。

- ・帯域幅対収束：メッセージサイズまたは周波数を増やすと収束速度が向上するが、パケット損失と遅延リスクが増加する。
- ・安全対性能：積極的なポテンシャル利得は、衝突回避を改善するが、タスク効率と振動のコストがかかる。
- ・集中チェックポイント：時折の集中集約はグローバルタスクを加速できるが、単一障害点を再導入する。
- ・運用上のリスク：近隣発見の不良または敵対的メッセージは、安全でない集団行動を引き起こす可能性がある；認証と堅牢なフォールト処理が必須である。

設計者は、合意レート、メッセージコンパクト性、および局所コントローラ堅牢性をバランスさせなければならない。ハードウェア試験の前に、高忠実度シミュレーションで検証する。

40.3 ケーススタディ：複数ヒューマノイドによる組立てタスク

これまでに説明した自律分散制御パターンとスワームの基礎を踏まえ、本ケーススタディではそれらの原理を現実的な複数ヒューマノイドによる組立てシナリオに適用する。この例は、自律分散コンセンサス、分散タスク割当、および制約付き全身制御が、工場およびサービス環境に適した信頼性の高い組立てパイプラインにどのように統合されるかを示す。

問題設定. 4 体のヒューマノイドエージェントが協調してコンベア化された治具上に 5 モジュールからなるペイロードを組立てる。タスクにはモジュールの取り出し、位置合わせ、協調リフティン

グ、精密挿入、および工具の受け渡しが含まれる。各ヒューマノイドは全身アクチュエータ、ステレオヘッド、手首の力・トルクセンシング、および無線 V2V リンクを備える。設計目標は：サイクルタイムを最小化し、バランスと安全マージンを維持し、単一エージェント故障に対してセルを停止させないことである。

技術解析. 各ヒューマノイドを局所状態 x_i を持つエージェント i としてモデルし、 x_i は関節構成 q_i 、デカルトエンドエフェクタ姿勢 p_i 、および力読み取り値 f_i を含む。知覚はノイズが存在し、エージェントはターゲットモジュールの局所姿勢推定値 p_i^m を維持する。共有姿勢とタイミングで合意に達するために、離散時間コンセンサス更新を用いる：

$$[H] p_i^m(k+1) = p_i^m(k) + \alpha \sum_{j \in \mathcal{N}_i} (p_j^m(k) - p_i^m(k)) \quad (336)$$

ここで $\alpha \in (0, 1)$, \mathcal{N}_i はエージェント i の隣接集合である。これは連結通信グラフ下で指数収束を保証する。

協調リフティングおよびレンチ配分のために、力共有を二次計画問題として定式化する。 $A = [J_1^T \ J_2^T \ \dots]^T$ を各エージェント接触レンチ f を合成レンチに写像させるとする。次を解く

$$\begin{aligned} \min_f \quad & \frac{1}{2} f^T W^{-1} f \\ [H] \text{ s.t. } \quad & A f = F_{\text{des}} \\ & f_{\min} \leq f \leq f_{\max} \end{aligned} \quad (337)$$

ここで W はアクチュエータ限界と疲労を符号化する対角重み付き行列、 F_{des} は所望の物体レンチである。制約なし解は解析的に有用な形式 $f^* = W A^T (A W A^T)^{-1} F_{\text{des}}$ を与え、制約は QP ソルバで処理される。

衝突回避とバランスは、各エージェントの全身 QP における不等式制約として実施される。関節速度 v に対するコンパクトな各エージェント QP は次の通り：

$$\begin{aligned} \min_v \quad & \|Jv - v_{\text{task}}\|^2 + \lambda \|v\|^2 \\ [H] \text{ s.t. } \quad & C_{\text{col}}(q)v \leq b_{\text{col}} \quad (\text{衝突}) \\ & C_{\text{zmp}}(q)v \leq b_{\text{zmp}} \quad (\text{安定性}) \end{aligned} \quad (338)$$

ここで J はタスクヤコビアン、 v_{task} は所望デカルト速度、 C_{zmp} は協調運動中にヒューマノイドを安定に保つゼロモーメントポイント制約を符号化する。

自律分散タスク割当. 動的ロールに対する軽量分散オークションを用いてサブタスク（ピック、ホールド、アライン、インサート）を割り当てる。各エージェントは、空き手の有無、近接性、推定完了時間、バッテリー状態を組み合わせた局所コスト指標を用いて入札する。オークションは隣接エージェント間で局所的に実行され、グローバル同期を回避するためタイムアウトを設ける。エージェントが応答しなければ、その入札は失効し、隣接エージェントがタスクを再配分する。

実装概要と検証. Isaac Sim で複数ヒューマノイドシステムを実装し、衝突、把握運動学、ケーブルルーティングを検証する。メッセージングには ROS2 を用い、各エージェントに小規模なコンセンサス／調停ノードを配置する。次の Python スニペットは、ROS2 ライク環境での最小限の分散オークションハートビートと入札ブロードキャストを実装する。ノンブロッキング I/O と単純な入札失効を重視する。

コードサンプル 137 分散オークションハートビートと入札ブロードキャスト（簡略化）

```
import asyncio
import json
import logging
import time
from collections import defaultdict
from dataclasses import dataclass, field
from typing import Dict, List, Optional

# モック通信API（実環境では差し替え）
async def send_broadcast(payload: str) -> None:
    """非公開の通信層へ転送（本番ではDDS／ROS2等に置換）"""
    pass

async def recv_broadcast() -> str:
    """非公開の通信層から受信（本番ではDDS／ROS2等に置換）"""
    pass

@dataclass
class Config:
    agent_id: str = "humanoid_A"
    bid_ttl: float = 1.5
    heartbeat: float = 0.2
    max_queue_size: int = 1000

@dataclass
class Bid:
    agent: str
    cost: float
    ts: float

@dataclass
class TaskBoard:
    queues: Dict[str, List[Bid]] = field(default_factory=lambda: defaultdict(list))

    def prune(self, ttl: float) -> None:
```

```

        now = time.time()
        for bids in self.queues.values():
            bids[:] = [b for b in bids if now - b.ts < ttl]

def winner(self, task: str) -> Optional[str]:
    bids = self.queues.get(task, [])
    if not bids:
        return None
    return min(bids, key=lambda b: b.cost).agent

class AuctionAgent:
    def __init__(self, cfg: Config) -> None:
        self.cfg = cfg
        self.board = TaskBoard()
        self.logger = logging.getLogger(self.cfg.agent_id)

    async def send_bid(self, task_id: str, cost: float) -> None:
        msg = {
            "type": "bid",
            "agent": self.cfg.agent_id,
            "task": task_id,
            "cost": cost,
            "t": time.time(),
        }
        await send_broadcast(json.dumps(msg))

    async def _handle_message(self, raw: str) -> None:
        try:
            msg = json.loads(raw)
        except json.JSONDecodeError:
            self.logger.warning("受信データがJSONではありません")
            return

        if msg.get("type") == "bid":
            task = msg.get("task")
            if task is None:
                return
            self.board.queues[task].append(
                Bid(agent=msg["agent"], cost=msg["cost"], ts=msg["t"])
            )

```

```

    )
    # メモリ保護
    if len(self.board.queues[task]) > self.cfg.max_queue_size:
        self.board.queues[task].pop(0)

async def listen_loop(self) -> None:
    while True:
        raw = await recv_broadcast()
        await self._handle_message(raw)

async def auction_loop(self) -> None:
    while True:
        self.board.prune(self.cfg.bid_ttl)

        for task in list(self.board.queues.keys()):
            winner = self.board.winner(task)
            if winner is None:
                continue
            await send_broadcast(
                json.dumps({"type": "award", "task": task, "agent": winner})
            )
            del self.board.queues[task]

        await asyncio.sleep(self.cfg.heartbeat)

async def run(self) -> None:
    await asyncio.gather(self.listen_loop(), self.auction_loop())

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    cfg = Config()
    agent = AuctionAgent(cfg)
    try:
        asyncio.run(agent.run())
    except KeyboardInterrupt:
        pass

```

テスト計画. 通信劣化およびセンサノイズ下でハードウェアインザループ試験を実施し, 安全マージンを検証する. シミュレーションでランダム初期姿勢を用い, 故障モードを記録する. 単一エージェント故障をエミュレートし, タスクの再割当がサイクルタイム制約内で完了することを確認

する。

運用への影響とトレードオフ。

- ・通信：分散オークションとコンセンサスはグローバルサイズではなく局所次数でスケールする。遅延が 100 ms を超えると精密ハンドオフの調整が劣化する。
- ・安全性：全身 QP は ZMP および衝突制約を実施するが、予期せぬ接触にはコンプライアンス制御と高速フォールバック動作が必要である。
- ・頑健性：重み行列 W はアクチュエータ負荷分散とエネルギー効率をトレードオフする。一方のエージェントへの重い重み付けは摩耗リスクを増大させる。
- ・検証：シミュレーションはセンサ遅延、コンプライアンス、構造たわみを含めて現実の組立て故障を明らかにする必要がある。
- ・リスク：複数エージェントが同一把握を競合するとデッドロックが発生する可能性がある；ランダムバックオフとロールタイムアウトを実装する。

技術者は初期展開において、冗長知覚、有界通信遅延、および保守的力制限を優先すべきである。これらの設計選択は、複数ヒューマノイド協調組立て中の故障連鎖を低減する。

41 実世界での応用

41.1 製造業における協働ロボット

これまでに議論したタスク割当戦略と分散協調パターンを基に、本小節では人間と同一作業空間で協働するヒューマノイド型協働ロボットの具体的な製造現場での導入事例を検討する。焦点は実践的な課題、制御上の制約、センシング要件、およびヒューマノイド・コボットを安全かつ効率的に導入するためにエンジニアが対処しなければならないトレードオフである。

問題定義：組立セルにヒューマノイド・コボットを統合し、人間が検査や巧緻性を要する作業を行う間、部品搬送および軽組立を実行する。目的はスループットを最大化し、安全な相互作用を確保し、人間パートナーの人間工学的快適性を維持することである。制約にはペイロード限界、安全接触力限界 (ISO/TS 15066)、知覚遅延、および人間の動作によるサイクルタイムの変動が含まれる。

技術分析：

- ・タスク分解およびスループットモデル。原子タスクのうちロボットが自律的に実行する割合を p とする。繰返しタスクの単純なサイクルタイムモデルは

$$[H]C(p) = pT_r + (1-p)T_h + H(p), \quad (339)$$

であり、 T_r と T_h はそれぞれ純ロボットおよび純人間の完了時間、 $H(p)$ は手渡しや同期遅延を含む協調オーバーヘッドをモデル化する。二次オーバーヘッド $H(p) = \alpha p(1-p)$ はバランスの取れた分担近傍での同期増加を捉える。 $C(p)$ を p について最小化すると閉形式の候補

$$[H]p^* = \text{proj}_{[0,1]} \left(\frac{1}{2} \left(1 + \frac{T_h - T_r}{\alpha} \right) \right), \quad (340)$$

が得られ、後述する安全駆動速度スケーリングおよび接触限界を受ける。この解析モデルは実センサデータによる詳細最適化の前のベースライン割当を導く。

- 安全性および接触力学. ISO/TS 15066 は異なる身体部位ごとの最大許容力を規定する. 手渡しシナリオでは, エンジニアは最大接触力 F_{\max} と相対接近速度 v_{rel} を課す. コンプライアント制御則を用いることで衝撃力のピークを低減する. アドミタンス制御は人間-ロボット手渡しに有効であり, 一般的に用いられるモデルは

$$[H]M_d\dot{v} + B_d v + K_d x = F_{\text{ext}}, \quad (341)$$

である. ここで M_d , B_d , K_d は設計行列, x は相対変位, v は指令速度, F_{ext} は測定外力である. 適切なチューニングは接近時に低剛性を確保し, 手渡し後は物体を安定化するために剛性を増加させる.

- 知覚遅延および決定閾値. 手渡しには頑健な把持姿勢推定および手検出が必要である. 知覚遅延を τ_p , 姿勢不確実性を σ_p とする. モーションプランナは, 衝突体積を膨張させ接近速度を制限することで不確実性を組み込まなければならない:

$$[H]v_{\max} = \min\left(v_{\text{nominal}}, \frac{F_{\max}}{B_{\text{imp}} + k \sigma_p}\right), \quad (342)$$

ここで B_{imp} は衝撃減衰, k は不確実性をリスクにスケーリングする. この知覚誤差と許容速度の明示的な結合はセンサ劣化下でも安全動作を強制する.

実装ブループリント:

1. センシングおよび状態推定

- ハンドポーズ用深度カメラまたは手首搭載ステレオ; 胴体およびエンドエフェクタポーズ用 IMU 融合.
- 到達/手渡しを予測するリアルタイム人間意図分類器, 信頼度スコア $c(t)$ 付き.

2. 制御スタック

- 主要モーションプランナは運動学および動的限界を考慮した公称軌道を生成.
- 安全層は式 (3) および (1) を用いて速度および分離制約を強制.
- アドミタンスコントローラ (式 (2)) は最終接近およびコンプライアント手渡しを処理.

3. タスク割当

- 式 (1) および (2) を用いて p を初期化. 測定された T_r , T_h , およびオーバーヘッド統計を移動ウィンドウで用いてオンラインで p を更新.

最小動作例: 力センシングに基づいてエンドエフェクタ速度を指令する ROS 風アドミタンスコントローラループ. 本スニペットはコア更新を示す; 本番では実ロボットミドルウェアおよび安全モニタと統合する.

コードサンプル 138 手渡し用単純アドミタンスコントローラループ, デモ用.

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from geometry_msgs.msg import TwistStamped
from std_msgs.msg import Float64
```

```

import numpy as np
import threading
import time

class AdmittanceController(Node):
    def __init__(self):
        super().__init__('admittance_controller')

        # 仮想バネ・ダンパ・質量パラメータ
        self.M = 0.5
        self.B = 50.0
        self.K = 200.0
        self.dt = 0.01

        # 内部状態
        self.v = 0.0
        self.x = 0.0
        self.F_ext = 0.0
        self.pose_uncertainty = 0.0

        # 安全定数
        self.v_nominal = 0.4
        self.F_MAX = 20.0 # 許容最大外力

        # QoS: センサデータはベストエフォートで最新のみ
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        # 外力購読
        self.force_sub = self.create_subscription(
            Float64, 'wrist_force', self.force_cb, qos)

        # 姿勢不確かさ購読
        self.uncertainty_sub = self.create_subscription(
            Float64, 'pose_uncertainty', self.uncertainty_cb, qos)

        # 速度指令出版

```

```

self.vel_pub = self.create_publisher(TwistStamped, 'cmd_vel', 10)

# 周期タイマ
self.timer = self.create_timer(self.dt, self.control_cycle)

# スレッドセーフ用ロック
self.lock = threading.Lock()

def force_cb(self, msg: Float64):
    with self.lock:
        self.F_ext = msg.data

def uncertainty_cb(self, msg: Float64):
    with self.lock:
        self.pose_uncertainty = msg.data

def safety_check(self, force: float, pose_unc: float) -> float:
    # 速度上限を外力と不確かさで調整
    vmax = min(self.v_nominal,
               max(0.05, self.F_MAX / (self.B + 10.0 * pose_unc)))
    return vmax

def control_cycle(self):
    with self.lock:
        F = self.F_ext
        unc = self.pose_uncertainty
        vmax = self.safety_check(F, unc)

        # 離散アドミタンス更新 (Euler積分)
        v_dot = (F - self.K * self.x - self.B * self.v) / self.M
        self.v += v_dot * self.dt
        self.v = np.clip(self.v, -vmax, vmax)
        self.x += self.v * self.dt

        # 速度指令メッセージ生成
        twist = TwistStamped()
        twist.header.stamp = self.get_clock().now().to_msg()
        twist.twist.linear.x = self.v
        self.vel_pub.publish(twist)

```

```
def main(args=None):
    rclpy.init(args=args)
    node = AdmittanceController()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

設計トレードオフおよび運用上のリスク：

- ・スループット対安全： p （ロボット分担）を増加させると、 T_r が短縮されスループット向上が期待されるが、人間との接触頻度が高まり安全リスクが増大する．逆に p を減少させれば安全性は高まるが、人間の負担増によりスループットが低下し、長期的な人間工学的負荷が懸念される．

41.2 災害対応ロボティクス

製造の例に続き、協調型ヒューマノイドは、はるかに制約の少ない災害環境で同様の通信およびタスク割り当てプリミティブを適用する。次の議論では、これらのプリミティブを探索・救助任務に拡張し、部分的な通信、センサの不確実性、不安定な地形に対する堅牢性を強調する。

災害対応の問題定義と運用上の制約：

- ・目的：崩壊した構造物シナリオで生存者を特定、トリアージ、救出し、人間のチームへの追加リスクを最小限に抑える。
- ・主な制約：断続的な通信、視覚および LiDAR の遮蔽、不均一で変形可能な足場、限られたバッテリーおよび操縦ペイロード、高リスクな救出に対するヒューマン・イン・ザ・ループの監督要件。
- ・パフォーマンス指標：最初の検出までの時間、検索領域内での検出確率、タスクごとの平均消費エネルギー、安全な介入率。

技術分析：分散知覚とタスク割り当て

- ・知覚：ローカル SLAM を共有占有および生存者確率グリッドと融合する。各ロボットは生存者存在に対するローカル事後確率 $p_v(x)$ を維持する。ロボットは、帯域幅が許すときに圧縮されたマップ記述子と高確率領域ハッシュを交換する。
- ・タスク割り当て：効用ベースのオークションを使用し、ロボット j は効用 $U_{jk} = \alpha I_{jk} - \beta E_{jk} - \gamma R_{jk}$ を用いてタスク k に入札する。ここで：
 - I_{jk} は情報利得（領域にわたる p_v のエントロピーの期待削減）、
 - E_{jk} はタスクに到達し実行するためのエネルギー・コスト、

- R_{jk} は推定リスク（構造的不安定、危険への曝露）、
- 係数 α, β, γ は任務優先度（救出優先対エネルギー保存）を重み付ける。
- 効用をソフトマックスを介して確率的入札に変換し、探索の多様性を保持する：

$$P_{jk} = \frac{\exp(U_{jk}/\tau)}{\sum_m \exp(U_{jm}/\tau)}$$
、ここで温度 τ は活用と探索を制御する。
- 検索カバレッジと期待検出時間：検索レート $r(x)$ （センサ走査レート）と事前確率 $p_v(x)$ が与えられた場合、瞬間的な期待検出時間は $E[T] \approx \int \frac{p_v(x)}{r(x)} dx$ と近似され、これは p_v が高く現在の $r(x)$ が低い領域にロボットを割り当てることを促す。

実装アーキテクチャとアルゴリズム

1. ヒューマノイドごとのローカルモジュール：
 - 瓦礫および変形可能表面のための足場選択を備えた反応的歩行およびバランスコントロール。
 - 知覚スタック：マルチモーダルセンサ融合（ステレオまたは深度カメラ、LiDAR IFR、サーモカメラ、IMU）。
 - ローカルマッピング：占有+生存者確率グリッド；パーティクルフィルタまたは EKF ベースの姿勢推定。
 - ビヘイビアマネージャ：オークション、遠隔操作へのフェイルオーバー、救出ルーチン。
2. チームレベルコーディネーション：
 - アドホックメッシュ上の分散オークションプロトコル。各ロボットは定期的に圧縮入札およびマップ要約をブロードキャストする。
 - 生存者位置の合意は信頼度重み付き融合を使用；高信頼度は人間への警告をトリガする。
 - フォールバックモード：通信が閾値を下回ると、ロボットは安全および帰還行動を重視したローカルポリシーで動作する。
3. モーションプランニング：
 - コストマップは動的安定性マージンを含む。プランナは複合コスト：経路長+安定性ペナルティ+操縦準備度を最小化する。
 - ステッピングおよびアームモーションのためのキノダイナミックプランニングを使用し、操縦中のバランス制御のためのインナーループモデル予測コントローラ（MPC）を用いる。

実用的アルゴリズムスニペット

- 以下の Python 例は、コンパクトなオークション入札ルーチンを実装する。簡略化された確率グリッドから情報利得を計算し、経路ヒューリスティックによってエネルギーを推定し、チームチャンネルに入札を投稿する。

コードサンプル 139 救出タスクのための分散オークション入札（ROS2 ライクな疑似コード）

```
import numpy as np
from typing import Tuple, Sequence

# センサモデル定数
```

```

_SENSOR_GAMMA: float = 0.5          # 観測後のエントロピー減衰率
_NOMINAL_SPEED: float = 0.5         # [m/s]
_EPS: float = 1e-9                 # 対数発散防止クリップ値

```

```

def info_gain(grid_prob: np.ndarray, region_idx: Tuple[int, int]) -> float:
    """

```

```

        指定領域の観測による情報利得をシャノンエントロピー差分で算出
    """

```

```

        p = grid_prob[region_idx]
        # エントロピー計算 (0 log 0 を回避)
        H_before = -np.sum(p * np.log(np.clip(p, _EPS, 1.0)))
        H_after = H_before * (1.0 - _SENSOR_GAMMA)
        return H_before - H_after

```

```

def energy_cost(robot_pose: Sequence[float], region_centroid: Sequence[float]) -> float:
    """

```

```

        ロボット位置から領域重心までの移動時間見積もり
    """

```

```

        d = np.linalg.norm(
            np.array(robot_pose[:2], dtype=float) -
            np.array(region_centroid[:2], dtype=float)
        )
        return d / _NOMINAL_SPEED

```

```

def estimate_risk(region_centroid: Sequence[float]) -> float:
    """

```

```

        ドメイン固有のリスク推定関数 (ダミー実装)
    """

```

```

        # TODO: 実環境に応じて置換
        return 0.0

```

```

def compute_bid(
    robot_pose: Sequence[float],
    grid_prob: np.ndarray,
    region_idx: Tuple[int, int],
    region_centroid: Sequence[float],

```

```

    alpha: float = 1.0,
    beta: float = 0.7,
    gamma: float = 1.2
) -> float:
    """
    """
    単一口ボットが領域に対する効用 (bid) を算出
    """
    I = info_gain(grid_prob, region_idx)
    E = energy_cost(robot_pose, region_centroid)
    R = estimate_risk(region_centroid)
    return alpha * I - beta * E - gamma * R

```

エンジニアリングへの影響、トレードオフ、運用上のリスク

- ・通信対自律：中央コーディネーションへの依存が高いほど効率は向上するが、単一障害点が生じる。分散オークションはこのリスクを低減するが、重複カバレッジのコストがかかる。
- ・センシングのトレードオフ：サーモカメラは遮蔽下での検出を改善するが、消費電力と誤報率を増加させる。重量および電力バジェットがヒューマノイドのセンサスイートを制約する。
- ・モーション対操縦の結合：不安定な瓦礫の積極的な操縦は転倒リスクを高める。救出中は保守的なバランスマージンを優先する。
- ・ヒューマンオーバーサイト：高リスクな救出決定には常にヒューマン・イン・ザ・ループを維持する；自動検出は検証されるまで助言的であるべきだ。
- ・安全性と認証：グレースフルデグラデーションおよびセーフフェイルモードを設計する。規制遵守およびロギング要件は計算およびストレージオーバーヘッドを追加する。

具体的な運用指針：

- ・任務優先度に応じて α, β, γ を調整し、代表的な瓦礫でのハードウェア・イン・ザ・ループテストを実施してリスクモデル R_{jk} を較正する。
- ・マップ圧縮（例：領域ハッシュ）を使用して、高負荷任務中の帯域幅使用を制限する。
- ・通信または姿勢不確実性が閾値を超えた場合の安全な帰還のための予備バッテリーおよびポリシーを維持する。

これらの設計選択は、ハイステークスな救助任務における検出速度、エネルギー消費、チーム生存性を制御する。

41.3 協働型ヒューマノイドロボットの未来

災害対応や製造の例を踏まえ、協働型ヒューマノイドチームの未来は、人間との極めて高い適応性と安全な共存の両立を強調する。非構造化環境や産業スループットに関する教訓は、知覚、協調、そして堅牢な制御の要件を形作る。

問題定義：ドメインを横断して堅牢に動作する協働型ヒューマノイドシステムを設計せよ。これらのシステムは、ヒューマノイドや他のロボット間でタスクを割り当て、動的なフォーメーションを維持し、人間と安全に対話しなければならない。主な制約として、オンボード計算資源の限界、通信遅

延のばらつき、安全認証済みの力相互作用が挙げられる。

技術分析

- 機能分解：

1. 知覚と意図推定：環境状態と人間の意図予測のためのリアルタイムセンサ融合。
2. マルチエージェント協調：通信制約下でのタスク割当と運動協調のためのスケーラブルなアルゴリズム。
3. 安全考慮制御：コンプライアントなインピーダンス制御と衝突回避の形式的検証。
4. システムレベル耐性：センサ損失、アクチュエータ故障、ネットワーク分断に対するグレースフルデグラデーション。

- マルチエージェントコンセンサスとフォーメーション制御非集中型コンセンサスは協調の基礎的プリミティブである。状態 $x_i \in \mathbb{R}^n$ を持つ N 台のロボットネットワークに対して、離散時間コンセンサス更新は

$$[H]x_i(k+1) = x_i(k) + \varepsilon \sum_{j=1}^N a_{ij}(x_j(k) - x_i(k)), \quad (343)$$

で与えられる。ここで $A = [a_{ij}]$ は隣接行列、 $\varepsilon > 0$ はステップサイズである。平均への収束には $\varepsilon < 1/d_{\max}$ が必要で、 d_{\max} は最大行和次数である。収束速度はグラフラプラシアン L の代数結合度 $\lambda_2(L)$ によって支配され、誤差減衰は概ね $(1 - \varepsilon \lambda_2)^k$ に比例する。

設計上の示唆：ネットワークトポロジーにより λ_2 を向上させ、またはリレーノードを追加して合意を加速するが、通信インフラの追加コストが生じる。

- 制約下でのタスク割当実タスクには時間的要求、ペイロード制約、安全マージンがある。割当を制約付き最適化として定式化する：

$$[H] \min_{y_{ij} \in \{0,1\}} \sum_{i=1}^M \sum_{j=1}^N c_{ij} y_{ij} \quad \text{s.t.} \quad \sum_j y_{ij} = 1, \sum_i w_i y_{ij} \leq W_j, \quad (344)$$

ここで c_{ij} はタスク i をロボット j に割り当てるコスト、 w_i はタスク負荷、 W_j はロボット容量である。リアルタイム動作には、集中型整数計画ではなく分散型オークションベースのヒューリスティクスを用いる。

実装パターン

- 知覚スタック：深度カメラと IMU が状態推定器に入力するモジュラーパイプラインを実行する。学習済み意図モデルを用いて人間の軌道を予測する。安全クリティカルパイプラインをリアルタイムコアに優先させる。
- 制御アーキテクチャ：接触タスクのためのインピーダンス制御と、歩行・障害物回避のための高レベルモデル予測制御（MPC）を組み合わせる。MPC は簡略化ダイナミクスを用いてヒューマノイドコントローラ上で計算処理可能に保つ。
- ミドルウェアとシミュレーション：Isaac Sim のような物理精度の高いシミュレータでマルチヒューマノイドシナリオを検証する。ROS2 を非集中型ディスカバリーに、GROOT ビヘイビアツリーを高レベルタスクシーケンシングに用いる。

代表的コード

コードサンプル 140 ヒューマノイドポーズ平均化のための簡単な非集中型コンセンサスループ

```
import numpy as np
import time
import logging
from typing import List, Tuple

# ログ設定
logging.basicConfig(level=logging.INFO, format='%(asctime)s-%(levelname)s-%(message)s')

class ConsensusController:
    """
    マルチエージェントの合意制御 (2D Consensus)
    """
    def __init__(self, adjacency_matrix: np.ndarray, initial_poses: np.ndarray, epsilon: float):
        """
        adjacency_matrix: 隣接行列 (N×N)
        initial_poses: 初期位置 (N×2)
        epsilon: ステップサイズ (1/d_max未満)
        """
        self.A = adjacency_matrix.astype(float)
        self.poses = initial_poses.astype(float)
        self.eps = epsilon
        self.N = self.A.shape[0]

        # ステップサイズの妥当性チェック
        max_degree = np.max(np.sum(self.A, axis=1))
        if self.eps >= 1.0 / max_degree:
            logging.warning(f"ε={self.eps}は理論的上限{1.0/max_degree:.3f}を超えています")

    def neighbor_states(self, i: int) -> List[np.ndarray]:
        """
        エージェントiの隣接エージェントの位置を返す
        """
        return [self.poses[j] for j in range(self.N) if self.A[i, j] > 0]

    def step(self) -> None:
        """
        1ステップ分の合意更新
        """
        new_poses = self.poses.copy()
        for i in range(self.N):
            nbrs = self.neighbor_states(i)
            if not nbrs:
```

```

        continue
        mean_nbr = np.mean(nbrs, axis=0)
        new_poses[i] = self.poses[i] + self.eps * (mean_nbr - self.poses[i])
    self.poses[:] = new_poses

    def run(self, max_steps: int = 1000, sleep_sec: float = 0.01, tol: float = 1e-4) -
        """
        〇〇〇〇〇〇〇〇 合意アルゴリズムを実行
        〇〇〇〇〇〇〇〇 max_steps: 〇 最大ステップ数
        〇〇〇〇〇〇〇〇 sleep_sec: 〇 通信遅延シミュレーション
        〇〇〇〇〇〇〇〇 tol: 〇 収束判定閾値
        〇〇〇〇〇〇〇〇 戻り値: 〇 (最終位置, 〇 収束ステップ)
        〇〇〇〇〇〇〇〇 """
        for k in range(max_steps):
            prev = self.poses.copy()
            self.step()
            time.sleep(sleep_sec)
            # 収束判定: 全エージェントの変化量が tol 未満
            if np.max(np.linalg.norm(self.poses - prev, axis=1)) < tol:
                logging.info(f"収束しました 〇 (step={k}) ")
                return self.poses, k
            logging.warning("最大ステップに到達しました")
        return self.poses, max_steps

if __name__ == "__main__":
    # 隣接行列 (完全グラフ)
    A = np.array([[0, 1, 1],
                  [1, 0, 1],
                  [1, 1, 0]])

    # 初期位置
    poses = np.array([[0.0, 0.0],
                      [1.0, 0.0],
                      [0.5, 0.8]])

    eps = 0.1

    controller = ConsensusController(A, poses, eps)
    final_poses, steps = controller.run(max_steps=1000, sleep_sec=0.01)
    logging.info(f"最終位置: \n{final_poses}")

```

このリストは、フォーメーション維持や分散推定に適した最小限の非集中型更新を示す。実際には、

パケット損失処理、タイムスタンプ、状態予測を更新に追加して遅延に耐える。

実用的トレードオフと工学制約

- 集中型対非集中型制御：
 - 集中型コントローラは大域最適化を簡素化するが、単一障害点を生じる。
 - 非集中型アルゴリズムはスケールし分断に耐えるが、収束が遅い。
- 計算対通信：
 - 重い最適化をエッジサーバにオフロードするとオンボード負荷は減るが、ネットワーク信頼性への依存が増す。
- 安全マージン対タスク効率：
 - 安全マージンを大きくすると（分離距離増加、接触力低減）、スループットが減る。
 - 厳格な安全エンベロップはセンシングと制御要求を増やす。

運用上のリスクと緩和策

- 通信障害：ネットワーク喪失下でミッション崩壊を回避するため、最終的整合性とローカル自律性を設計する。
- 人間の信頼と予測不能性：意図予測に明示的拒否モードを統合し、人間が安全にロボット動作を上書きできるようにする。
- 敵対的入力：知覚パイプラインをスプーフィングに対して堅牢化し、クロスセンサ検証を実装する。

具体的工学示唆

- リレーノードを追加するか物理レイアウトを変更して代数結合度 λ_2 を高め、協調遅延を最適化する。
- 観測された d_{\max} に基づいて式 343 に従いステップサイズ ε を選択し、可変トポロジー下で収束を保証する。
- リアルタイム動作にはオークションベース分散割当を優先し、計算集約的で非時間クリティカルなフェーズに集中型ソルバーを確保する。

設計上のトレードオフ要約

- 冗長性を高めて耐性を向上させるが、重量と消費電力が増加する。
- エッジに計算を移してオンボードサイクルを節約するが、ネットワーク堅牢性への依存を生じる。
- 保守的な安全エンベロップを用いて認証を簡素化するが、タスクスループットの低下を受け入れる。

実世界におけるロボティクス応用

42 医療におけるロボット

42.1 高齢者ケアのための支援ロボット

前述のヒューマノイド構造とヒューマン・ロボット・インタラクションの基礎を踏まえ、本項では高齢者ケアに実用的なヒューマノイド支援ロボットを実現するための工学原理に焦点を当てる。運用上の課題を定式化し、主要な技術要件を導出し、長期間の在宅展開に適した実装パターンを提示する。問題定義と運用上の制約

- ・ ミッション：ユーザの尊厳とプライバシーを保ちながら、日常生活動作（ADL）の安全かつ信頼性の高い支援、移動支援、服薬リマインダ、社会的エンゲージメントを実現する。
- ・ 制約：非構造化された家庭環境、人間の歩容・行動のばらつき、限られた電力予算、ネットワークの信頼性の低さ、厳格な安全・規制要件。
- ・ 性能指標：タスク成功率、アシストまでの時間、転倒検知の誤陽性率、支援動作あたりのエネルギー、ユーザ満足度。

技術分析設計は五つの結合したサブシステムに分解される：メカニカルプラットフォーム、知覚、制御、行動管理、プライバシー／セキュリティ。各サブシステムは他に影響し、共設計が必要である。

1. メカニカルおよび駆動設計

- ・ コンプライアンスと軽量リムは偶発的接触時の衝撃を低減する。安全で予測可能なインタラクションのため、直列弾性アクチュエータまたはトルク制御ブラシレスモータを用いる。
- ・ 低重心と受動的足首関節は、家庭環境で一般的な側方擾乱に対する立位安定性を向上させる。
- ・ マニピュレータ到達距離とグリップ設計は家庭内物体に最適化する：広開口、触覚パッド、力制限把持により錠剤、カップ、衣類に対応。

2. 知覚とセンシング

- ・ マルチモーダルセンシングは頑健性を向上させる：
 - ビジョン（RGB＋深度）による物体・姿勢推定。
 - ウェアラブル IMU またはベッドセンサによる冗長転倒検知。
 - マイクロホンアレイによる音声・音源定位。
- ・ アルゴリズムはエッジハードウェア上で動作し、クラウドサービスが利用できない場合にも優雅に劣化しなければならない。

3. 運動制御と物理的インタラクション

- ・ 腕による歩行誘導またはハンドオーバーなどのインタラクションタスクにはインピーダンス制御を用いる。
- ・ バランス・歩行は ZMP（ゼロモーメントポイント）プランナと雑然とした床面での反動的足場選択を統合する。
- ・ 安全エンベロープは、人間までの最小距離と関節トルクの動的制限をリアルタイム監視することで強制される。

4. 行動管理とパーソナライゼーション

- ビヘイビアツリーまたは階層的状態機械は、ケアタスクの説明可能な意思決定構造を提供する。
- パーソナライゼーションは、スケジュール、好ましい表現、支援閾値を数週間のインタラクションにわたって適応させる。
- 信頼度認識プランナは、知覚信頼度が低い場合に支援を延期したり人間の支援を要求したりする。

コンパクトな多目的コスト形式がトレードオフの調整に役立つ。決定方針は重み付きコストを最小化する：

$$[H]J(\pi) = w_{\text{task}} C_{\text{task}}(\pi) + w_{\text{safety}} C_{\text{safety}}(\pi) + w_{\text{energy}} C_{\text{energy}}(\pi) + w_{\text{privacy}} C_{\text{privacy}}(\pi), \quad (345)$$

ここで C_{task} は負のタスク効用、 C_{safety} は境界違反を罰し、 C_{energy} は消費を測り、 C_{privacy} はデータ露出リスクを定量化する。重み w は運用上の優先度を反映し、展開先ごとに調整可能である。

制御例：コンプライアントインタラクションのためのインピーダンス則

$$\tau = K_p(q_{\text{des}} - q) + K_d(\dot{q}_{\text{des}} - \dot{q}) + \tau_{\text{ff}},$$

ここでゲイン K_p, K_d は知覚された人間との近さによってスケジュールされ、ユーザ近傍で剛性を下げる。

実装パターン：転倒検知と行動選択

- 転倒検知は極めて低い偽陰性率が必要である。ビジョンベースの骨格追跡とウェアラブルセンサのIMU 閾値を組み合わせる。
- 転倒を検知した場合、ビヘイビアマネージャは以下を実行しなければならない：
 1. 1–2 s 以内にセカンダリセンサで検証し、誤報を回避する。
 2. バーチャルチェックインを発し、応答を聴取する。
 3. 肯定応答がない場合、介護者に通知し、安全距離を保ちながらアシスト態勢を取る。

実装コード：融合転倒検知とビヘイビアトリガの ROS2 ノード

コードサンプル 141 Fused fall-detection and trigger node

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import Imu
from std_msgs.msg import Bool
import threading

class FallDetector(Node):
    def __init__(self) -> None:
        super().__init__('fall_detector')
```

```

qos = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=1
)

self._imu_sub = self.create_subscription(
    Imu, '/wearable/imu', self._imu_cb, qos)
self._vision_sub = self.create_subscription(
    Bool, '/vision/fall_flag', self._vision_cb, qos)
self._trigger_pub = self.create_publisher(
    Bool, '/assist/trigger', 10)

self._imu_fall = False
self._vision_fall = False
self._lock = threading.Lock()

self._timer = self.create_timer(0.1, self._timer_cb) # 10 Hz

def _imu_cb(self, msg: Imu) -> None:
    acc = (msg.linear_acceleration.x ** 2 +
           msg.linear_acceleration.y ** 2 +
           msg.linear_acceleration.z ** 2) ** 0.5
    with self._lock:
        self._imu_fall = acc > 25.0 # 閾値: 25 m/s2

def _vision_cb(self, msg: Bool) -> None:
    with self._lock:
        self._vision_fall = msg.data

def _timer_cb(self) -> None:
    with self._lock:
        trigger = (self._imu_fall and self._vision_fall) or self._vision_fall
        self._trigger_pub.publish(Bool(data=trigger))

def main(args=None) -> None:
    rclpy.init(args=args)
    node = FallDetector()
    try:
        rclpy.spin(node)

```

```

except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

(インラインコメントは工学上の選択を示し、閾値は実証的検証が必要。)

設計上のトレードオフと運用上のリスク

- ・安全性対自律性：高い自律性は介護負荷を減らすすが、誤った判断による責任が増大する。保守的な安全重みはリスク行動を減らすすが誤陽性を増やす。
- ・エネルギー対応性：常時高 readiness はバッテリーを消耗させ、イベント駆動センシングは消費を減らすすが検知遅延を加える。
- ・プライバシー対状況認識：連続ビデオは最高の知覚を与えるがプライバシーリスクが増大する。エッジ推論と一時的ストレージは露出を減らす。
- ・規制と人間要因：接触力、ロギング、フェイルセーフ動作を認証する。スタッフと介護者がロボットのプロンプトを解釈し動作を上書きする訓練を行う。

工学上の示唆：胴体、センサ、意思決定ロジックを共設計し、説明可能なビヘイビアマネージャを優先し、段階的な臨床試験的検証でシステムを検証する。安全余裕を定量化し、式 (1) の重みを現地規制閾値と介護者期待に合わせて調整する。

42.2 外科・リハビリテーションにおけるロボット

高齢者ケア支援ロボットに関する前節の議論を踏まえ、外科・リハビリテーション用途ではより高い精度、予測可能なコンプライアンス、認証済みの安全性が求められる。以下では、手術室・リハビリテーションクリニックに導入されるヒューマノイドシステムに対する工学要件、制御手法、センシング統合、実装上の注意、具体的なトレードオフを展開する。

問題定義と運用要件。ヒューマノイドロボットは脆弱な組織を操作し、外科手術器具を扱い、歩行訓練中の患者を支持しなければならない。主要要件は以下の通りである：

- ・顕微外科手術向けのサブミリメートル単位のエンドエフェクタ位置決め。
- ・組織損傷閾値以下の制御された相互作用力。
- ・器具用の無菌インターフェースまたは着脱可能な無菌カバー。
- ・低遅延テレオペレーションと信頼性の高い自律フォールバック。
- ・予測不能な患者の動きと生理的ばらつきへのコンプライアンス。

技術分析：ダイナミクス、制御、センシング。安全な物理的相互作用のため、インピーダンス制御が主要な手法である。インピーダンス制御は、エンドエフェクタの運動と外力との間に所望の動的関係を実現し、ロボットをマス・スプリング・ダンパのように振る舞わせる。位置誤差を $e = x - x_d$ と

定義する。一般的な連続時間定式化は

$$[H]M_d\ddot{e} + B_d\dot{e} + K_de = F_{\text{ext}}, \quad (346)$$

である。ここで M_d, B_d, K_d は設計者が選ぶ慣性、減衰、剛性行列であり、 F_{ext} は測定された外力である。この振る舞いをヒューマノイドアームで実現するには、ジャコビアン $J(q)$ を介して関節トルク τ とカルテシアン力 F を写像する必要がある：

$$[H]\tau = J(q)^\top F + \tau_{\text{null}}, \quad (347)$$

τ_{null} は運動学的ヌル空間での姿勢制御に用いられる。

アドミタンス制御は双対的である：外力を測定し積分して運動指令を生成する。アクチュエータのインピーダンスが低い場合や力センサが信頼できる測定を提供する場合にアドミタンスを選択する。テレオペレーションでは、大域運動に位置制御、接触相にインピーダンス／アドミタンスを組み合わせたハイブリッド方式が用いられる。

センシング・知覚統合には以下を含める必要がある：

- ・局所相互作用フィードバック用の手首高帯域 6 軸力・トルク。
- ・器具位置決め用のステレオ／単眼内視鏡ビジョンとフィデューシャル追跡の融合。
- ・リハビリテーション中の患者姿勢用の深度センサまたは構造化光。
- ・歩行支援デバイスでの意図検出用の生理センサ（四肢への EMG、IMU）。

実装：ソフトウェアアーキテクチャと検証。層状制御スタックを用いる：

1. 安定性のためリアルタイムハードウェア上で ≥ 1 kHz で動作する低レベルトルクリープ。
2. (349) とヌル空間姿勢制御を実装するミドルレベルカルテシアンコンプライアンスコントローラ。
3. イメージング、テレオペレータ入力、安全モニタを統合する高レベルタスクプランナ。

Isaac Sim によるシミュレーション駆動開発により、知覚、接触ダイナミクス、患者固有解剖モデルとのヒューマンロボット共シミュレーションの段階的検証が可能となる。検証に有用な指標は、追従誤差、接触力ピーク、制御ループ受動性である。

実用的なコントローラ例。以下の Python スニペットは、ROS2 およびリアルタイムトルクインターフェースとの統合に適した簡略化カルテシアンインピーダンスコントローラループを示す。レート制限と組織損傷防止のための接触力安全クランプを含む。

コードサンプル 142 ヒューマノイドアーム用簡略カルテシアンインピーダンスコントローラ

```
#!/usr/bin/env python3
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.clock import Clock
from builtin_interfaces.msg import Time
from geometry_msgs.msg import WrenchStamped, TwistStamped, PoseStamped
from std_msgs.msg import Float64MultiArray, Bool
```

```

from sensor_msgs.msg import JointState
import threading
import time

class CartesianImpedanceController(Node):
    def __init__(self):
        super().__init__('cartesian_impedance_controller')

        # パラメータ
        self.declare_parameter('dt', 0.001)
        self.declare_parameter('Kd_diag', [2000.0]*3 + [50.0]*3)
        self.declare_parameter('Bd_diag', [50.0]*3 + [5.0]*3)
        self.declare_parameter('Md_diag', [1.0]*3 + [0.1]*3)
        self.declare_parameter('force_limit', 30.0)

        self.dt = self.get_parameter('dt').value
        self.Kd = np.diag(self.get_parameter('Kd_diag').value)
        self.Bd = np.diag(self.get_parameter('Bd_diag').value)
        self.Md = np.diag(self.get_parameter('Md_diag').value)
        self.force_limit = self.get_parameter('force_limit').value

        # 購読
        self.create_subscription(PoseStamped, 'current_pose', self.cb_pose, 1)
        self.create_subscription(TwistStamped, 'current_twist', self.cb_twist, 1)
        self.create_subscription(WrenchStamped, 'external_wrench', self.cb_wrench, 1)
        self.create_subscription(PoseStamped, 'desired_pose', self.cb_des_pose, 1)
        self.create_subscription(TwistStamped, 'desired_twist', self.cb_des_twist, 1)
        self.create_subscription(JointState, 'joint_states', self.cb_joints, 1)

        # 配信
        self.tau_pub = self.create_publisher(Float64MultiArray, 'joint_torque_command', 1)
        self.stop_pub = self.create_publisher(Bool, 'emergency_stop', 1)

        # 状態変数
        self.x = np.zeros(6)
        self.xd = np.zeros(6)
        self.x_dot = np.zeros(6)
        self.xd_dot = np.zeros(6)
        self.Fext = np.zeros(6)
        self.q = np.array([])

```

```

self.J = np.zeros((6, 7)) # 7DoF假定

self.lock = threading.Lock()
self.timer = self.create_timer(self.dt, self.control_loop)

def cb_pose(self, msg: PoseStamped):
    p = msg.pose.position
    q = msg.pose.orientation
    # 簡易的に位置+ZYXオイラー角に変換
    with self.lock:
        self.x[:3] = [p.x, p.y, p.z]
        # オイラー角変換省略（実機では適切に変換）

def cb_twist(self, msg: TwistStamped):
    with self.lock:
        self.x_dot[0] = msg.twist.linear.x
        self.x_dot[1] = msg.twist.linear.y
        self.x_dot[2] = msg.twist.linear.z
        self.x_dot[3] = msg.twist.angular.x
        self.x_dot[4] = msg.twist.angular.y
        self.x_dot[5] = msg.twist.angular.z

def cb_des_pose(self, msg: PoseStamped):
    p = msg.pose.position
    with self.lock:
        self.xd[:3] = [p.x, p.y, p.z]

def cb_des_twist(self, msg: TwistStamped):
    with self.lock:
        self.xd_dot[0] = msg.twist.linear.x
        self.xd_dot[1] = msg.twist.linear.y
        self.xd_dot[2] = msg.twist.linear.z
        self.xd_dot[3] = msg.twist.angular.x
        self.xd_dot[4] = msg.twist.angular.y
        self.xd_dot[5] = msg.twist.angular.z

def cb_wrench(self, msg: WrenchStamped):
    with self.lock:
        self.Fext[0] = msg.wrench.force.x
        self.Fext[1] = msg.wrench.force.y

```

```

        self.Fext[2] = msg.wrench.force.z
        self.Fext[3] = msg.wrench.torque.x
        self.Fext[4] = msg.wrench.torque.y
        self.Fext[5] = msg.wrench.torque.z

def cb_joints(self, msg: JointState):
    with self.lock:
        self.q = np.array(msg.position)

def compute_jacobian(self):
    # 実機ではFK/Jacobianライブラリ使用
    return np.eye(6, len(self.q))

def compute_nullspace_torque(self):
    # 簡易ポスチャ制御（重力補償含む）
    return np.zeros(len(self.q))

def control_loop(self):
    with self.lock:
        x = self.x.copy()
        xd = self.xd.copy()
        x_dot = self.x_dot.copy()
        xd_dot = self.xd_dot.copy()
        Fext = self.Fext.copy()
        J = self.compute_jacobian()

    e = x - xd
    edot = xd_dot - x_dot
    # インピーダンス則
    Fdes = self.Md @ (-edot / self.dt) + self.Bd @ edot + self.Kd @ e

    # 安全チェック
    if np.linalg.norm(Fext) > self.force_limit:
        Fdes = np.zeros(6)
        self.stop_pub.publish(Bool(data=True))

    tau = J.T @ Fdes + self.compute_nullspace_torque()
    msg = Float64MultiArray()
    msg.data = tau.tolist()
    self.tau_pub.publish(msg)

```

```
def main(args=None):
    rclpy.init(args=args)
    node = CartesianImpedanceController()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

設計上のトレードオフと工学上の含意。

- ・剛性対コンプライアンス：剛性を高めると位置決め精度は向上するが、衝突時の組織損傷リスクが増大する。患者接触タスクでは K_d を保守的に設計する。
- ・帯域とセンシング：力センサノイズとコントローラ帯域が達成可能なコンプライアント振る舞いを決定する。ヒューマノイドアームで安定トルク制御を行うには ≥ 1 kHz でサンプリングする。
- ・自律対テレオペレーション：完全自律は外科医の負荷を軽減するが、検証負荷が増大する。監視付き自律のハイブリッドモードは規制リスクを低減する。
- ・機械設計：軽量・バックドライバブル関節はコンプライアンスとエネルギー効率を向上させる。重いアクチュエータは精度を提供するが安全な接触を損なう。

運用上のリスクと緩和策。

- ・知覚パイプラインの遅延は不安定相互作用を誘発する；タイムアウトとフォールバックを強制する。
- ・センサ故障モードは、ゼロトルクまたは関節ブレーキのような受動的安全挙動をトリガしなければならない。
- ・手術室の電磁干渉はシールドケーブルと冗長通信路を必要とする。
- ・規制・臨床検証要件は、トレーサブルロギング、安全臨界コンポーネントの形式検証、導入前の広範な死体・動物実験を義務付ける。

特定の工学上の選択は認証パス、クリニック統合、患者アウトカムに影響する。ヒューマノイド外科・リハビリテーションロボットを設計する際には、予測可能・検証可能なコンプライアンス、高レートトルク制御、層状安全モニタを最優先とする。

42.3 医療ロボティクスの課題

前の小節では、外科における精密な器具操作、リハビリテーションにおける適応的な移動・運動プロトコル、高齢者ケアに求められる支援的役割といった具体的な応用を説明した。これらの使用例は、重複するシステム要件と医療ヒューマノイドに特有の失敗モードを露呈し、独特の困難さを生み出す。

問題定義：患者の安全、規制適合、臨床的有效性を確保しながら、医療環境にヒューマノイドロボットを展開する。主な制約には、人間との近接、予測不能な相互作用、感染制御、厳格な妥当性確

認要件が含まれる。工学上のトレードオフは、自律性と検証可能性、操作的敏捷性と安全なコンプライアンス、センシング忠実度と処理遅延の間に現れる。

技術分析は、中核となるサブシステムと測定可能な要件に焦点を当てる。

1. 安全臨界制御とコンプライアント操作

- ・ヒューマノイドマニピュレータは、接触中に柔らかく予測可能な相互作用力を提供しなければならない。モデルベースの動力学がベースラインを提供する：

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J^T(q)f_{\text{ext}}, \quad (348)$$

ここで q は関節角度、 M は慣性行列、 C はコリオリ項を集めたもの、 g は重力、 τ はアクチュエータトルク、 J はヤコビアン、 f_{ext} は外部接触力である。力認識制御は、相互作用動力学を形成するためのインピーダンス制御を用いる。実用的な制御則は：

$$[H]\tau = J^T(q)f_{\text{des}} + K_p(q_{\text{des}} - q) + K_d(\dot{q}_{\text{des}} - \dot{q}), \quad (349)$$

これはタスク空間の所望力と関節空間の剛性・減衰をブレンドする。ゲイン K_p, K_d は、センサおよびアクチュエータ遅延の下で受動性を保つように調整されなければならない。

2. 知覚、センサ融合、臨床シーン理解

- ・医療シーンでは、人、器具、微妙な組織変形の検出が必要である。ステレオ/深度カメラ、ToF センサ、IMU、触覚アレイの融合が曖昧さを減らす。
- ・センサ遅延 τ_s とジッタ σ_s は制御安定性を劣化させる。閉ループ接触制御では、代表的な帯域幅で発振を回避するため、最悪ケースのループ遅延 $\tau_{\text{loop}} < 50 \text{ ms}$ を維持する。
- ・確率的フィルタ（拡張カルマンフィルタ、EKF）は非線形運動学と時変観測雑音を扱う。EKF の予測・更新サイクルはリアルタイムで境界されなければならない。

3. 滅菌可能性とハードウェア信頼性

- ・材料とアクチュエータは、繰り返される滅菌サイクルと化学暴露に耐えなければならない。密封された触覚センサと IP 定格ジョイントはメンテナンスコストを増大させる。
- ・力および位置のための冗長センシングは単一点故障を緩和する。

4. ヒューマン・ロボット・インタラクション（HRI）と臨床ワークフロー統合

- ・振る舞いは臨床医に解釈可能でなければならない。意思決定コンポーネントは監査のためのトレーサビリティを必要とする。認証を促進するために、明示的な失敗モードを持つビヘイビアツリーを用いる。

実装ガイドラインと実用的レシピ

- ・リアルタイムアーキテクチャ：ハードリアルタイム制御ループ（高周波、 $< 1 \text{ ms}$ ジッタ）を非臨界知覚層から分離する。高層にはリアルタイムマイクロカーネルまたは RT-PREEMPT Linux と ROS2 を用いる。
- ・安全層：ハードウェア非常停止、ソフトウェア監視トルク制限、異常センサ読み取りで動作を停止するランタイム安全監督者を実装する。
- ・検証と妥当性確認：記録された臨床シナリオを持つテストハーネスを採用する。Isaac Sim のような物理ベースシミュレータをシナリオ再生とストレステストに用いる。

リアルタイムインピーダンス制御ループの例（簡略化）。この疑似コードは、センサ融合が`f_ext`を提供し、リアルタイムループスケジューラがタイミングを管理することを想定している。

コードサンプル 143 リアルタイムインピーダンス制御ループ（簡略化）

```
import time
import logging
import numpy as np
from typing import Tuple, Optional
from dataclasses import dataclass
from rt_sched import wait_next_cycle

# ロギング設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class ImpedanceGains:
    Kp: np.ndarray # 関節剛性 (Nm/rad)
    Kd: np.ndarray # 関節減衰 (Nms/rad)

class ImpedanceController:
    def __init__(self, gains: ImpedanceGains, torque_limits: np.ndarray):
        self.gains = gains
        self.torque_limits = torque_limits
        self._is_running = False

    def compute_torque(self,
                       q: np.ndarray,
                       qd: np.ndarray,
                       q_des: np.ndarray,
                       qd_des: np.ndarray,
                       f_des: np.ndarray,
                       jacobian: np.ndarray) -> np.ndarray:
        """インピーダンス則によるトルク計算"""
        tau_task = jacobian.T @ f_des
        tau_feedback = self.gains.Kp * (q_des - q) + self.gains.Kd * (qd_des - qd)
        return tau_task + tau_feedback

    def saturate_torque(self, tau: np.ndarray) -> np.ndarray:
        """トルク制限の適用"""
```

```

        return np.clip(tau, -self.torque_limits, self.torque_limits)

def read_joint_state() -> Tuple[np.ndarray, np.ndarray]:
    """ 関節状態読み取り（モック実装） """
    return np.zeros(7), np.zeros(7)

def read_force_sensors() -> np.ndarray:
    """ 力センサ読み取り（モック実装） """
    return np.zeros(6)

def jacobian_transpose(q: np.ndarray) -> np.ndarray:
    """ ヤコビアン計算（モック実装） """
    return np.eye(6, 7)

def send_torques(tau: np.ndarray) -> None:
    """ トルク送信（モック実装） """
    pass

class Planner:
    def get_setpoint(self) -> Tuple[np.ndarray, np.ndarray]:
        return np.zeros(7), np.zeros(7)

    def get_desired_force(self) -> np.ndarray:
        return np.zeros(6)

def control_loop(controller: ImpedanceController,
                 planner: Planner,
                 control_freq: float = 1000.0) -> None:
    """ リアルタイム制御ループ """
    dt = 1.0 / control_freq
    controller._is_running = True

    try:
        while controller._is_running:
            t0 = time.time()

            # センサ読み取り
            q, qd = read_joint_state()
            f_ext = read_force_sensors()
            q_des, qd_des = planner.get_setpoint()

```

```

        f_des = planner.get_desired_force()

        # トルク計算
        J = jacobian_transpose(q)
        tau = controller.compute_torque(q, qd, q_des, qd_des, f_des, J)

        # 安全チェック
        if np.any(np.abs(tau) > controller.torque_limits):
            tau = controller.saturate_torque(tau)
            logger.warning("トルク制限到達")

        send_torques(tau)
        wait_next_cycle(t0, dt)

    except KeyboardInterrupt:
        logger.info("制御ループ停止")
    finally:
        send_torques(np.zeros_like(controller.torque_limits))

if __name__ == "__main__":
    # ゲイン設定
    gains = ImpedanceGains(
        Kp=np.full(7, 50.0),
        Kd=np.full(7, 5.0)
    )

    # トルク制限 (例: 各関節150Nm)
    torque_limits = np.full(7, 150.0)

    controller = ImpedanceController(gains, torque_limits)
    planner = Planner()

    control_loop(controller, planner)

```

設計メトリクスと検証目標

- 位置精度：外科タスクでサブミリメートル、支援タスクで 1–5 mm。
- 力センシング分解能：繊細な組織取り扱いで 0.1–0.5 N。
- 安全遅延：ヒト接触シナリオで検出から作動まで 50 ms 未満。
- 平均故障間隔 (MTBF)：臨床環境で目標 MTBF > 1,000 時間。

運用上の意味、トレードオフ、リスク

- ・トレードオフ：

1. 高いコンプライアンスは組織損傷リスクを減らす但し位置精度を下げる。
2. センサ冗長性の増加は安全性を向上させるがコストとメンテナンスを増大させる。
3. リアルタイム保証は、より単純な知覚パイプラインまたは専用ハードウェアを要求する。

- ・リスク：

1. 検出されないセンサドリフトは、長時間運転中に潜在的な安全でない振る舞いを引き起こす。
2. 自律性への過度の信頼は臨床医の監督を希薄化し、責任を高める。
3. 不適切な滅菌設計は感染リスクと停止時間を引き起こす。

エンジニアは、設計時に検証可能性、決定論的タイミング、ヒト中心の失敗モードを優先しなければならない。文書化されたゲイン調整手順、形式化された安全監督者、継続的なシミュレーションからハードウェアへの妥当性確認は、臨床展開リスクを軽減する。

43 教育におけるロボット

43.1 ヒューマノイドロボットを用いた STEM 教育

本章で扱った実環境配備ロボットの現実的制約を踏まえ、ヒューマノイドを STEM 教育に統合する際にも、信頼性・安全性・測定可能な成果への同様の配慮が必要である。教室という場合は、生徒の能力のばらつき、技術スタッフの不足、厳格な安全・プライバシー要件といった追加の制約を課す。

問題定義：物理学・プログラミング・制御・知覚の体験型学習をヒューマノイドプラットフォームで実現せよ。システムは以下を満たすこと：

- ・カリキュラム標準に準拠した漸進的タスクを提示；
- ・異なる学習レベル向けの制御抽象層（ビジュアルプログラミング、スクリプト、ROS）を公開；
- ・繰り返しの毎日利用に対して安全で保守可能；
- ・学習成果とロボット性能の両方に対する客観的メトリクスを提供。

技術分析：教育学的目標を工学要件に翻訳する。

- ・機械・電氣的頑健性：アクチュエータは頻繁な低力インタラクションと偶発的衝突に耐えなければならない。安全接触のため、トルク制限付きモータにコンプライアント要素または series elastic actuation を指定する。
- ・センサ・計算能力：知覚タスクには、バランス・運動デモで応答的フィードバックを得るため 30–60 Hz のカメラと 200 Hz 以上の IMU が必要。リアルタイムビジョンモデルを動作させる際のオンボード推論には GPU アクセラレーションが必要；NVIDIA Jetson クラスまたは Xavier クラスモジュールが適切。
- ・ソフトウェアスタックと抽象層：3つの制御層を提供する。
 1. 高レベルグラフィカル層（ビジュアルプログラミング）は意味論的命令を出力。
 2. 中レベルスクリプト可能 API は Python または Jupyter notebook 向けで、アルゴリズム授業に適す。

3. 低レベル ROS2 トピック・サービスは、メッセージングとリアルタイム制約を学ぶ上級生向け。
- 安全性と監視：ハードウェア E-stop とソフトウェアウォッチドッグが暴走動作を防ぐ。ジョイント電流と近接センサを監視する監督コントローラを統合。

定量的制約と例.

- センササンプリングは、対象信号の最高周波数に対するナイキストを尊重しなければならない。動特性を f_{\max} Hz まで測定する IMU について、

$$[H]f_s > 2f_{\max}, \quad (350)$$

を用い、実際には頑健な推定・フィルタリングのため $f_s \geq 5-10 f_{\max}$ を選ぶ。

- リアルタイム制御ループは通常、閉ループ帯域幅 f_b の 1 桁高速にサンプリングする。制御周期 T_c を

$$[H]f_s = 1/T_c \geq 10f_b, \quad (351)$$

を満たすように選ぶ。

- 信頼性は授業スケジュールに影響する。平均故障間隔 (MTBF) と平均修理時間 (MTTR) を用いて期待稼働率を計算：

$$[H]\text{稼働率} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}. \quad (352)$$

定期授業利用には稼働率 0.95 以上を目標とし、この要件が予備品在庫と遠隔診断を促進する。

カリキュラムマッピングと評価. ハードウェア機能を学習目標にマッピングする。

- 運動学・動力学：歩行デモを用いて逆運動学を教える。パラメータを提供し、生徒が歩行位相時間を変更可能にする。
- 制御理論：生徒が胴体角度を調節する PID コントローラをチューニング；定着時間とオーバーシュートを測定。
- 知覚と機械学習：生徒がロボットでラベル付きデータセットを収集し Isaac Sim でシミュレーション内で物体検出器を学習し、モデルをロボットに展開。
- 計算的思考：ROS2 ベースのメッセージパッシングとレイテンシ解析の演習を提供。

実装レシピと教室ワークフロー.

1. 準備：ロボットファームウェアをミラーするシミュレーションツイン (Isaac Sim) を維持する。初期デバッグと生徒練習にシミュレーションを用いる。
2. 段階的インタフェース：教師が低レベルアクセスをロックしながら、生徒が高レベル動作をプログラムできるようにする。
3. モニタリング：ジョイントヘルス、CPU/GPU 負荷、カメラドロップを Web ダッシュボードにレポートする軽量テレメトリを実装。

実用的コード例. 以下の ROS2 Python ノードは安全な「ガイド付き授業」モードを実装する。高レベルの生徒命令を受信し、安全制限を適用してから低レベルコントローラにアクションを転送する。

```

#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import String, Bool
from sensor_msgs.msg import JointState
from builtin_interfaces.msg import Time
import re
import threading

MAX_JOINT_VEL = 1.0 # rad/s
MAX_TORQUE = 2.0 # Nm
VEL_REGEX = re.compile(r"vel\s*=\s*([+-]?[0-9]*\.[0-9]+)")
TORQUE_REGEX = re.compile(r"torque\s*=\s*([+-]?[0-9]*\.[0-9]+)")

class GuidedLessonBridge(Node):
    def __init__(self):
        super().__init__('guided_lesson_bridge')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self.sub_cmd = self.create_subscription(String, '/student_command', self.cb_cmd)
        self.pub_ctrl = self.create_publisher(String, '/low_level_command', qos)
        self.pub_estop = self.create_publisher(Bool, '/emergency_stop', qos)
        self.sub_joint = self.create_subscription(JointState, '/joint_states', self.cb_joint)
        self.timer = self.create_timer(1.0, self.heartbeat)
        self.last_joint_state = None
        self.lock = threading.Lock()

    def cb_cmd(self, msg: String):
        with self.lock:
            cmd = msg.data.strip()
            if not cmd:
                return
            # 速度リミット適用
            cmd = VEL_REGEX.sub(lambda m: f"vel={min(float(m.group(1)), MAX_JOINT_VEL)}")

```

```

        # トルクリミット適用
        cmd = TORQUE_REGEX.sub(lambda m: f"torque={min(float(m.group(1)), MAX_TORQUE)}", cmd)
        self.pub_ctrl.publish(String(data=cmd))

    def cb_joint(self, msg: JointState):
        with self.lock:
            self.last_joint_state = msg
            # エフォートが閾値超過なら緊急停止
            if msg.effort:
                if any(abs(e) > MAX_TORQUE for e in msg.effort):
                    self.pub_estop.publish(Bool(data=True))

    def heartbeat(self):
        self.get_logger().info('guided_lesson_active', throttle_duration_sec=1.0)

def main(args=None):
    rclpy.init(args=args)
    node = GuidedLessonBridge()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

運用上の影響、トレードオフ、リスク。

- コスト対能力：高度なオンボード計算は自律性を高めるが、ユニットコストと熱管理要件を増大。
- 複雑性対アクセシビリティ：より強力な API は研究レベルプロジェクトを可能にするが、教員サポートを要する。
- 安全性のトレードオフ：コンプライアンス追加は衝突深刻度を低減するが、精密制御タスクを複雑化。
- データプライバシー：教室の記録と生徒データは同意と安全な保管を要する。
- 保守負担：主要予備品の在庫、ファームウェア更新の自動化、展開前検証へのシミュレーション活用で高稼働率を達成。

したがって、ヒューマノイドを用いた効果的な STEM 体験の設計は、ハードウェア特性・ソフトウェア抽象・保守運用を教育目標にマッチさせ、信頼性・安全余裕・期待学習効果を定量化する工学演習となる。

43.2 対話的学習体験

前の小節では、ヒューマノイドを用いたカリキュラムに準拠した STEM 活動と教室ワークフローについて説明した。本小節では、その基盤を拡張し、対話的学習体験が生徒の信号に応じてリアルタイムで教授法を適応させる方法を詳述する。

対話的学習は、ヒューマノイド、生徒、環境の間に閉ループシステムを創出する。問題提起：生徒の状態を感知し、課題難易度とロボット挙動を適応させ、安全性と教育的価値を保証する対話制御器を設計せよ。運用要件は、低遅延センシング、解釈可能な適応ルール、物理的対話のためのフェイルセーフ挙動を含む。以下の分析はこれらの要件を形式化し、実装可能なパターンを示す。

技術分析 — センシングと生徒状態モデリング：

- マルチモーダルセンシングは、エンゲージメントと理解度について相補的な証拠を提供する。代表的なセンサ：視線と姿勢のための RGB-D カメラ、音声応答のためのマイクロホンアレイ、誘導操作のためのロボット手の触覚/力センサ、教室小道具に組み込まれた慣性計測ユニット (IMU)。
- 生徒エンゲージメント指標 E_t をモダリティ特化型信号の正規化線形結合として定義する。 V_t を視覚エンゲージメントスコア、 A_t を音声エンゲージメントスコア、 T_t を触覚相互作用スコアとする。すると

$$[H]E_t = w_v V_t + w_a A_t + w_t T_t, \quad \sum_i w_i = 1, \quad 0 \leq w_i \leq 1. \quad (353)$$

重み w_i はカリキュラム設計者により設定されるか、フィールドデータから調整される。各モダリティスコアは同じレンジ、例えば $[0, 1]$ に較正されなければならない。

適応教授法制御則：

- 課題難易度 $d_t \in [0, 1]$ に対して単純な比例適応を用いる。生徒の成功度が目標 \bar{r} を超えるとロボットは難易度を上げ、そうでなければ下げる。堅牢な離散更新は

$$[H]d_{t+1} = \text{clip}(d_t + \alpha(r_t - \bar{r}) + \beta(E_t - \bar{E}), 0, 1), \quad (354)$$

ここで r_t は 2 値またはスカラー課題成功度、 α と β はゲインパラメータ、 \bar{E} は目標エンゲージメントレベルである。クリップは範囲外値を防ぐ。このルールは認知的成功とエンゲージメントを結合し、生徒が非エンゲージの際に難易度が漂流するのを防ぐ。

対話ループのための設計パターン：

- 感知：カメラ、マイク、タッチセンサを適切なレートでサンプリングする。視覚姿勢/視線には 10–30 Hz パイプラインが典型的。
- 知覚：視線、音声意図、接触イベントのための軽量モデルを実行する。遅延削減のため量子化出力を用いる。
- モデル： V_t, A_t, T_t を計算し、(1) により E_t を導出する。ノイズ平滑化のため短いスライディングウィンドウを適用する。
- 決定：(2) により d_{t+1} を計算する。ライブラリ（説明、ヒント、デモ、物理的誘導）から挙動プリミティブを選択する。
- 行動：安全性監視と制約チェックを伴い挙動プリミティブを実行する。
- 記録：オフライン評価のためセンサストリームと決定を記録する。

実装例：ROS2 ノードが適応レッスン制御を実行する。当該ノードは処理済知覚トピックを購読し挙動コマンドをパブリッシュする。以下のコードは教室実験のための最小限で展開可能なスケルトンである。

コードサンプル 145 ROS2 adaptive lesson node for humanoid interactions

```
import rclpy
from rclpy.node import Node
from rcl_interfaces.msg import SetParametersResult
from std_msgs.msg import Float32, String
import threading
import yaml
import os
from ament_index_python.packages import get_package_share_directory

class AdaptiveLessonNode(Node):
    def __init__(self):
        super().__init__('adaptive_lesson')

        # パラメータ宣言&デフォルト値
        self.declare_parameter('difficulty_init', 0.5)
        self.declare_parameter('alpha', 0.3)
        self.declare_parameter('beta', 0.2)
        self.declare_parameter('target_success', 0.8)
        self.declare_parameter('target_engagement', 0.6)
        self.declare_parameter('rate_hz', 10.0)
        self.declare_parameter('config_file', '')

        # パラメータ読み込み
        self.d = self.get_parameter('difficulty_init').value
        self.alpha = self.get_parameter('alpha').value
        self.beta = self.get_parameter('beta').value
        self.target_success = self.get_parameter('target_success').value
        self.target_engagement = self.get_parameter('target_engagement').value
        rate_hz = self.get_parameter('rate_hz').value

        # YAML設定ファイルがあれば上書き
        config_path = self.get_parameter('config_file').value
        if config_path and os.path.isfile(config_path):
            with open(config_path, 'r') as f:
```

```

        cfg = yaml.safe_load(f)
        self.d = float(cfg.get('difficulty_init', self.d))
        self.alpha = float(cfg.get('alpha', self.alpha))
        self.beta = float(cfg.get('beta', self.beta))
        self.target_success = float(cfg.get('target_success', self.target_success))
        self.target_engagement = float(cfg.get('target_engagement', self.target_engagement))

# 購読／配信
self.create_subscription(Float32, '/engagement', self.eng_cb, 10)
self.create_subscription(Float32, '/success', self.succ_cb, 10)
self.pub = self.create_publisher(String, '/behavior_cmd', 10)

# 状態変数
self.last_eng = 0.0
self.last_succ = 0.0
self.lock = threading.Lock()

# パラメータ変更コールバック
self.add_on_set_parameters_callback(self.param_cb)

# タイマー駆動で更新（受信タイミングに依存しない）
self.timer = self.create_timer(1.0 / rate_hz, self.update_and_issue)

def eng_cb(self, msg):
    with self.lock:
        self.last_eng = msg.data

def succ_cb(self, msg):
    with self.lock:
        self.last_succ = msg.data

def param_cb(self, params):
    # 実行中のパラメータ更新を許容
    for p in params:
        if p.name == 'alpha':
            self.alpha = p.value
        elif p.name == 'beta':
            self.beta = p.value
        elif p.name == 'target_success':
            self.target_success = p.value

```

```

        elif p.name == 'target_engagement':
            self.target_engagement = p.value
        return SetParametersResult(successful=True)

def update_and_issue(self):
    with self.lock:
        eng = self.last_eng
        succ = self.last_succ

        # 難易度更新 (式2)
        delta = self.alpha * (succ - self.target_success) + self.beta * (eng - self.target_engagement)
        self.d = max(0.0, min(1.0, self.d + delta))

        cmd = self.select_behavior(succ, eng)
        self.pub.publish(String(data=cmd))

def select_behavior(self, succ, eng):
    # 状態に応じた振る舞い選択
    if succ < 0.5:
        return 'hint'
    if eng < 0.4:
        return 'reengage'
    if self.d < 0.4:
        return 'practice'
    if self.d < 0.8:
        return 'challenge'
    return 'assessment'

def main(args=None):
    rclpy.init(args=args)
    node = AdaptiveLessonNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

```

評価と指標：

- ・学習効果（事前/事後テスト）、定着度、エンゲージメント時間を測定する A/B フィールド研究を用いる。効果量を信頼区間とともに報告する。
- ・安全指標を監視する：近接接触イベント、予期せぬ衝突、強制シャットダウンの頻度。
- ・センシングから作動までのシステム遅延を追跡し、応答的 HRI のため閉ループ遅延を 200 ms 未満に保つ。

エンジニアリングへの影響、トレードオフ、運用上のリスク：

- ・トレードオフ：リッチな知覚はパーソナライズを高めるが、遅延、計算コスト、プライバシー曝露を増大させる。教室制約に合わせてモダリティとモデル複雑度を選択する。
- ・頑健性：騒がしい教室音響と遮蔽によりモデル忠実度が低下する。マルチモーダル融合と不確実性を意識した決定閾値を用いる。
- ・安全リスク：物理的誘導はトルク制限と接触検出を要する。ハードウェアレベルでトルク飽和とソフトウェアウォッチドッグを実装する。
- ・スケーラビリティ：多くの教室へ展開するには自動較正、軽量モデル、遠隔更新機能を要する。
- ・倫理的・プライバシー配慮：センサデータを匿名化し、同意を得、生ストリームの持続的保存を最小化する。

上記の設計パターンはヒューマノイド展開に直接マッピングされる。リアルタイム適応のため軽量モデルを採用し、すべての物理的挙動に対して安全エンベロープを維持し、制御された研究により教育的便益を定量化する。これらの実践は効果的かつ運用上安全な対話的学習体験をもたらす。

43.3 ケーススタディ：教室におけるロボット

これまでのインタラクティブな教室シナリオと STEM チュータリング戦略の議論を踏まえ、本ケーススタディでは異年齢混合の教室への人型アシスタントのエンドツーエンド導入を検証する。焦点は、エンジニアリング要件、知覚・制御パイプライン、シミュレーション主導の開発、測定可能な学習成果にある。

問題定義と運用目標

- ・目的：教師の能力を拡張するため、少人数グループごとに 1 体の人型を配置し、個別指導、実演、形成的評価を実施する。
- ・主要パフォーマンス指標：生徒のエンゲージメント、タスク支援の正確さ、時間あたりの安全インシデント数、教師の受容性。
- ・運用制約：45 分授業、20–30 名の教室、限られたネットワーク帯域、厳格なプライバシー規則。

技術分析：センシング、知覚、インタラクション

- ・センサと配置：
 1. 前方 RGB-D カメラ：顔検出・ジェスチャ認識用。
 2. ファーフィールドマイクアレイ：話者定位・音声検出用。
 3. 手部タクティルセンサ：安全な物体受け渡し用。
 4. IMU：姿勢・転倒検出用。
- ・知覚スタック：

1. マルチモーダルフュージョン：視線注意、音声到来方向、近接を統合。
2. リアルタイムエンゲージメント推定器：各生徒に対し 0-1 のスカラー engagement_score を出力。
3. 感情・意図分類器：短時間ウィンドウで実行し、誤検出を削減。

エンゲージメントモデル生徒ごとの期待学習獲得量を、直接的なロボット対話時間とパーソナライズ精度の関数として操作化する単純線形モデルを用いる：

$$[H]\Delta S = \alpha t + \beta p + \gamma I, \quad (355)$$

ここで：

- ΔS は期待スコア向上、
- t はロボット-生徒対話時間（分）、
- $p \in [0, 1]$ はパーソナライズ精度（ロボットが授業を適応できた度合い）、
- I は能動的練習対受動的聴取のインジケータ、
- α, β, γ はパイロット研究から経験的に推定された係数。

キャパシティプランニングロボットが授業あたり t_{eff} の効果的対話時間を維持できる場合、クラスサイズ C に必要なロボット台数 N は

$$[H]N \geq \left\lceil \frac{C \cdot t_{\min}}{T_{\text{class}} \cdot \eta} \right\rceil, \quad (356)$$

を満たす。 t_{\min} は生徒あたり最小目標時間、 T_{class} は授業時間、 η は移動・管理オーバーヘッドを考慮した利用率因子。

制御・安全戦略

- プロクセミクス制御：最小接近距離を強制する有限状態機械で、人間の快適性のため速度ランプを滑らかに変更。
- コンプライアンス：低レベルトルク制御と高頻度安全モニタで衝突を 50 ms 以内に検出。
- フォーマル安全エンベロープ：センサフュージョンでロボット周囲に占有グリッドを生成し、遮蔽や予期せぬ動作時は受け渡しを禁止。

実装：シミュレーション優先パイプライン

- Isaac Sim を用いて教室のデジタルツインを構築。身長・座席配置が異なる人間アバターを配置。
- 合成データで知覚モデルを学習し、プライバシー準拠の匿名化済み教室記録でファインチューニング。
- 振る舞いロジックは GROOT ビヘイビアツリーで符号化し、フレキシブルなミッションスクリプティングを実現。教師がロボットコードを変更せず授業フェーズを設定可能。

実践的コード例以下の ROS2 Python ノードは生徒ごとのエンゲージメント推定値を配信する。オンボード計算向けの軽量オンライン推定器を示す。インラインコメントは簡潔で運用的。

コードサンプル 146 ROS2 node: publish engagement estimate

```

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Float32
from typing import Deque
from collections import deque
import yaml
import os
from ament_index_python.packages import get_package_share_directory

class EngagementNode(Node):
    def __init__(self) -> None:
        super().__init__('engagement_node')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self._pub = self.create_publisher(Float32, 'engagement_score', qos)
        self.declare_parameter('timer_period', 0.5)
        self.declare_parameter('window_size', 10)
        self.declare_parameter('weights', [0.6, 0.3, 0.1])
        timer_period = self.get_parameter('timer_period').value
        self._window_size = self.get_parameter('window_size').value
        weights = self.get_parameter('weights').value
        self._w_vis, self._w_aud, self._w_prox = weights
        self._window: Deque[float] = deque(maxlen=self._window_size)
        self.create_timer(timer_period, self._timer_cb)
        self.get_logger().info("EngagementNode initialized")

    def _timer_cb(self) -> None:
        try:
            visual = self._read_visual_attention()
            audio = self._read_audio_activity()
            prox = self._read_proximity()
        except Exception as e:
            self.get_logger().warning(f"Sensor read failed: {e}")
            return
        # 重み付き融合 & クリップ
        score = max(0.0, min(1.0, self._w_vis * visual +

```

```

        self._w_aud * audio +
        self._w_prox * (1.0 - prox)))

    self._window.append(score)
    avg = sum(self._window) / len(self._window) if self._window else 0.0
    msg = Float32(data=float(avg))
    self._pub.publish(msg)

def _read_visual_attention(self) -> float:
    # TODO: 実際の視線推定モジュールと接続
    return 0.8

def _read_audio_activity(self) -> float:
    # TODO: 実際の音声検出モジュールと接続
    return 0.2

def _read_proximity(self) -> float:
    # TODO: 実際の距離センサと接続
    return 0.1

def main(args=None):
    rclpy.init(args=args)
    node = EngagementNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

シミュレーションから現実への移行と評価

- ドメインランダム化により、Isaac Sim からハードウェアへの移行時の脆弱性を軽減。
- パイロット時にログすべき主要指標：誤認識率、平均応答遅延、教師による上書き頻度。
- A/B 管理試験により、式 (1) の係数 α, β, γ を定量化。

設計トレードオフと運用リスク

- トレードオフ：
 1. オンボード対エッジ計算：オンボードは遅延を削減するが重量・発熱が増加。
 2. センサ冗長性は信頼性を向上させるがコストとプライバシー曝露が増加。
 3. 高パーソナライズ精度は成果を向上させるが、より多くの生徒データを必要とする。

- ・リスク：
 1. 知覚エラーによる誤同定や不適切なプロンプト。
 2. 生徒がロボットに過度に依存し、ピアコラボレーションが減少する社会動態。
 3. 映像・音声保存を誤るとデータプライバシーおよび規制コンプライアンス違反。

エンジニアリングへの影響

- ・低遅延安全モニタと人間の快適性のための堅牢なプロクセミクス制御を最優先。
- ・シミュレーション主導の学習により、教室試行を削減し、現実的な故障モードを計測。
- ・その場で学習向上を測定し、式 (1) のパラメータを洗練。
- ・明確なデータガバナンスとオプトイン方針を採用し、プライバシーリスクを軽減。

本ケーススタディの運用化は、測定可能な教育効果と運用複雑性・コスト・プライバシーのバランスを取ることを要する。上記のエンジニアリング選択がそれらのトレードオフを定義し、人型アシスタントが教室の成果をスケールして改善できるかを決定する。

44 産業用ロボット

44.1 物流および倉庫におけるヒューマノイド

医療および教育における前述のケーススタディを踏まえ、物流および倉庫では高スループット、繰り返しの操縦、高密度の人間-ロボット相互作用という独自の運用上の制約が課される。以下の分析では、エンジニアリング問題を定式化し、実用的なサイジングおよび力要件を導出し、最小限のソフトウェアディスパッチを概説し、物流におけるヒューマノイドの展開に関するトレードオフとリスクを強調する。

問題定義. 倉庫では、ロボットが通路や動的なワークセル間で物品をピック、搬送、仕分け、パレタイズする必要がある。主要な性能指標は、スループット、平均サイクルタイム、ピック精度、稼働時間、人間の同僚に対する安全性である。設計目標は通常、注文量およびサービスレベル合意から導かれ、例えば時間当たり X ピックを達成し、オンタイム配達率を Y パーセント維持する。

技術分析.

- ・スループットおよびフリートサイジング. タスク到着を率 λ (時間当たりタスク数) としてモデル化する。平均ロボットサイクルタイムを C (タスクあたり時間数) とし、移動、知覚、把持、配置を含む。スループットを維持するために必要な最小ロボット数は

$$[H]N \geq \lceil \lambda C \rceil, \quad (357)$$

である。なぜなら各ロボットは時間当たり $1/C$ タスクを完了するから。 N 台のロボットがタスクを処理する場合、利用率 ρ は

$$[H]\rho = \frac{\lambda}{N}C = \frac{\lambda}{N\mu}, \quad (358)$$

であり、ここで $\mu = 1/C$ はサービス率である。目標利用率はばらつきおよび保守を考慮し、概ね 0.8 未満に留めるべきである。

- ・グリップングおよび操縦. 棚およびトートピッキングでは、把持力はペイロードの慣性および重

力要求を上回る必要がある。ペイロード質量 m 、最大腕加速度 a_{\max} 、重力加速度 g 、摩擦係数 μ_f に対し、法線把持力 F_n は

$$[H]F_n \geq \frac{m(g + a_{\max})}{\mu_f}, \quad (359)$$

に安全率 s_f (例: $s_f = 1.5$) を乗じた値を満たす必要がある。指先および触覚センシングの設計は、分散接触および再現可能な摩擦推定を確保すべきである。

- バランスおよびペイロードシフト. ヒューマノイドのバランスは、重心投影を支持ポリゴン内に維持する必要がある。準静的な平衡近傍動作では、ペイロードによる重心水平変位 Δx が支持ポリゴン端までのマージンを超えないよう確保する。動的な操縦では、リアルタイム ZMP またはキャプチャポイント制御を全身逆動力学と統合する必要がある。
- 知覚およびオクルージョン. 積み上げラックでの棚アクセスはオクルージョンおよび微小公差を生じる。RGB-D をアクティブセンシング (手首搭載深度センサ) と融合し、オンラインポーズリファインメントを用いる。知覚レイテンシは C に組み込む必要がある。

実装 (システムおよび制御). 堅牢な倉庫ヒューマノイドスタックは以下で構成される:

1. タスク層: オーダ管理、バッチ化、優先スケジューリング。
2. フリートマネージャ: 式 (1) に従いタスクをロボットに割り当て、バッテリー状態およびダウンタイムを監視。
3. 知覚および把持プランナ: 品質スコア付き把持候補を生成。
4. モーションプランナおよび全身コントローラ: 衝突フリーパスおよびバランス制約を保証。
5. 安全層: 速度制限、分離監視、および非常停止インタフェース。

実用的なタスクアロケータは最短完了時間ヒューリスティックに従う。以下のリストは、フリートマネージャ上で動作する最小 Python ディスパッチャを概略示す。共有 `task_queue` と推定完了時間を持つロボット状態を仮定する。

コードサンプル 147 推定完了時間を最小化する貪欲タスク割当。

```
#!/usr/bin/env python3
import heapq
import threading
import time
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Tuple

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from rmf_fleet_msgs.msg import RobotState, TaskSummary, FleetState
from rmf_task_msgs.msg import TaskProfile, TaskType

# 単一タスクを表す不変オブジェクト
```

```

@dataclass(frozen=True)
class Task:
    id: str
    travel: float
    pick: float
    place: float
    priority: int = 0 # 大きいほど優先

# ロボットの内部状態
@dataclass
class Robot:
    id: str
    eta: float = 0.0
    tasks: List[Task] = field(default_factory=list)
    battery: float = 1.0 # 0-1
    needs_swap: bool = False

# タスク予約管理ノード
class FleetManagerNode(Node):
    def __init__(self):
        super().__init__('fleet_manager')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,
            depth=10
        )
        self.task_sub = self.create_subscription(
            TaskProfile, 'task_requests', self._on_task, qos
        )
        self.fleet_sub = self.create_subscription(
            FleetState, 'fleet_states', self._on_fleet, qos
        )
        self.assign_pub = self.create_publisher(
            TaskSummary, 'task_assignments', qos
        )

        self._robots: Dict[str, Robot] = {}
        self._task_queue: List[Tuple[int, float, Task]] = []
# (-priority, stamp, task)
        self._lock = threading.Lock()

```

```

        self._timer = self.create_timer(0.5, self._assign)

# タスク到着時に優先度付きキューへ追加
def _on_task(self, msg: TaskProfile):
    task = Task(
        id=msg.task_id,
        travel=msg.description.travel_time,
        pick=msg.description.pick_time,
        place=msg.description.place_time,
        priority=msg.priority
    )
    with self._lock:
        heapq.heappush(self._task_queue, (-task.priority, time.time(), task))

# FleetState からロボット状態を更新
def _on_fleet(self, msg: FleetState):
    with self._lock:
        for r in msg.robots:
            if r.name not in self._robots:
                self._robots[r.name] = Robot(id=r.name)
            self._robots[r.name].battery = r.battery_percent / 100.0
            self._robots[r.name].needs_swap = r.battery_percent < 15.0

# ETA推定：バッテリー交換時間を考慮
def _estimate(self, robot: Robot, task: Task) -> float:
    base = robot.eta + task.travel + task.pick + task.place
    if robot.needs_swap:
        base += 300.0 # 5分交換
    return base

# 割当て実行
def _assign(self):
    with self._lock:
        while self._task_queue and self._robots:
            _, _, task = heapq.heappop(self._task_queue)
            best = min(self._robots.values(),
                       key=lambda r: self._estimate(r, task))
            best.eta = self._estimate(best, task)
            best.tasks.append(task)

```

```

        # 割当て通知
        assign = TaskSummary()
        assign.task_id = task.id
        assign.robot_name = best.id
        self.assign_pub.publish(assign)

def main():
    rclpy.init()
    node = FleetManagerNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

統合ノート. モーションプランニングは移動時間と操縦成功確率を共に最適化する必要がある。優先度付き逆運動学を器用なピックに用い、狭い通路での全身操縦には制約付きサンプリングプランナを用いる。知覚信頼度はスケジューラにフィードバックすべき：信頼度の低いタスクは人間のレビューのためバッチ化するか、より優れたセンシングを搭載したロボットに割り当てる。

エンジニアリングへの影響、トレードオフ、およびリスク。

- 速度対安全：速度を上げると C が短縮し式 (1) により必要 N が減るが、衝突リスクおよび機械的摩擦が増加する。安全規格は共有ワークスペースでの保守的な速度制限を要求する。
- エネルギーおよびデューティサイクル：高速ピックは消費電力を増加させる。バッテリーサイジングおよびホットスワップ戦略は、平均サイクルエネルギーに加え、持ち上げ時のピーク電力を考慮する必要がある。
- 保守オーバーヘッド：複雑なヒューマノイドは固有感覚、グリッパー、バランスコントローラの頻繁なキャリブレーションを要求する。保守ダウンタイムを効果的 ρ マージンに組み込む。
- 知覚故障モード：誤分類は物品落下および SKU エラーを引き起こす。冗長センシングおよび触覚検証は運用上のリスクを低減する。
- ROI およびフットプリント：ヒューマノイドは非構造化タスクでの器用さを提供するが、専用ロボットよりコストが高い。柔軟性、人間のようなリーチ、またはワークステーション共有が高い capex を正当化する場所にヒューマノイド展開を選択する。

運用計画は初期フリートサイジングに式 (1) および (2) を、グリップ設計に式 (3) を用いる。解析サイジングにタスク到着ばらつきのモンテカルロシミュレーションを組み合わせ、実用的な安全マージンを設定する。

44.2 高度な製造タスク

これまでのロジスティクスにおけるヒューマノイドに関する議論では、器用なピック・アンド・プレイスおよび人間を意識したナビゲーション能力に焦点を当て、これらはより要求の高い製造の役割を直接可能にする。本小節では、これらのスキルが精度、力制御、および統合センシングを要求する高度な製造タスクにどのようにスケールするかを検討する。

問題提示：現代の組立ラインは、精密挿入、ファスナー締結、部品位置合わせ、ハイブリッド加工といったタスクに対して柔軟な自動化を要求する。ヒューマノイドは、擬人化されたリーチと手の幾何形状によって治具の必要性を減らす点で利点を提案する。エンジニアリング目標は：指定公差以下の位置精度を確保し、サイクルタイム目標を満たし、高力動作中の安定性を維持し、共有ワークスペースにおける人間の安全を保証することである。

技術分析はタスク分解から始まる。各製造タスクは、知覚、運動計画、力相互作用のフェーズに分解される。サイクルあたりの典型的なタイミングモデルは

$$[H]T_{\text{cycle}} = T_{\text{percept}} + T_{\text{plan}} + T_{\text{exec}} + T_{\text{safety}}, \quad (360)$$

ここで T_{percept} はセンシング時間、 T_{plan} は経路および把持計算、 T_{exec} はアーム実行、 T_{safety} はコンプライアンスおよび検証である。スループット R は定常運転下で $R = 1/T_{\text{cycle}}$ である。

運動学的実現可能性と安定性制約が主要である。到達可能ワークスペースマージンを、ターゲットとロボットの運動学的特異点多様体との最小距離として定義する。動作マージン M_{reach} は

$$[H]M_{\text{reach}} = \min_{t \in T} \sigma_{\min}(J(q(t))), \quad (361)$$

ここで σ_{\min} はマニピュレータヤコビアン $J(q)$ の最小特異値である。設計はコンプライアント制御に適した条件数を保証するため $M_{\text{reach}} > \epsilon$ を要求する。

力およびトルクバジェットはアクチュエータ選択を導く。長さ l のマニピュレータリンクが質量 m を扱う場合、静的肩トルク見積もりは

$$[H]\tau_{\text{req}} \geq mgl \sin \theta + \tau_{\text{tool}}, \quad (362)$$

重力 g 、関節角度 θ 、ツールトルク τ_{tool} である。安全率 S_f を含めて指定アクチュエータトルク $\tau_{\text{spec}} = S_f \tau_{\text{req}}$ とする。

製造のための制御戦略は、ハイブリッド位置/力制御および力相互作用によって拘束される動作のためのモデル予測制御（MPC）を強調する。MPC 目的は典型的に追従誤差と制御努力を最小化する：

$$[H] \min_{u_{0:N-1}} \sum_{k=0}^{N-1} \|x_k - x_k^*\|_Q^2 + \|u_k\|_R^2 \quad (363)$$

離散ダイナミクスおよび力制約に従う。ここで x は状態、 u は制御入力、 Q, R は重み行列である。関節トルク限界を超えないように接触力 f の不等式制約を組み込む。

知覚統合は厳しい公差にとって重要である。深度カメラ、構造化光、触覚センサを組み合わせる。センサ融合は姿勢不確実性を削減する。視覚姿勢の共分散を Σ_v 、触覚推定の共分散を Σ_t とすると、融合共分散は

$$[H]\Sigma_f = (\Sigma_v^{-1} + \Sigma_t^{-1})^{-1}, \quad (364)$$

独立したガウス誤差を仮定。より低い Σ_f は挿入失敗確率を直接削減する。

代表的な精密挿入タスクの実装チェックリスト：

- ・ハードウェア：手首に力トルクセンサを備えたヒューマノイドアーム、パッシブコンプライアンスを備えた器用な手、高帯域 IMU、頭部に搭載された深度カメラ。
- ・ソフトウェア：低遅延制御のためのリアルタイムスタック（ROS2）、拘束軌道のための MPC ソルバ、不確実性出力を備えた視覚姿勢推定器、力ベースインピーダンスコントローラ。
- ・安全：保守的な力限界を設定し、人間共有ゾーンでは保護エンクロージャまたは動的速度限界を使用。

実用的な制御コードパターンは以下に続く。以下のリストは、視覚誘導インピーダンス挿入のための簡略化された ROS2 スタイルループを示す。コメントは主要なエンジニアリング動作を注釈する。

コードサンプル 148 視覚誘導インピーダンス挿入ループ（簡略化）。

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from geometry_msgs.msg import PoseStamped, WrenchStamped
from sensor_msgs.msg import JointState
import numpy as np
from typing import Optional, Tuple
from scipy.spatial.transform import Rotation as R

# 自作モジュール（同一パッケージ内に配置）
from control_api import ImpedanceController, MPCPlanner, VisionPose

class ImpedanceMPCNode(Node):
    """ROS2 ノード：視覚姿勢を使った MPC+インピーダンス制御"""

    def __init__(self) -> None:
        super().__init__('impedance_mpc_node')

        # QoS：リアルタイム優先
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        # パブリッシャ／サブスクライバ
```

```

self.cmd_pub = self.create_publisher(JointState, '/robot/joint_cmd', qos)
self.wrench_sub = self.create_subscription(
    WrenchStamped, '/robot/wrench', self.wrench_cb, qos)
self.joint_sub = self.create_subscription(
    JointState, '/robot/joint_states', self.joint_cb, qos)

# 制御器／プランナ
self.imp = ImpedanceController(
    stiffness=np.diag([800, 800, 800, 50, 50, 50]))
self.mpc = MPCPlanner(horizon=20, dt=0.02)

# 内部状態
self.latest_wrench = np.zeros(6)
self.latest_joint = JointState()

# タイマー：500 Hz
self.timer = self.create_timer(0.002, self.control_loop)

def wrench_cb(self, msg: WrenchStamped) -> None:
    self.latest_wrench = np.array([
        msg.wrench.force.x, msg.wrench.force.y, msg.wrench.force.z,
        msg.wrench.torque.x, msg.wrench.torque.y, msg.wrench.torque.z
    ])

def joint_cb(self, msg: JointState) -> None:
    self.latest_joint = msg

def current_state(self) -> np.ndarray:
    # 簡易：ジョイント位置をMPC状態へ変換
    return np.array(self.latest_joint.position)

def send_to_hw(self, u: np.ndarray) -> None:
    cmd = JointState()
    cmd.header.stamp = self.get_clock().now().to_msg()
    cmd.name = self.latest_joint.name
    cmd.effort = u.tolist()
    self.cmd_pub.publish(cmd)

def log_metrics(self, pose: np.ndarray, u: np.ndarray) -> None:
    self.get_logger().debug(

```

```

        f'pose={pose[:3]},cmd={u[:3]}', throttle_duration_sec=1.0)

def control_loop(self) -> None:
    pose_cam, cov = VisionPose.get_estimate()
    if cov.trace() > 1e-3:
        # 視覚信頼度低下 → 柔らかく
        self.imp.set_stiffness(reduce=True)

    goal = self.mpc.solve(self.current_state(), target=pose_cam)
    u = self.imp.compute_command(goal, self.latest_wrench)
    self.send_to_hw(u)
    self.log_metrics(pose_cam, u)

def main(args=None):
    rclpy.init(args=args)
    node = ImpedanceMPCNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計トレードオフおよび運用上の考慮事項：

- 精度対速度：剛性を増加させると位置誤差は減少するが、伝達衝撃およびリスクを上昇させる。タスククリティカル公差によって剛性を設定。
- ペイロードおよびリーチ：より大きいペイロード能力は質量および慣性を増加させ、動的応答およびエネルギー消費を損なう。
- センシング冗長性は不確実性を減少させるが、システム複雑性およびキャリブレーション負担を増加させる。
- ヒューマンコラボレーションは遅延および許容安全閾値を導入し、スループットを低下させる。

製造におけるヒューマノイドに固有の運用上のリスク：

- 力任務中の予期せぬ接触はバランスを不安定化させる可能性がある；トルクの再配分のための全

身制御を統合。

- ・ ツール変更および配線はエンドエフェクタ慣性を変更し、MPC 予測に影響を与える；ツール交換後に検証を実施。
- ・ 触覚表面の摩耗またはビジョンシステムのキャリブレーションずれは精度を低下させる；頻繁なキャリブレーションおよびヘルスチェックをスケジュール。

エンジニアはこれらのトレードオフを提供された方程式および制御パターンを用いて早期に定量化すべきである。測定可能メトリクスを優先： T_{cycle} 、 M_{reach} 、トルクマージン、および融合センサ共分散 Σ_f 。これらはアクチュエータ選択、制御設計、および高度な製造へのヒューマノイド展開のための安全エンベロープを導く。

44.3 倫理的および実践的な課題

先進製造におけるタスク固有の制約や、物流における人間・機械混在通路について述べた後、ここでは工場フロアにヒューマノイドを導入する際に重複する倫理的・実践的な課題を考察する。先行サブセクションでは精度・ペイロード・ナビゲーションを強調したが、ここでは運用の実現可能性に影響を与える故障モード、人間要因、ガバナンスを分析する。

ヒューマノイドの導入は、人間に似た形状が期待や責任を変えるため、独自の倫理的ベクトルを生む。主な倫理的懸念は以下の通りである：

- ・ 労働者の置き換えと経済的影響。自動化されたヒューマノイドは反復作業を代替し、労働需要をシフトさせる。タスク時間の自動化量と利用可能労働時間を見積もり、潜在的な置き換えを定量化する。
- ・ 監視とプライバシー。ヒューマノイドは視覚・音声センサを搭載することが多く、データ収集は従業員を暗黙的に監視し、法的・信頼の問題を生む。
- ・ 責任と賠償責任。知覚・計画・制御がベンダー・システムインテグレータ・雇用者に分散する場合、有害行為の責任主体を特定することは複雑である。
- ・ 外見に基づく信頼と欺瞞。人間に似た外見は不適切な信頼を誘発する可能性がある。システムは自律性や能力についてユーザを誤認させないように配慮しなければならない。

実践的な課題はこれら倫理的懸念を補強する。主な運用上の問題は以下の通りである：

- ・ 共有ワークスペースにおける安全性。ヒューマノイドは人間との安全な距離・接触力を維持しなければならない。基本的な安全制約は距離ベースで次のように表される： $d = \|p_{\text{robot}} - p_{\text{human}}\| \geq d_{\text{min}}$ ，ここで p は位置ベクトルを表す。ジョイントトルク制約も重要である：各アクチュエータは $|\tau_i| \leq \tau_{\text{max},i}$ を満たし、損傷・負傷を回避しなければならない。
- ・ 知覚故障モード。遮蔽、照明変化、反射面は偽陰性・偽陽性を生じ、安全でない計画や不要な停止を引き起こす。
- ・ 認証と検証。産業規格は決定的な安全事例を要求する。学習ベースのコンポーネントを含むヒューマノイドは形式的証明を複雑化する。
- ・ 保守性と稼働率。複雑なヒューマノイドは頻繁なキャリブレーション・センサ清掃・再調整を必要とし、ダウンタイムコストは生産性向上を打ち消す可能性がある。
- ・ サイバーセキュリティ。接続されたヒューマノイドは新たな攻撃対象面を露出させ、侵害により

物理的被害や知的財産の損失をもたらす。

簡潔なリスク指標が工学上のトレードオフを導く。リスクを時間当たりの期待損失として定義する： $R = P_{\text{fail}} \times S$ ，ここで P_{fail} は危険な故障の確率、 S は期待される深刻度である。技術者は故障確率を下げるか深刻度を抑制することのいずれかで両項を削減しなければならない。

緩和戦略は工学アクションに対応する：

1. 階層化安全アーキテクチャ：

- ハードウェア層：コンプライアントアクチュエータ、トルクリミット、機械的ガード。
- 低レベル制御：保証された受動性を持つインピーダンス制御。
- ミドルレベル：モデルベース衝突回避と制約付き最適化。
- 監視層：不変条件を強制するランタイム安全モニタ。

2. 知覚冗長性：

- モダリティ（LIDAR、ステレオビジョン、レーダー、IMU）を融合し単一センサ故障を削減。
- 信頼度指標を用い、低信頼度観測をソフト制約として扱う。

3. 人間中心インタフェース設計：

- 明示的意図シグナリング（ライト、ディスプレイ、触覚キュー）。
- 自律レベルの明確なラベル表示でユーザ期待を設定。

4. ガバナンスとデータ管理：

- 最小限のデータ保持；人間中心記録を匿名化。
- ロールベースアクセスと安全なテレメトリチャネル。

実装上、ランタイム保証はしばしば作動前に危険なコマンドを除去する軽量安全モニタを用いる。次の例示的 Python スニペットは、距離・トルク閾値をチェックし、リスクが閾値を超えた際に緊急停止を発行する簡易安全モニタを示す。

コードサンプル 149 ヒューマノイド導入用ランタイム安全モニタ

```
import numpy as np
import rclpy
from rclpy.node import Node
from std_msgs.msg import Bool
from typing import NamedTuple
```

```
class State(NamedTuple):
    p_robot: np.ndarray
    p_human: np.ndarray
    taus: np.ndarray
    p_fail: float
    severity: float
```

```

class SafetyMonitor(Node):
    def __init__(self,
                  d_min: float,
                  tau_max: float,
                  risk_threshold: float,
                  node_name: str = "safety_monitor"):
        super().__init__(node_name)
        self._d_min = d_min
        self._tau_max = tau_max
        self._risk_threshold = risk_threshold

        # 緊急停止指令パブリッシャ
        self._estop_pub = self.create_publisher(Bool, "/safety/emergency_stop", 1)

        # ログ
        self.get_logger().info(
            f"d_min={d_min:.3f}, tau_max={tau_max:.3f}, risk_threshold={risk_threshold:.3f}"
        )

    def compute_risk(self,
                    p_robot: np.ndarray,
                    p_human: np.ndarray,
                    taus: np.ndarray,
                    p_fail: float,
                    severity: float) -> float:
        d = np.linalg.norm(p_robot - p_human)
        # 距離要因: d_min未満なら危険度上昇
        dist_factor = max(0.0, (self._d_min - d) / self._d_min)
        # トルク要因: 正規化平均
        torque_factor = np.mean(np.abs(taus) / self._tau_max)
        # 統合リスク
        risk = p_fail * severity * (1.0 + dist_factor + torque_factor)
        return float(risk)

    def check_and_act(self, state: State) -> None:
        risk = self.compute_risk(
            state.p_robot,
            state.p_human,
            state.taus,
            state.p_fail,

```

```

        state.severity
    )
    if risk > self._risk_threshold:
        self.emergency_stop()
    else:
        self.allow_motion()

def emergency_stop(self) -> None:
    # 緊急停止フラグを送信
    msg = Bool()
    msg.data = True
    self._estop_pub.publish(msg)
    self.get_logger().error("Emergency_stop_triggered!")

def allow_motion(self) -> None:
    # 通常運転継続（何もしない）
    pass

```

設計者はこのようなモニタを最後の手段の保護として扱い、プライマリコントローラとはしない。センサ劣化下でフェイルセーフすることを検証しなければならない。

具体的なトレードオフと運用上のリスク：

- ・頑健性対コスト。冗長センサやコンプライアントアクチュエータを追加すると初期コストと重量が増加する。予算制約の設置では残留リスクを高く受け入れる可能性がある。
- ・透明性対知的財産。振る舞いモデルを開示すると監査に役立つが、独自アルゴリズムの露出リスクがある。
- ・自律レベル対人間監視。高自律は人間労働を削減するが、責任と認証の確立がより困難になる。
- ・性能対説明可能性。学習ベースコントローラはスループットで優れるが、形式的保証に抵抗する。

運用上、以下の工学アクションを優先する：

1. フィールドデータとシミュレーションを用い P_{fail} と S を定量化し、 R を継続的に更新する。
2. 物流・製造で判明した既知故障モードに対して知覚パイプラインを強化する。
3. 検証可能な低レベル振る舞いと認証済み緊急経路を持つ階層化安全を実装する。
4. センシング・ロギングのデータガバナンスポリシーを定義し、暗号化とアクセス制御を実施する。

これらの課題を無視すると、規制制裁、職場での負傷、公の信頼の喪失が生じる。工学の目的は、生産性向上を測定可能な残留リスクとバランスさせ、倫理的保護をシステムアーキテクチャと運用手順に組み込むことである。

デバッグとトラブルシューティング

45 機械的トラブルシューティング

45.1 一般的な機械故障の特定

機械的トラブルシューティングの概要に続き、本小節ではヒューマノイドロボットにおける最も一般的な機械故障を一覧し、それらを検出するための測定可能な特徴を示す。各故障には実用的なテスト、簡易モデル、およびオンロボット診断のための実装ノートが付随している。

問題設定：ヒューマノイドロボットは多数の結合した電気機械サブシステムを含む。わずかな機械的劣化がバランス損失、モータ電流増加、制御不安定化、安全リスクへと連鎖する。効果的なデバッグには、症状を根本原因にマッピングし、検出閾値を定量化し、継続的モニタリングのための低コストセンサを選定することが必要である。

列挙された一般的な故障、診断特徴、および緩和手順。

1. ジョイントバックラッシュおよび伝達遊び。
 - ・症状：位置応答にヒステリシス、指令と測定ジョイント角の間にデッドバンド、振動的な補正指令。
 - ・定量的テスト：ゆっくりとランプ指令を印加；残差 $r(t) = \theta_{\text{cmd}}(t) - \theta_{\text{meas}}(t)$ を測定。バックラッシュは符号依存遷移を伴う非零 r を示す。
 - ・緩和：ギアを交換するか、コンプライアントスプリングでプレロード；制御デッドバンドをチューニングしてハンチングを回避。
2. ベアリング摩耗および摩擦増加。
 - ・症状：同じトルク要求でモータ電流が上昇、低速でスティクション、過剰振動。
 - ・診断：モータ電流 I を期待トルクと比較、 $\tau = K_t I - \tau_f(\dot{\theta})$ 、ただし K_t はトルク定数、 τ_f は粘性およびクーロン摩擦をモデル化。
 - ・測定：電流対速度マップを使用；過剰トルク $\Delta\tau = \tau_{\text{meas}} - \tau_{\text{expected}}$ を算出。
3. ギア歯損傷および滑り。
 - ・症状：インパルスのトルクスパイク、可聴クリック音、負荷下での突発的位置ジャンプ。
 - ・診断：モータ電流または振動の高周波成分。加速度計または電流のFFTを使用して高調波を特定。
 - ・実装：スペクトルピークがベースラインを定義された係数で超えたら、視覚検査をスケジュールし、摩耗したギアセットを交換。
4. ケーブル伸び、シース摩耗、およびルーティング故障。
 - ・症状：断続的センサ読み値、作動遅延、エンコーダ読み値の進行性オフセット。
 - ・診断：静的負荷下で端から端のコンプライアンスを測定。過剰コンプライアンスはケーブル伸びまたはルーティング滑りを示す。
 - ・緩和：ケーブルを再張力、ストレインリリーフを追加、クリープの少ない材料に交換。
5. エンコーダ不整列および分解能損失。
 - ・症状：量子化ノイズ、ステップロスト、ジョイントゼロの不一致。
 - ・診断：エンコーダ対運動学的キャリブレーションを実施。順運動学で残差を算出；系統的バイアスは不整列を示唆。

- ・緩和：再キャリブレーション，ホーミングセンサの再インデックス，安全臨界ジョイントにはデュアル冗長エンコーダへアップグレード。
6. 構造疲労および微細亀裂.
- ・症状：モード周波数シフト，可視亀裂，コンプライアンスの進行性変化。
 - ・診断：制御励起を用いたモーダル解析．寿命を通じて固有周波数 f_n を追跡；著しい低下は弱体化を示す。
 - ・緩和：応力部材を交換し，メンテナンスログに有限寿命追跡を追加。
7. アクチュエータ過熱および巻線劣化.
- ・症状：トルク定数低下，断続的抵抗増加，サーマルトリップ。
 - ・診断：巻線抵抗 $R(T)$ を測定し，温度補正ベースラインと比較．熱老化は絶縁完整性を低下させることが多い。
 - ・緩和：サーマルセンサを追加，デューティサイクルをデレート，必要に応じて強制冷却を設計。
8. 締結具緩みおよびプレロード損失.
- ・症状：ガタ，変化するジョイント剛性，大振幅低周波振動。
 - ・診断：締結具をトルクチェック；ジョイントインピーダンステストを使用して剛性マトリクス成分の変化を検出。
 - ・緩和：スレッドロックを塗布，ロックワッシャを使用，または自己クランプインタフェースへ再設計。
9. ワイヤハーネス摩耗および断続的電気接触.
- ・症状：センサドロップアウト，スプリアスエンコーダジャンプ，EMI 増加。
 - ・診断：動作下での絶縁抵抗テストおよび導通チェック．ジョイント角と関連したエラーパターンをログ。
 - ・緩和：ハーネスを再ルーティング，耐摩耗スリーブを追加，接触冗長性を実装。
10. センサ取付けおよび整列ドリフト.
- ・症状：IMU 方位不正，バイアス付き姿勢推定，制御不安定化。
 - ・診断：ロボットが既知ポーズを実行中に自動キャリブレーションルーチンを実施．回転行列または四元数を用いて方位誤差を算出。
 - ・緩和：フィデューシャルベース再キャリブレーション，より優れた機械クランプ，または単一センサドリフトをマスクするマルチセンサ融合を使用。

信頼性モデリングはメンテナンス計画を支援する．Weibull 信頼性関数を使用して経時的故障確率を推定：

$$[H]R(t) = \exp\left(-\left(\frac{t}{\lambda}\right)^k\right) \quad (365)$$

ただし λ は特徴寿命， k は形状パラメータである． λ, k をログされた故障時刻から推定し，点検間隔を最適化する．

実用的なオンロボ検出コード：持続的な位置または電流異常を持つジョイントをフラグ付けする軽量残差ベースモニタ．

コードサンプル 150 Example: joint-failure detector using residuals

```

import numpy as np
from typing import Dict, List, Optional

# 閾値と履歴長は外部から上書き可能にする
DEFAULT_WINDOW: int = 100          # 統計計算に使うサンプル数
DEFAULT_TH_POS: float = 0.02        # 位置残差閾値 [rad]
DEFAULT_TH_CURR: float = 0.5        # 電流異常閾値 [A]
DEFAULT_TH_NOISE: float = 0.01      # 標準偏差ベースライン [rad]

class JointFailureDetector:
    """
    関節の位置指令・実測・電流履歴から簡易故障特徴を抽出する
    """

    def __init__(
        self,
        window: int = DEFAULT_WINDOW,
        th_pos: float = DEFAULT_TH_POS,
        th_curr: float = DEFAULT_TH_CURR,
        th_noise: float = DEFAULT_TH_NOISE,
    ) -> None:
        self.window: int = window
        self.th_pos: float = th_pos
        self.th_curr: float = th_curr
        self.th_noise: float = th_noise

    def detect(
        self,
        cmd_pos_hist: List[float],
        meas_pos_hist: List[float],
        curr_hist: List[float],
    ) -> Dict[str, bool]:
        """
        最新の履歴を受け取り、故障フラグを返す
        """
        # 履歴不足なら全フラグFalseで早期リターン
        if len(cmd_pos_hist) < self.window:
            return {"backlash_like": False, "friction_like": False, "intermittent": False}

```

```

# 残差と統計量を計算
pos_res = np.array(cmd_pos_hist[-self.window :]) - np.array(
    meas_pos_hist[-self.window :])
)
pos_mean: float = float(np.mean(pos_res))
pos_std: float = float(np.std(pos_res))
curr_mean: float = float(np.mean(curr_hist[-self.window :]))

# 各故障モードの判定
flags: Dict[str, bool] = {
    "backlash_like": abs(pos_mean) > self.th_pos and pos_std < self.th_pos,
    "friction_like": curr_mean > self.th_curr and pos_std < self.th_noise,
    "intermittent": pos_std > 3.0 * self.th_noise,
}
return flags

```

工学への影響，トレードオフ，および運用上のリスク：

- ・センサ追加は検出を改善するが，重量，電力，配線複雑性を増加させる．
- ・予防交換は予期しない停止時間を減らすが，部品および労働コストを上昇させる．
- ・モニタリングアルゴリズムは，不要なミッション中止を回避するため，誤検出と見逃し故障のバランスを取る必要がある．
- ・臨界ジョイントは冗長性，サーマルモニタリング，振動センシングを採用し，安全リスクを低減すべきである．

設計トレードオフには，センサカバレッジ，Weibull 推定寿命に基づく点検周期，および進行性機械劣化に対する制御頑健性が含まれる．

45.2 保守のベストプラクティス

前述の一般的な機械的故障の特定を踏まえ、これらの保守ベストプラクティスは再発防止と運用ダウンタイムの最小化に焦点を当てている。本指針は、予防および予測保守、実用的な検査指標、予備品戦略、ソフトウェア診断との統合を重視する。

問題提起：ヒューマノイドロボットは多くの電気機械サブシステムを組み合わせている。アクチュエータ、ギアボックス、ハーモニックドライブ、ケーブル、ベアリング、センサが可変負荷下で相互作用する。構造化された保守プログラムなしでは、小さな劣化がミッションを終了させる故障に連鎖する。目標は、可用性を維持しながらライフサイクルコストを抑制する、繰り返し可能なアクションと自動チェックを定義することである。

技術的分析と実施可能な手順：

- ・保守カテゴリと適用タイミング：
 1. 予防保守：経過時間またはサイクル数に基づく定期潤滑、トルク再確認、再較正。
 2. 予測保守：テレメトリトレンドと異常検知によってトリガされる状態ベースのアクション。

3. 修正保守：診断が故障コンポーネントを特定した際の計画修理。
 4. 信頼性中心保守（RCM）：安全性、ミッション影響、故障モードによってタスクを優先付け。
- 確率的観点からの検査間隔の選択。定常ハザード率 λ でモデル化されたメモリレス故障に対して、平均故障間隔（MTBF）は

$$\text{MTBF} = \frac{1}{\lambda}.$$

検査間の故障確率を P_{\max} に制限するため、次を満たす検査間隔 t_i を選ぶ

$$[H]1 - e^{-\lambda t_i} \leq P_{\max} \Rightarrow t_i \leq -\frac{\ln(1 - P_{\max})}{\lambda}. \quad (366)$$

単一ランの仮定ではなく、フリーテレメトリからの実証的 λ 推定値を使用する。

- ヒューマノイド向け状態監視指標：
 - トルク過負荷検出のためのモータ電流 RMS と積分値。
 - ベアリング劣化のためのジョイントハウジング振動 RMS。
 - バックラッシュおよびカップリング摩耗のためのエンコーダジッターとステップロス率。
 - 潤滑および絶縁故障のためのジョイント温度トレンドと熱サイクル。
 - 断続的配線故障のためのケーブルハーネスフレックスサイクルと絶縁抵抗試験。
- 閾値設定と異常検知：
 - コミッショニング時および制御試験プロファイルでベースラインを確立。
 - $v_{\text{rms}} > v_{\text{baseline}} + k\sigma_{\text{baseline}}$ のような正規化閾値を用いて初期ベアリング摩耗をフラグ付け。
 - 複数の指標を統合してスケジューリング判断のための複合健康指標 $H \in [0, 1]$ を作成。

設計および実装チェックリスト：

1. 計測とベースライン取得：
 - クリティカルジョイントにテストポイント、サーミスタ、IMU、電流センサを追加。
 - 既知負荷下でベースラインシグネチャを記録。
2. 自動データパイプライン：
 - テレメトリを時系列データベースにストリーム。
 - 軽量異常検知器を実行するエッジコンピュータノードを実装。
3. モジュールハードウェア設計：
 - クイックリリースコネクタ付きモジュールジョイントカートリッジを使用して迅速な交換を実現。
 - ファスナートルクを標準化し、トルク制限ツールを使用して締め付け過ぎを回避。
4. 予備品および消耗品戦略：
 - ベルト、シール、ハーモニックドライブコンポーネントなど摩耗品を在庫。
 - MTBF と運用ペースをキーとした部品交換リードタイムマトリクスを維持。
5. 手順およびトレーニング：
 - フィールド技術者向けに簡潔なチェックリストを作成。
 - 保守中のエラーを削減するためキャッシュ診断スクリプトを使用。

保守アクションのための主要な工学公式：- ボルト締結において、プレロード F を達成するために必要な概算組立トルク T は

$$[H]T = KFd, \quad (367)$$

ここで、 K はナット係数、 d は公称ボルト径である。コーティングおよび潤滑のために K を実証的に検証する。

- 監視指標 $m(t)$ の単純な指数劣化（劣化率 α ）に対して、閾値 m_{thr} 到達時刻を予測するには

$$[H]m(t) = m_0 e^{\alpha t} \Rightarrow t_{\text{to_thr}} = \frac{\ln(m_{\text{thr}}/m_0)}{\alpha}. \quad (368)$$

実用的モニタリングスクリプト例（ROS2 テレメトリ取り込みおよび検査スケジューリング）：

コードサンプル 151 モータテレメトリを読み取り検査間隔を計算する保守スケジューラ例

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Float32MultiArray
import math
import time
from typing import List, Optional
from threading import Lock

class MaintenanceNode(Node):
    def __init__(self) -> None:
        super().__init__('maintenance_node')

        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self._sub = self.create_subscription(
            Float32MultiArray, '/telemetry/motors', self._telemetry_cb, qos)

        self._failures: List[float] = []
        self._lock = Lock()
        self._last_fail_t: Optional[float] = None
        self._current_t: float = 0.0

        # 閾値はパラメータで変更可能
        self.declare_parameter('current_threshold', 50.0)
        self.declare_parameter('p_max', 0.01)
        self.declare_parameter('min_failures', 2)
```

```

        self._timer = self.create_timer(1.0, self._publish_inspection_interval)

def _telemetry_cb(self, msg: Float32MultiArray) -> None:
    threshold = self.get_parameter('current_threshold').value
    if max(msg.data) > threshold:
        with self._lock:
            now = time.monotonic() / 3600.0  # 時間単位
            if self._last_fail_t is None or (now - self._last_fail_t) > 1e-3:
                self._failures.append(now)
                self._last_fail_t = now

def _publish_inspection_interval(self) -> None:
    with self._lock:
        if len(self._failures) < self.get_parameter('min_failures').value:
            return
        intervals = [self._failures[i+1] - self._failures[i]
                     for i in range(len(self._failures)-1)]
        mtbf = sum(intervals) / len(intervals)
        lam = 1.0 / mtbf
        p_max = self.get_parameter('p_max').value
        t_i = -math.log(1.0 - p_max) / lam
        self.get_logger().info(f'Inspection_interval: {t_i:.2f}h')

def main(args=None):
    rclpy.init(args=args)
    node = MaintenanceNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

運用上の考慮事項、トレードオフ、リスク：

- 計測を増やすと予測精度が向上するが、コストと配線複雑性が増大する。センシング密度と故障影響を釣り合う。
- 検査間隔を短くするとリスクは低下するが、ダウンタイムと保守労働力が増加する。
- 閾値を過度に厳しくすると誤検知が増え、不要な部品交換とコストが増大する。
- 保守性（モジュール性、アクセス）を考慮した設計は、初期コストと質量を犠牲にしてライフサ

イクルコストを低減する。

- ・保守ソフトウェアが認証済みテレメトリを使用し、侵害されたセンサからの誤ったアラートを回避するようにする。

具体的な影響：λ 推定値を洗練するためフリートテレメトリ集約を実装する。MTBF が低くミッション影響が大きいコンポーネントの予備品在庫を優先する。技術者に機械手順とテレメトリ解釈の両方をトレーニングし、介入時のヒューマンエラーを削減する。

45.3 ケーススタディ：アクチュエータ故障の修理

以下のケーススタディは、前述の保守プロトコルと故障モード分類を基に、予防チェックリスト項目とベアリング摩耗やエンコーダドリフトなどの一般的な症状を出発点として使用する。ヒューマノイド膝関節アクチュエータのアクチュエータ故障を切り離し、診断し、修理する段階的なエンジニアリングプロセスを示す。

問題文：ヒューマノイドの右膝アクチュエータが低速立ち上がりタスク中に指令トルクを断続的に保持できない。症状として、負荷下での小さな位置ドリフト、低速動作時の可聴グラインド音、モータドライバの過渡電流スパイク、およびギアボックス表面温度の上昇が挙げられる。エンジニアリング目標は、故障が機械的、電気的、または制御関連のいずれであるかを特定し、安全な動作を復元する最小侵襲修理を選択することである。

技術分析

- ・検討すべき仮説：

1. バックドライバビリティと位置ドリフトを引き起こすギアボックス摩耗または歯損傷。
2. クーロン摩擦を増加させグラインド音を発生させるベアリング故障。
3. 誤ったフィードバックを引き起こすエンコーダ不整合または信号劣化。
4. 電流スパイクを引き起こすモータ位相不平衡またはドライバ整流エラー。

- ・関連物理モデル。観測電流をトルクおよび摩擦に関連付けるために、簡約化ロータ＋負荷ダイナミクスモデルを使用：

$$[H]J\ddot{\theta} + b\dot{\theta} + \tau_{\text{fric}}(\dot{\theta}) + \tau_{\text{load}} = \tau_{\text{motor}} = K_t I \quad (369)$$

ここで J は結合慣性、 b は粘性減衰、 τ_{fric} はクーロンおよび速度依存摩擦をモデル化、 K_t はモータトルク定数、 I はモータ電流である。これを制御試験中の電流および測定運動学から摩擦を推定するために使用。

- ・定量的チェック：

1. トルクマージン： $\tau_{\text{available}} - \tau_{\text{expected_load}} \geq \text{安全マージン}$ を確保。 $\tau_{\text{available}} = K_t I_{\text{limit}}$ を計算。マージンが小さい場合、ギアボックス劣化が動作を制限している可能性。
2. バックラッシュ測定：低速双方向位置ランプを指令しヒステリシスループ面積を測定。過度のヒステリシスは歯車遊びまたはフレキシブルカップリングを示す。
3. ジッタ周波数解析：位置誤差のFFTを実行し共振ピークを見つける。ギアメッシュ周波数およびベアリング欠陥周波数は明確なスペクトルラインとして現れる。

実装：実用的診断ステップ

1. 安全および切り離し

- ・電源を遮断し、四肢を安全な姿勢で機械的に保持。
 - ・モータ電流制限を有効にし、ソフトウェアインターロックを設定。
2. その場での非侵襲試験
 - ・エンコーダ健全性：小さな正弦波速度を指令し、高速エンコーダカウントを予想される整流角度と比較。量子化ステップまたはパルス欠落を探す。
 - ・電流対トルクステップ試験：制御されたトルクステップを適用し、モータ電流および関節位置を高サンプルレート（ ≥ 1 kHz）で記録。準静的モデルを使用して摩擦を推定。
 - ・サーモグラフィ：低負荷連続運動中にギアボックスハウジングをスキャンし、摩擦発熱を示すホットスポットを局在化。
 3. 機械的寄与の切り離し
 - ・モータ無効時のバックドライブ試験：手動で関節をバックドライブし、なめらかさを感じ、局所的な結束ポイントを検出。
 - ・エンドスコープ検査：サービスポートを通じてギアボックスシールおよび歯車歯の損傷または破片を検査。
 4. 電氣的チェック
 - ・オシロスコープを使用してモータ位相電流および電圧を測定し、整流不規則性を検出。
 - ・ドライバテレメトリで過電流イベントを測定し、モーションログとタイムスタンプを比較。
 5. 制御ベンチ試験（ハードウェア取外しが必要な場合）
 - ・アクチュエータアセンブリを取外し、トルク試験ベンチで試験。既知の負荷下で効率、ヒステリシス、およびトルクリップルを測定。

診断メトリクスおよび閾値（例）

- ・クーロン摩擦推定値 $\tau_c \approx K_t(I_{\text{hold_pos}} - I_{\text{no_load}})$ 。
- ・膝関節の許容バックラッシュ：通常精密バランスタスクで $< 0.5^\circ$ 。大きい値はギアボックス交換またはプリロードを要する。
- ・温度上昇：小さな周期運動 10 分以内に周囲温度より $> 30^\circ\text{C}$ 上昇は内部結束または潤滑故障を示す。

コード例：ROS 風コマンドインターフェースを使用した自動トルクステップ診断。スクリプトはトルクステップを指令し、後解析のために電流およびエンコーダ値を記録。

コードサンプル 152 トルクステップ診断スクリプト（簡略化）

```
#!/usr/bin/env python3
import time
import numpy as np
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float64
from sensor_msgs.msg import JointState
from typing import List, Tuple
```

```

# 仮想APIのROS 2ラッパー
class KneeDriver(Node):
    def __init__(self) -> None:
        super().__init__('knee_driver')
        self._cmd_pub = self.create_publisher(Float64, '/right_knee/command', 1)
        self._state_sub = self.create_subscription(
            JointState, '/right_knee/state', self._state_cb, 1)
        self._latest_state: JointState = JointState()

    def _state_cb(self, msg: JointState) -> None:
        self._latest_state = msg

    def command_torque(self, tau: float) -> None:
        self._cmd_pub.publish(Float64(data=tau))

    def read_state(self) -> Tuple[float, float, float]:
        # pos[rad], vel[rad/s], current[A]
        return (self._latest_state.position[0],
                self._latest_state.velocity[0],
                self._latest_state.effort[0])

# 定数
K_T: float = 0.08          # Nm/A トルク定数
SAMPLE_HZ: int = 1000
STEPS: List[float] = [0.0, 2.0, -2.0, 0.0] # Nm
STEP_DURATION: float = 2.0 # s

def main() -> None:
    rclpy.init()
    driver = KneeDriver()

    log: List[Tuple[float, float, float, float, float, float]] = []

    for tau_cmd in STEPS:
        driver.command_torque(tau_cmd)
        t0 = time.perf_counter()
        while (time.perf_counter() - t0) < STEP_DURATION:
            rclpy.spin_once(driver, timeout_sec=0.0)
            pos, vel, cur = driver.read_state()
            est_tau = K_T * cur

```

```

log.append((time.perf_counter(), tau_cmd, pos, vel, cur, est_tau))
time.sleep(1.0 / SAMPLE_HZ)

driver.command_torque(0.0) # 安全停止
np.save('torque_step_log.npy', np.array(log, dtype=np.float64))
rclpy.shutdown()

if __name__ == '__main__':
    main()

```

解釈および修理判断ツリー

- ・エンコーダノイズまたはカウント欠落が存在する場合、エンコーダ交換または再着座を最優先。誤カウントは機械故障を模倣できる。
- ・クーロン摩擦推定値が温度とともに著しく上昇し、サーモグラフィでホットスポットが示される場合、ベアリングおよび潤滑を検査。摩耗破片が見えればベアリング交換。
- ・バックラッシュが許容値を超え、検査で歯車歯摩耗が示される場合、ギアボックス交換またはプリロード済みハーモニックドライブへのアップグレード。
- ・位相電流に整流スパイクが見られるが機械試験がなめらかな場合、ドライバファームウェア、ホール／エンコーダアライメント、またはモータ巻線問題を疑う。

エンジニアリング影響、トレードオフ、および運用上リスク

- ・ギアボックス交換は精度を回復するが、ダウンタイムおよびコストを増加。部品待ち中は制御ゲインを一時的に調整しトルク要求を削減。
- ・機械摩擦を隠蔽するため制御ゲインを締めると、モータ過熱および電気故障のリスク。電流制限および熱保護を使用。
- ・低バックラッシュアクチュエータへのアップグレードはバランス制御を改善するが、機械カップリングおよび制御ゲインの再設計を要する可能性。
- ・運用上リスク：診断されていない断続的アクチュエータ故障は転倒を引き起こし、ハードウェアを損傷し人を危険に晒す。修理完了まで保守的な安全マージンおよび劣化運転モードを実装。

この方法的アプローチは、非侵襲診断、定量的モデリング、およびベンチ試験を優先し、安全なヒューマノイド動作を復元する最小侵襲修正措置を選択する。

46 ソフトウェアデバッグ

46.1 ROS および GROOT を用いたデバッグ

機械的な原因を切り分けた後は、アクチュエータを指令しセンサを解釈するソフトウェアスタックにデバッグの焦点を移す。ハードウェア検査で確立されたパターンとタイミングは、どの ROS トピックと GROOT ビヘイビアツリーのティックを詳細に計測すべきかを指し示す。

問題提起と運用上の意義：ヒューマノイドロボットは高速フィードバックループ、知覚パイプライン

ン、意思決定レベルのビヘイビアツリーを組み合わせる。障害は遅延またはドロップした指令、不整合なフレーム変換、ビヘイビアツリーの状態振動として顕在化する。エンジニアリングゴールは、ライブで安全臨界なループを妨げることなく欠陥を局所化し、エンドツーエンドのタイミングを計測し、メッセージとブラックボードエントリの意味的正しさを検証することである。

技術的分析

• 考慮すべき故障モード：

1. メッセージ型の不一致とトピック上の静かなデシリアライズエラー。
2. 誤った frame_id または遅延した/tf 更新によって引き起こされる不整合な座標フレーム。
3. タイミング違反：デッドラインのミス、キューオーバーフロー、過度のジッタ。
4. ビヘイビアツリーの非決定性：ティックの競合、共有ブラックボード競合、不適切な前提条件。
5. リソース枯渇：CPU、メモリ、ネットワーク飽和によるドロップまたはセグフォルト。

定量的診断：エンドツーエンド遅延予算を確立し、計測された遅延に対して検証する。遅延を

$$[H]L_{\text{total}} = L_{\text{sense}} + L_{\text{comm}} + L_{\text{proc}} + L_{\text{act}} + J, \quad (370)$$

と分解し、 J はジッタを表す。ヒューマノイド足首トルクリープでは、許容される L_{total} は数百マイクロ秒オーダーである。高レベルの歩行切替では、許容遅延は数十ミリ秒である。

単一コアでのスケジューラビリティについては、利用率の境界

$$[H]U = \sum_i \frac{C_i}{T_i} \leq 1, \quad (371)$$

を検証し、 C_i は最悪実行時間、 T_i は周期タスクの周期である。 $U > 1$ であれば、システムは確定的にデッドラインをミスする。

実用的計測・検証戦略

1. ベースライントレースを確立：

- センサに合わせた周波数で同期された ROS（または ROS2）bag を記録する。/tf、IMU、ジョイント状態、およびすべてのビヘイビアツリー状態トピックを含める。
- システムレベル指標をキャプチャ：CPU、ネットワーク、プロセス RSS、オープンファイルディスクリプタ。

2. ビヘイビアツリーを計測：

- GROOT でティックログとトレースビューを有効にする。ノード入退場時刻と返却状態（RUNNING、SUCCESS、FAILURE）をログする。
- 重要なブラックボードキーを ROS トピックに公開し、BT 状態をセンサイバントと関連させる。

3. 遅延源を局所化：

- タイムスタンプ付きメッセージを用いて L_{comm} と L_{proc} を計測する。
- タイムスタンプ付きハートビートトピックをエコーして往復時間を計測する。

コード例：ROS2 ハートビート遅延計測。このノードはタイムスタンプ付きハートビートを公開し、往復エコーサブスクリバで計測された遅延をログする。歩行またはマニピュレーションタスク中にネットワークおよびノードスケジューリング動作を検証するために使用する。

コードサンプル 153 ROS2 ハートビート遅延計測 (rclpy)。

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
from std_msgs.msg import Header
import time
import argparse

class HeartbeatNode(Node):
    def __init__(self, hz: int = 100):
        super().__init__('heartbeat')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=10,
            durability=DurabilityPolicy.VOLATILE
        )
        self.pub = self.create_publisher(Header, 'heartbeat', qos)
        self.sub = self.create_subscription(Header, 'heartbeat_echo', self.on_echo, qos)
        self.timer = self.create_timer(1.0 / hz, self.send)
        self.declare_parameter('frame_id', 'heartbeat')
        self.frame_id = self.get_parameter('frame_id').value
        self.get_logger().info(f'Heartbeat running at {hz} Hz')

    def send(self):
        msg = Header()
        msg.stamp = self.get_clock().now().to_msg()
        msg.frame_id = self.frame_id
        self.pub.publish(msg)

    def on_echo(self, msg: Header):
        now_ns = self.get_clock().now().nanoseconds
        sent_ns = msg.stamp.sec * 1_000_000_000 + msg.stamp.nanosec
        latency_ms = (now_ns - sent_ns) / 1e6
        self.get_logger().info(f'RTT: {latency_ms:.3f} ms')

def main(args=None):
    rclpy.init(args=args)
    node = HeartbeatNode()
```

```

try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

ROS と GROOT 間の統合技術

- ROS アクション、サービス、トピックを GROOT ブラックボードキーに変換する薄いアダプタノードを使用する。これにより密結合を回避し、ビヘイビアツリーのブラックボックステストを可能にする。
- ログを時刻同期：複数マシンを使用する場合、すべてのプロデューサが共通クロックまたは同期 NTP/Ptp を使用するよう確実にする。
- IsaacSim でテストする際、ブラックボードエントリをシミュレーションログにミラーし、記録されたセンサノイズで再生して障害を再現する。

段階的デバッグチェックリスト

1. 健全性チェック：トピックが存在し、正しいメッセージ型、パブリッシャ／サブスクライバ数。
2. ランタイムグラフを可視化 (*rqt_graph* または *ros2 topic info*) して接続を確認。
3. 可能な限りシミュレーションで確定的に再現し、記録された bag と同じビヘイビアツリーティックペースを使用。
4. 漸進的隔離：高速知覚ノードを無効にし、制御ループの安定性をチェック。
5. ハードウェアインザループ漸増：低トルク、低速動作、次に障害が再発するまで増加。

設計トレードオフと運用上のリスク

- 計測オーバヘッド対観測可能性：冗長ログと高頻度ハートビートはタイミングを攪乱する。高忠実度トレースは制御されたテストランでのみ有効にする。
- ビヘイビアツリーの粒度：細かいノード分解は推論を簡素化するが、ティックオーバヘッドとブラックボード競合を増加させる。
- ネットワーク対ローカル処理：知覚のオフロードはオンボード CPU 負荷を減らす、 L_{comm} を増加させネットワーク故障モードを導入する。
- ウォッチドッグまたはフェイルセーフモニタが欠けると、ソフトウェア障害がヒューマノイドを安全でない状態に遷移させる。緊急停止のために独立したハードウェアまたは RTOS ウォッチドッグを含める。

具体的含意：設計段階で計測可能な遅延および利用率予算を確立する。これらの予算を最悪負荷下でフィールド配備前に検証する。そうしないことは、動的タスクを実行するヒューマノイドにとって

主要な運用上リスクである。

46.2 ロボティクスにおける一般的なソフトウェアの落とし穴

前の小節では、トピック、ノードライフサイクル、およびビヘイビアツリーの障害をトレースするための実用的な ROS および GROOT テクニックを示した。それらの診断を基に、本節では人型プラットフォームで巧妙かつ断続的な障害を引き起こす繰り返し発生するソフトウェアの落とし穴をカタログ化し、開発およびフィールド展開中に適用できる正確なエンジニアリング修正を示す。

問題提起：人型ロボットは異なるレートで多くの相互作用するソフトウェアコンポーネントを実行する。小さなソフトウェアのミスがバランス損失、アクチュエータ損傷、または安全でない動作に連鎖する。以下の焦点領域はセンシング、制御ループ、プロセス間通信、および運用上の安全性に結びついている。

一般的な落とし穴と技術分析：

- 競合状態と非決定的順序。

- 症状：2つのコールバックが共有状態を更新するときに断続的な制御スパイク。
- 根本原因：コールバックが適切な並行プリミティブなしに共有メモリを使用するか、リアルタイムコールバック内でブロッキング呼び出しを使用する。
- 修正：ロックフリーデータ転送またはリアルタイムセーフな単一ライター、複数リーダーバッファを使用する。ROS2 では、決定的エグゼキュータポリシーを持つコールバックグループを優先する。

- 遅延蓄積と位相余裕損失。

- 症状：振動的な歩容または全身制御での位相ラグ。
- 分析：全フィードバック遅延 τ は周波数 ω で位相ラグ $-\omega\tau$ を導入する。ループ交差周波数 ω_c では、位相余裕 ϕ_m を維持する：

$$[H]\tau < \frac{\phi_m}{\omega_c}. \quad (372)$$

- エンジニアリングの含意：コンポーネントごとの遅延（センサ取得、シリアルライゼーション、ネットワーク、デシリアルライゼーション、コントローラ計算、アクチュエータコマンド）を測定する。 $\omega_c = 2\pi \cdot 5 \text{ rad/s}$ および $\phi_m = 30^\circ = \pi/6$ の場合、 $\tau < (\pi/6)/(2\pi \cdot 5) = 1/60 \text{ s} \approx 16.7 \text{ ms}$ 。

- サンプリングミスマッチとエイリアシング。

- 症状：高周波振動が低周波信号にエイリアスし、誤った推定を引き起こす。
- ルール： $f_s > 2f_{\max}$ （ナイキスト）でサンプリングする。200 Hz までの内容を持つ IMU 信号の場合、 $f_s \geq 500 \text{ Hz}$ およびアンチエイリアスフィルタを使用する。

- 時刻とフレームの不整合。

- 症状：歩行プランナと状態推定器が不一致；TF ルックアップが失敗する。
- 根本原因：デバイス間で同期されていないクロック、古いメッセージタイムスタンプ、または不整合したフレーム命名。
- 修正：単一の時刻ソースを強制し、利用可能な場合はセンサにハードウェア時刻を使用し、しきい値より古いメッセージをドロップし、標準的なフレーム命名を採用する。ROS では、使用前に `message header.stamp` をシステムクロックと比較する。

- センサ融合における共分散の誤指定。

- 症状：フィルタがノイズの多いセンサを過度に信頼し、大きな推定誤差を生成する。
- 分析：カルマン型フィルタは逆共分散でセンサに重みを付ける。ノイズを過小評価すると融合状態にバイアスがかかる。
- 修正：経験的にセンサノイズを推定し、共分散行列をチューニングする；残差ホワイトニングテストで検証する。
- クリティカルスレッドでのブロッキング操作。
 - 症状：制御サイクルのミスとウォッチドッグトリップ。
 - 根本原因：ディスク I/O、長い計算、または制御スレッドでの同期 RPC。
 - 修正：ロギングと非クリティカル I/O を別スレッドまたはプロセスにオフロードする；ログ書き込みレートを制限する。
- メモリリークとリソース枯渇。
 - 症状：長いミッションでの遅いメモリ増加、最終的なノードクラッシュ。
 - 検出：サニタイザー、valgrind、および耐久テスト中のヒーププロファイリングを使用する。
 - 修正：タイトループ内の隠れた割り当てを避ける；ストリーミングデータ用にバッファを再利用する。
- 誤った単位と座標規約。
 - 症状：コントローラコマンドが度単位を使用するがアクチュエータはラジアンを期待し、大きなトルクコマンドを生成する。
 - 緩和：メッセージ境界で単位をアサートし、表現間の変換に単体テストを追加する。
- 過負荷通信チャンネルと QoS 誤設定。
 - 症状：高センサスループット下でのメッセージドロップが古い状態につながる。
 - 修正：各トピックに適切な QoS 深度と信頼性を選択する。高レートセンサには帯域幅スロットリングとメッセージ集約を使用する。
- 安全インターロックをバイパスする安全でないエラー処理。
 - 症状：コントローラがセンサタイムアウトを無視し、アクチュエータをコマンドし続ける。
 - 修正：明示的な安全フォールバック状態を実装し、ウォッチドッグを検証し、クリティカルフォルト時に低エネルギー姿勢にフェイルする。

実装例と防御パターン：

1. 処理前にタイムスタンプチェックで古いメッセージをドロップする。
2. 無制限バッファの代わりに有界キューと背圧を使用する。
3. 遅延、ジッター、およびメッセージ損失を追跡するためにループごとの診断を維持する。
4. 回帰テストを自動化する：シミュレートされた接触遷移、長時間メモリプロファイル、およびハードウェアインザループ安全テスト。

ROS2 パターンを示す実用的なコード例：時刻チェック、有界キュー、および IMU と関節状態の近似時刻同期。

コードサンプル 154 ROS2 ノードスニペット：タイムスタンプチェック、有界 QoS、メッセージ同期

```
import rclpy
from rclpy.node import Node
```

```

from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from rclpy.duration import Duration
from rclpy.time import Time
from sensor_msgs.msg import Imu, JointState
from geometry_msgs.msg import TwistStamped
from message_filters import ApproximateTimeSynchronizer, Subscriber
import numpy as np
from typing import Optional, Tuple

class SensorFusionNode(Node):
    def __init__(self) -> None:
        super().__init__('sensor_fusion')

        # QoS: 信頼性を保ちつつリアルタイム性を重視
        qos = QoSProfile(
            depth=10,
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST
        )

        # サブスクライバ
        self.imu_sub = Subscriber(self, Imu, '/imu', qos_profile=qos)
        self.joint_sub = Subscriber(self, JointState, '/joint_states', qos_profile=qos)

        # 同期器: スラック 10 ms, キューサイズ 50
        self.sync = ApproximateTimeSynchronizer(
            [self.imu_sub, self.joint_sub],
            queue_size=50,
            slop=0.01
        )
        self.sync.registerCallback(self.fused_callback)

        # パブリッシャ: 融合後の速度推定値を配信
        self.fused_pub = self.create_publisher(TwistStamped, '~/fused_twist', qos)

        # パラメータ
        self.declare_parameter('max_staleness_s', 0.05)
        self.max_staleness = Duration(seconds=self.get_parameter('max_staleness_s').value)

```

```

# キャリブレーション用バッファ
self.gyro_bias: Optional[np.ndarray] = None
self.calib_count = 0
self.calib_samples = 200

self.get_logger().info('SensorFusionNode initialized')

def fused_callback(self, imu: Imu, joints: JointState) -> None:
    now = self.get_clock().now()

    # 時刻チェック: 古いメッセージは破棄
    imu_time = Time.from_msg(imu.header.stamp)
    if (now - imu_time) > self.max_staleness:
        return

    # キャリブレーション
    if self.gyro_bias is None:
        self._calibrate_gyro(imu)
        return

    # バイアス除去後の角速度
    omega = np.array([
        imu.angular_velocity.x,
        imu.angular_velocity.y,
        imu.angular_velocity.z
    ]) - self.gyro_bias

    # ホイール半径・間隔のパラメータ (仮)
    wheel_radius = 0.08 # [m]
    wheel_base = 0.4 # [m]

    # ジョイント名から左右ホイール速度を抽出
    left_vel = right_vel = 0.0
    for name, vel in zip(joints.name, joints.velocity):
        if 'left' in name:
            left_vel = vel
        elif 'right' in name:
            right_vel = vel

    # 差動二輪モデルで速度推定

```

```

v_left = left_vel * wheel_radius
v_right = right_vel * wheel_radius
v = (v_left + v_right) * 0.5
w = (v_right - v_left) / wheel_base

# 融合：角速度はIMU優先，並進速度はエンコーダ優先
fused_w = omega[2] if abs(omega[2]) > 0.01 else w

# パブリッシュ
twist = TwistStamped()
twist.header.stamp = imu.header.stamp
twist.header.frame_id = 'base_link'
twist.twist.linear.x = float(v)
twist.twist.angular.z = float(fused_w)
self.fused_pub.publish(twist)

def _calibrate_gyro(self, imu: Imu) -> None:
    if self.calib_count == 0:
        self.get_logger().info('Calibrating_gyro_bias...')
        self.gyro_buffer: list = []

    self.gyro_buffer.append([
        imu.angular_velocity.x,
        imu.angular_velocity.y,
        imu.angular_velocity.z
    ])
    self.calib_count += 1

    if self.calib_count >= self.calib_samples:
        self.gyro_bias = np.mean(self.gyro_buffer, axis=0)
        self.get_logger().info(f'Gyro_bias_calibrated:{self.gyro_bias}')
        self.gyro_buffer.clear()

def main(args=None):
    rclpy.init(args=args)
    node = SensorFusionNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:

```

```

        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

エンジニアリングの含意、トレードオフ、および運用上のリスク：

- トレードオフ：QoS 信頼性を上げると遅延が増加する；深いフィルタはノイズを減らすが遅延を増加させる。
- リスク：サンプリング、タイミング、または共分散について楽観的な仮定は、回復不能なバランス障害を引き起こす可能性がある。
- 推奨：遅延予算を計測し、タイミングおよびメッセージ処理の単体テストを強制し、ソフトウェアフォルト時に高エネルギーアクチュエータを無効にする安全フォールバック動作を設計する。

46.3 性能分析のためのツール

これまでの ROS および GROOT デバッグ、一般的なソフトウェアの落とし穴に関する議論を踏まえ、次のステップはターゲットを絞った性能分析である。本小節では、ヒューマノイドロボットソフトウェアスタックにおいて遅延、ジッター、CPU/GPU 負荷、エンドツーエンドタイミングを定量化するためのエンジニアリングワークフロー、測定、および具体的なツールを説明する。

性能分析の問題定義。ヒューマノイドロボットは厳格なタイミングと協調したマルチセンサ処理を要求する。重要な問いは以下の通りである：

- 制御ループは典型的および最悪負荷下でデッドラインを満たすか？
- エンドツーエンド遅延はセンサ取得からアクチュエータ指令までどこで発生するか？
- タスクごとの CPU/GPU 予算はどれくらいか、どのタスクが競合を引き起こすか？

これらに答えるには計測、軽量トレーシング、事後プロファイリングが必要である。

測定すべき運用指標。安全性と安定性に対応する測定可能な量を優先する：

- 各周期タスクの制御ループ周期 T および実行時間 C 。
- センサタイムスタンプからアクチュエータ出力までのエンドツーエンド遅延 L 。
- ループ遅延の標準偏差としてのジッター σ 。
- CPU および GPU の利用率 U 、バス上の I/O 飽和。

スケジューラビリティと遅延予算。周期リアルタイムタスクに対して利用率を計算する

$$[H]U = \sum_{i=1}^n \frac{C_i}{T_i}, \quad (373)$$

ここで C_i は最悪実行時間、 T_i は周期である。レート単調スケジューリング下では、保守的な境界は n タスクに対し $U \leq n(2^{1/n} - 1)$ である。単純な制御ループでは $C \leq T$ を強制し、ジッターと I/O 遅

延のためのスラックを確保する。エンドツーエンド制御遅延は加算的である：

$$[H]L_{\text{total}} = L_{\text{sense}} + L_{\text{proc}} + L_{\text{comm}} + L_{\text{act}}, \quad (374)$$

各項は正確な安定性マージンのため測定され境界付けられなければならない。

実用的なツールとその役割。

- トレーシングとタイムライン取得：

- NVIDIA Nsight Systems：CPU スレッドと CUDA カーネルをホストおよび GPU 間で相関付ける。GPU 加速知覚および ML 推論パイプラインに使用。
- Linux ftrace および LTTng：割込み遅延とコンテキストスイッチのためのカーネルおよびユーザ空間トレーシング。
- ros2_tracing：ROS2 計測トレースのための LTTng 統合、ROS コールバックとトピックの相関を可能にする。

- プロファイラ：

- perf および FlameGraphs：CPU ホットスポットおよびロック競合；経時サンプリングされたコールスタックを生成。
- Intel VTune：x86 開発時の詳細な CPU マイクロアーキテクチャホットスポット。
- Nsight Compute：推論に使用される CUDA カーネルのカーネルレベル GPU 性能カウンタ。

- リアルタイム遅延テスト：

- cyclicttest および rt-tests：PREEMPT_RT カーネル上で最悪スケジューリング遅延を測定。
- タイムスタンプ付きハードウェアメッセージを用いたハードウェアインループテストでエンドツーエンドタイミングを検証。

- ロギングおよび集約：

- rosbag2：センサおよび制御メッセージの同期記録のため。
- Prometheus + Grafana：長期テレメトリ（CPU、メモリ、ネットワーク）のため。

- ネットワークおよびプロセス間：

- Wireshark および tc（トラフィックコントロール）：ROS 2 DDS 遅延およびパケットロスの特徴付ける。
- tc qdisc および netem：頑健性テストのためのネットワーク劣化エミュレーション。

- 動的トレーシング：

- eBPF（bcc/tools）：カーネルイベントおよびシステムコール遅延の低オーバーヘッドライブトレーシング。

測定手法。テストでは以下の手順に従う：

1. 負荷を合成：知覚と移動ワークロードを同時に実行し共有リソースに負荷をかける。
2. PTP（Precision Time Protocol）でホスト間クロックを同期しサブミリ秒相関を実現。
3. 統合トレースを取得：ROS イベント、カーネルスケジューライイベント、GPU タイムラインを単一トレースセッションで記録。
4. 統計を計算するための後処理：平均、中央値、95/99 パーセンタイル、最悪値。
5. ホットスポットをコードにマッピングし予算を割り当てるまたはリファクタリングする。

具体的な実装パターン。以下の ROS 2 rclpy スニペットはヒューマノイド知覚パイプラインでメッ

セージ遅延をロギングする。遅延ヒストグラムを計算し統計を CSV に書き出す。トピックおよびメッセージ型はシステムの型に置き換えること。

コードサンプル 155 ROS 2 latency logger for sensor-to-callback latency

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import Image # 実際のメッセージ型
import numpy as np
import csv
import os
import atexit
from typing import List

class LatencyLogger(Node):
    def __init__(self) -> None:
        super().__init__('latency_logger')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self.sub = self.create_subscription(
            Image,
            '/camera/image_stamped',
            self.cb,
            qos
        )
        self.latencies: List[float] = []
        self.csv_path = 'latency_stats.csv'
        # 既存ファイルを上書きしないように初回のみヘッダ書き込み
        if not os.path.exists(self.csv_path):
            with open(self.csv_path, 'w', newline='') as f:
                csv.writer(f).writerow(['count', 'mean', 'p95', 'p99'])
        atexit.register(self._write_stats) # 終了時にも出力

    def cb(self, msg: Image) -> None:
        now = self.get_clock().now().to_msg()
        send_ts = msg.header.stamp.sec + msg.header.stamp.nanosec * 1e-9
        recv_ts = now.sec + now.nanosec * 1e-9
```

```

        lat = recv_ts - send_ts
        self.latencies.append(lat)
        if len(self.latencies) % 1000 == 0:
            self._write_stats()

def _write_stats(self) -> None:
    if not self.latencies:
        return
    a = np.array(self.latencies)
    mean = float(a.mean())
    p95 = float(np.percentile(a, 95))
    p99 = float(np.percentile(a, 99))
    with open(self.csv_path, 'a', newline='') as f:
        csv.writer(f).writerow([len(a), mean, p95, p99])

def main(args=None) -> None:
    rclpy.init(args=args)
    node = LatencyLogger()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計トレードオフと計測の影響。計測は CPU を消費しタイミングを攪乱する可能性がある。層化された戦略を用いる：

- 低オーバーヘッド本番メトリクス（実行可能カウンタ、ヒストグラム）。
- オフラインデバッグのための高忠実度トレース。
- 継続的オーバーヘッドなしでホットスポットを見つけるためのサンプリングプロファイラ。

エンジニアリングへの影響と運用リスク。

- 最悪ケース尾部（99 パーセンタイル）を見落とすと稀なイベント時に不安定化するリスク。
- GPU/CPU 競合は黙ってステップジッタを増加させバランスコントローラを不安定化させる。
- クロックスキューは相関を無効化し；同期不良は根本原因を隠す。
- リアルタイムループでの重いトレーシングはバグを隠すまたは偽陰性を生む。

性能テストを設計する際、測定忠実度と干渉のバランスを取る。各サブシステムに厳格な遅延予算を確保し、フライト中は控えめに計測する。

47 統合のデバッグ

47.1 通信エラーの診断

ソフトウェアレベルのデバッグパターンに倣い、焦点は知覚・計画・作動の統合を破壊する通信層の障害へ移る。以下の診断手法は、通信エラーを人間型ロボットの安定性と安全性に具体的な影響を与える測定可能な確率過程として扱う。

問題定義. 人間型ロボットでは、通信エラーはメッセージの損失、可変遅延、ジッタ、重複、または破損として現れる。これらの障害は閉ループ制御を不安定化し、知覚融合を劣化させ、または安全デッドラインに違反する可能性がある。目標は、通信障害をネットワーク、ミドルウェア (ROS/ROS2/GROOT)、またはハードウェアリンクのいずれかに検出、定量化、および局在化することである。

技術的分析と測定可能メトリクス.

• 記録すべき主要指標:

1. パケット損失率 $p = \text{lost/sent}$.
2. 一方向遅延分布 $L(t)$ とその平均 μ_L および標準偏差 σ_L .
3. 到着間ジッタ $J = \text{std}(\text{interarrival times})$.
4. 順序外または重複メッセージ数。
5. 秒あたりバイト数のスループット。

• 損失推定の統計的信頼性. 経験的損失率 \hat{p} を持つ N 回の観測送信に対して、正規近似を用いて 95%信頼区間を近似する:

$$[H]\hat{p} \pm z_{0.975} \sqrt{\frac{\hat{p}(1-\hat{p})}{N}}, \quad (375)$$

ここで $z_{0.975} \approx 1.96$. 損失が小さい場合は区間を狭めるために N を増やす。

制御互換性バジェット. 目標帯域幅 ω_b (rad/s) と位相余裕 ϕ_m (ラジアン) を持つ制御ループに対して、許容最大定常通信遅延 τ_{\max} は

$$[H]\tau_{\max} = \frac{\phi_m}{\omega_b}. \quad (376)$$

モデル化されたプラントおよびコントローラ動特性の後の残存位相余裕を ϕ_m とする。例えば、 30° の余裕は $\tau_{\max} = \frac{\pi/6}{\omega_b}$ を与える。

診断ワークフロー (実用的、順序付き手順) .

1. パッシブ測定:

- 各トピックにパブリッシャおよびサブスクリバでのタイムスタンプ付けを実装する。
- 到着タイムスタンプ、ヘッダスタンプ、およびペイロードチェックサムを記録する。

2. メトリクス算出:

- $\hat{p}, \mu_L, \sigma_L, J$ を推定し、順序外シーケンスを検出する。
- 統計的有意性を設定するために式 (1) を用いる。

3. 障害局在化：

- 単一 NIC にバインドされたトピック全体で損失が相関する場合、ネットワークまたはスイッチを疑う。
- 大きなペイロードを持つトピックのみに損失が影響する場合、MTU、フラグメンテーション、またはシリアライズオーバーヘッドを疑う。
- ジッタのスパイクが CPU 負荷と一致する場合、ミドルウェアでのスケジューリングまたはガベージコレクションを疑う。

4. 制御負荷下での再現：

- 測定帯域幅で合成トラフィックを実行する。
- リアルタイムスケジューリングと QoS（ROS2/RTOS 内）を切り替えて設定を検証する。

5. 修復と検証：

- QoS プロファイルを適用する：ROS2 での信頼性、履歴深度、およびデッドライン。
- 小さく遅延に敏感なメッセージに対して TCP_NODELAY を有効にする。
- 安全な場合は MTU を調整し、ジャンボフレームを有効にする。
- 改善を確認するために測定を再実行する。

実装例：ROS2 遅延および損失プローブ。以下のスニペットはトピックをサブスクライブし、パブリッシャヘッダタイムスタンプを用いて一方向遅延を算出し、到着間隔を期待周期と比較して損失のようなイベントをログに記録する。/joint_states および expected_rate をデプロイメント値に置き換えること。

コードサンプル 156 ROS2 プローブ：遅延、ジッタ、および近似損失を測定

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
import time
import statistics
from typing import List, Optional
import threading

class CommProbe(Node):
    def __init__(self,
                  topic: str = '/joint_states',
                  expected_rate: float = 100.0,
                  window_size: int = 1000) -> None:
        super().__init__('comm_probe')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
```

```

)
self._sub = self.create_subscription(
    JointState, topic, self._cb, qos)

self._expected_dt = 1.0 / expected_rate
self._window_size = window_size

self._arrivals: List[float] = []
self._latencies: List[float] = []
self._last_recv: Optional[float] = None
self._miss_count = 0
self._recv_count = 0
self._lock = threading.Lock()

self._timer = self.create_timer(1.0, self._print_stats)

def _cb(self, msg: JointState) -> None:
    now = self.get_clock().now().nanoseconds * 1e-9
    pub_stamp = msg.header.stamp.sec + msg.header.stamp.nanosec * 1e-9
    latency = now - pub_stamp
    with self._lock:
        self._latencies.append(latency)
        if self._last_recv is not None:
            dt = now - self._last_recv
            self._arrivals.append(dt)
            if dt > 1.5 * self._expected_dt:
                self._miss_count += int(round(dt / self._expected_dt)) - 1
            self._last_recv = now
        self._recv_count += 1
        # リングバッファでメモリ使用量を制限
        if len(self._latencies) > self._window_size:
            self._latencies.pop(0)
        if len(self._arrivals) > self._window_size:
            self._arrivals.pop(0)

def _print_stats(self) -> None:
    with self._lock:
        if not self._latencies or not self._arrivals:
            return
        self.get_logger().info(

```

```

        f"recv={self._recv_count}_miss~={self._miss_count}_\n"
        f"latency_mean={statistics.mean(self._latencies):.3f}s_\n"
        f"jitter={statistics.pstdev(self._arrivals):.3f}s"
    )

def main(args=None) -> None:
    rclpy.init(args=args)
    probe = CommProbe()
    try:
        rclpy.spin(probe)
    except KeyboardInterrupt:
        pass
    finally:
        probe.destroy_node()
        rclpy.shutdown()

```

根本原因と対象修正.

- ネットワーク混雑：トラフィックシェーピングを追加し、QoS 優先度を上げる、または制御トラフィックを専用インターフェースに分離する。
- シリアルライズオーバーヘッド：ゼロコピートランスポートを使用し、メッセージを必須フィールドに絞る。
- ミドルウェア設定ミス：パブリッシャおよびサブスクライバ間で QoS パラメータを一致させる。
- CPU スケジューリングおよびリアルタイムの問題：リアルタイム優先度を設定し、ハードデッドラインには PREEMPT_RT カーネルを使用する。
- ハードウェア障害：不良 NIC、スイッチバッファ、不良ケーブル；リンク層カウンタで確認する。

エンジニアリングへの影響、トレードオフ、および運用上のリスク.

- 信頼性（再送信）を上げると遅延が増える；信頼性と遅延のバランスはドメイン固有である。
- 専用ネットワークまたは VLAN を予約すると干渉は減るが、システムの複雑さとコストが増える。
- 厳密な遅延バジェットは、より高いソフトウェア決定論とより高価なリアルタイムハードウェアを要求する。
- 診断されていない断続的通信エラーは偶発的な作動器スパイクを引き起こし、安全上の危険をもたらす可能性がある。

ロボット上での継続的モニタリングと自動アラート閾値を採用する。式 (1) および (2) を用いてトレードオフを早期に定量化し、測定されたネットワーク条件下でコントローラが安定したままであることを確保する。

47.2 同期問題の取り扱い

これまでの通信障害の分析では、メッセージの損失とプロトコルの不整合を根本原因として挙げた。これらの障害に続いて同期問題が生じることが多い。タイミングエラーは、複数のサブシステム

が断続的に通信するときに融合、制御、安全の失敗を増幅するからである。

問題の定義。ヒューマノイドロボットは高速の関節コントローラ、慣性センサ、ビジョン、プランナモジュールを協調させる。各サブシステムは通常独自のローカルクロックで動作する。同期（時刻基準の整合）は、異なるソースのタイムスタンプが同じ物理時刻を指していることを保証するプロセスである。同期の不良は三つの運用上の失敗を引き起こす：センサ融合の誤り、サンプルタイミングの不整合による制御ループの不安定化、事後デバッグを妨げる不正確なログ。

技術的分析。ローカルクロックと基準クロックの関係をアフィンなドリフトとオフセットでモデル化する：

$$[H]t_{\text{local}}(t_{\text{ref}}) = \alpha t_{\text{ref}} + \beta, \quad (377)$$

ここで α はクロックスキューを、 β はオフセットをモデル化する。ネットワーク遅延、可変キューイング、OS スケジューリングジッタはノイズを加える。離散コントローラの公称サンプリング周期 T_s に対し、サンプリングジッタ δt_k は実効サンプル時刻を $T_s + \delta t_k$ とする。コントローラ帯域近傍の角周波数 ω に対し、位相摂動は $\Delta\phi \approx \omega \delta t$ と近似されるため、ジッタは位相余裕を減らし歩行のバランスコントローラを不安定化させ得る。

主要な測定量：

- センサとマスタクロック間のオフセット推定値 $\hat{\beta}$ 。
- 時間窓内のスキュー推定値 $\hat{\alpha}$ 。
- 往復遅延 RTT と片方向遅延分散（ジッタ）。
- パケット到着時刻の不規則性（到着間分布）。

診断ワークフロー。同期障害を切り分けるには以下の手順を用いる：

1. 決定論的負荷下で症状を再現する。
2. センサと基準ホストからタイムスタンプペアを収集する。ハードウェアタイムスタンプが利用可能ならばそれを用いる。
3. 線形回帰により式 (1) のモデルに当てはめ $\hat{\alpha}, \hat{\beta}$ を得る。
4. 残差を測定しジッタを定量化し、自己相関を調べバースト遅延を検出する。
5. タイミングエラーを制御劣化または融合不整合と関連させる。

実用的な緩和技術と実装上の注意：

- サブマイクロ秒整合のためハードウェアタイムスタンプとタイムスタンプ付き NIC を優先する。
- スイッチ Ethernet でサブマイクロ秒同期には Precision Time Protocol (PTP、IEEE 1588) を用いる。利用可能な場合はスイッチに Boundary Clock または Transparent Clock を用いる。
- ネットワークハードウェアが限られたシステムでは、NTP を頻繁なポーリングでソフト同期タスクにのみ用いる。
- 決定性が重要な場合は、センサとモーションコントローラ間に同期トリガライン（TTL パルス）を実装しネットワーク起因ジッタを回避する。
- ROS ベースのスタックでは、到着時刻ではなくヘッダタイムスタンプを持つ同期メッセージフィルタを用いる。センサドライバが一貫した時刻ドメインでハードウェアタイムスタンプを公開することを確認する。
- 絶対同期が非現実的な場合は、エピソードごとの相対時刻整合を用いる：オフセットをランごと

に計算し、ストリームをオフラインで整合してログとモデル学習に用いる。

コンパクトな診断ユーティリティ。以下の Python スニペットはペアタイムスタンプからクロックスキューとオフセットを推定する。roscpp エクスポートまたは直接ソケットキャプチャによるオンロブットテストに実用的である。

コードサンプル 157 タイムスタンプペアからクロックスキューとオフセットを推定

```
#!/usr/bin/env python3
import numpy as np
import pathlib
import argparse
import logging
from typing import Tuple

# ロギング設定
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

def load_pairs(path: pathlib.Path) -> np.ndarray:
    """CSVから(t_ref, t_loc)を読み込み、shape=(N,2)のndarrayを返す。"""
    data = np.loadtxt(path, delimiter=",", ndmin=2)
    if data.shape[1] != 2:
        raise ValueError("CSVは2列(t_ref, t_loc)である必要があります")
    return data

def estimate_skew_offset(t_ref: np.ndarray, t_loc: np.ndarray) -> Tuple[float, float, float]:
    """最小二乗法でskew( $\alpha$ )とoffset( $\beta$ )を推定し、残差の標準偏差も返す。"""
    A = np.vstack([t_ref, np.ones_like(t_ref)]).T
    alpha, beta = np.linalg.lstsq(A, t_loc, rcond=None)[0]
    residuals = t_loc - (alpha * t_ref + beta)
    jitter_std = np.std(residuals)
    return alpha, beta, jitter_std

def windowed_skew(t_ref: np.ndarray, t_loc: np.ndarray, w: int, step: int = 1):
    """スライディングウィンドウで局所skewを推定。"""
    skews = []
    for i in range(0, len(t_ref) - w + 1, step):
        A = np.vstack([t_ref[i : i + w], np.ones(w)]).T
        a, _ = np.linalg.lstsq(A, t_loc[i : i + w], rcond=None)[0]
        skews.append(a)
```

```

return np.array(skews)

def main():
    parser = argparse.ArgumentParser(description="タイムスタンプペアからクロックスキュー  

    parser.add_argument("csv", type=pathlib.Path, help="timestamp_pairs.csv")
    parser.add_argument("-w", "--window", type=int, default=100, help="ウィンドウ幅 (秒)")
    parser.add_argument("-s", "--step", type=int, default=1, help="ウィンドウスライド幅 (秒)")
    args = parser.parse_args()

    pairs = load_pairs(args.csv)
    t_ref, t_loc = pairs[:, 0], pairs[:, 1]

    alpha, beta, jitter_std = estimate_skew_offset(t_ref, t_loc)
    logging.info(f"alpha(skew)={alpha:.9f}, beta(offset)={beta:.6f}μs")
    logging.info(f"residual_jitter_std={jitter_std*1e3:.3f}μms")

    skews = windowed_skew(t_ref, t_loc, args.window, args.step)
    drift_ppm = (skews - alpha).mean() * 1e6
    logging.info(f"mean_skew_drift={drift_ppm:.2f}ppm")

if __name__ == "__main__":
    main()

```

設計上のトレードオフと工学上の含意：

- ハードウェアタイムスタンプと PTP はコストを増やすがタイミング不確定性を大幅に低減する。ヒューマノイドのバランスや高速知覚制御ループがサブミリ秒整合に依存する場合に用いる。
- 同期トリガ配線は配線の複雑さを増やしモジュール性を低下させる。フットコンタクトスイッチのような安全臨界センシングにはトリガを優先する。
- ソフトウェアによるソフト同期は安価だが、ネットワーク負荷の変動時に断続的な不安定化のリスクがある。
- 適切な時刻整合なしのログは学習やオフライン解析のためのデータセットの価値を低下させる。

監視すべき運用上のリスク：

- 補正されないスキューは蓄積し、高速コントローラとプランナ間で長期ドリフトを引き起こす。
- CPU またはネットワーク負荷ピーク時のジッタスパイクは一時的な制御損失を生じ転倒を引き起こす。
- パッケージ間で不整合のあるタイムスタンプドメインは再現性と認証を妨げる。

同期障害をデバッグする際は、オフセット、スキュー、ジッタを数値的に定量化する。高信頼タス

クにはハードウェアタイムスタンプまたは有線トリガを採用する。タイミングエラーが制御余裕とセンサ融合精度にどう伝播するかを追跡し、弁護可能な工学選択を行う。

47.3 実環境でのテスト

これらのテストは、実際の運用制約下でハードウェア・ソフトウェアパイプライン全体を動作させることにより、同期および通信診断を拡張する。以下の資料では、観測された遅延、ジッタ、およびパケット損失を、ヒューマノイドロボットのための実践的なテスト手順および自動チェックにどう変換するかを示す。

問題の定義と運用上の関連性。実環境テストは、シミュレーションでは現れない統合障害を明らかにしなければならない。典型的な障害モードには、低レベルバランスコントローラを不安定化する累積遅延、温度や衝撃によるセンサバイアスの変化、無線リンクでの断続的なパケット損失、長時間運転後のアクチュエータ飽和などがある。各障害モードは測定可能なメトリクスを生じる。エンジニアは、以下を実現する再現可能なテストを必要とする：

- センシング、ネットワーキング、計算、および作動にわたるエンドツーエンド遅延とジッタを定量化；
- 環境摂動にわたるコントローラの頑健性を検証；
- 通信劣化下での回復および安全停止動作を行使。

技術分析：遅延およびジッタバジェット。エンドツーエンド遅延バジェットを定義する

$$[H]L_{\text{total}} = L_{\text{sense}} + L_{\text{comm}} + L_{\text{compute}} + L_{\text{act}}. \quad (378)$$

公称帯域幅 f_c を持つ閉ループコントローラに対して、保守的な制約を課す

$$[H]L_{\text{total}} \leq \frac{\alpha}{f_c}, \quad (379)$$

ヒューマノイドバランスコントローラには $\alpha \approx 0.1$ を推奨する。この制約はコントローラ設計をネットワークおよび計算要件に写像する。ジッタ、すなわちエンドツーエンド遅延の標準偏差 σ_L は位相余裕を劣化させる。最悪ケースの余裕減少見積もりを用いる：

$$[H]\Delta\phi \approx 2\pi f_c \sigma_L, \quad (380)$$

そして $\Delta\phi$ が利用可能な位相余裕を下回ることを確保する。

テスト設計とシナリオ。(1) の各項をストレスし、(2) および (3) の診断を生成するようテストを設計する。

1. 静的遅延測定：

- ロボットを静止させた状態で L_{total} を測定。
- 平均、中央値、および σ_L を算出。

2. 閉ループ帯域幅スイープ：

- 増加する周波数で正弦波セットポイントを注入。
- 追従振幅が 3 dB 低下する f_{cutoff} を特定。

3. 摂動回復：

- 歩行中に既知の横方向インパルスを印加し、回復時間と必要トルクを記録。

4. 通信劣化シナリオ：

- リンクエミュレータを用いてパケット損失、増加遅延、およびジッタを導入。
- 安全停止トリガおよびコントローラフォールバックを検証。

5. 環境頑健性：

- 照明を変更し、反射面を追加、または床摩擦を変動させる。
- 知覚パイプラインおよび足部接触推定を検証。

収集すべき実践的メトリクス。

- 遅延平均および標準偏差： μ_L , σ_L 。
- パケット損失率 p_{loss} 。
- RMS 追従誤差 $e_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{i=1}^N e_i^2}$ 。
- 擾乱後の回復時間 t_{recover} 。
- テストサイクルあたりのエネルギー消費。

実装：自動ロギングおよび閾値ベースフェイルオーバー。以下の ROS2 ノード例は、着信センサメッセージおよび発信コマンドにタイムスタンプを付与する。Welford のアルゴリズムを用いてエンドツーエンド遅延のランニング平均および分散を計算する。遅延が設定可能な閾値を超えた場合、ノードは安全停止コマンドをトリガする。

コードサンプル 158 ROS2 遅延モニタおよびヒューマノイドテスト用安全停止トリガ。

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
from std_msgs.msg import Bool
import csv
import time
import os
from datetime import datetime
from typing import Optional, Tuple

class LatencyMonitor(Node):
    def __init__(self) -> None:
        super().__init__('latency_monitor')

        # QoS設定：信頼性を高めて遅延を正確に測定
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
```

```

self.sub = self.create_subscription(
    JointState, '/joint_states', self.cb, qos)
self.safe_pub = self.create_publisher(Bool, '/safe_stop', 10)

self.declare_parameter('latency_threshold', 0.05)
self.declare_parameter('log_dir', '/tmp/latency_logs')
self.threshold: float = self.get_parameter(
    'latency_threshold').get_parameter_value().double_value

# Welford オンライン分散計算
self._n: int = 0
self._mean: float = 0.0
self._M2: float = 0.0

# ログファイル初期化
log_dir: str = self.get_parameter(
    'log_dir').get_parameter_value().string_value
os.makedirs(log_dir, exist_ok=True)
timestamp: str = datetime.now().strftime('%Y%m%d_%H%M%S')
log_path: str = os.path.join(log_dir, f'latency_{timestamp}.csv')
self._csv_file = open(log_path, 'w', newline='')
self._writer = csv.writer(self._csv_file)
self._writer.writerow(['recv_time_sec', 'msg_time_sec', 'latency_sec'])

self.get_logger().info(f'LatencyMonitor started, threshold={self.threshold}s')

def cb(self, msg: JointState) -> None:
    now = self.get_clock().now()
    msg_time = rclpy.time.Time.from_msg(msg.header.stamp)
    latency_sec = (now - msg_time).nanoseconds * 1e-9

    # オンライン統計更新
    self._n += 1
    delta = latency_sec - self._mean
    self._mean += delta / self._n
    self._M2 += delta * (latency_sec - self._mean)

    self._writer.writerow([now.nanoseconds * 1e-9,
                           msg_time.nanoseconds * 1e-9,

```

```

        latency_sec]))

    if latency_sec > self.threshold:
        self.get_logger().warn(
            f'Latency exceeded: {latency_sec:.4f}s > {self.threshold}s')
        self.safe_pub.publish(Bool(data=True))

def get_stats(self) -> Tuple[float, float]:
    if self._n < 2:
        return self._mean, 0.0
    var = self._M2 / (self._n - 1)
    return self._mean, var ** 0.5

def destroy_node(self) -> None:
    self._csv_file.close()
    super().destroy_node()

def main(args=None) -> None:
    rclpy.init(args=args)
    node = LatencyMonitor()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        mean, stddev = node.get_stats()
        node.get_logger().info(
            f'Shutting down: {node._n} samples, {node._mean} mean, {node._stddev} std')
        node.destroy_node()
        rclpy.shutdown()

```

解釈および判断ロジック。

- 記録された μ_L および σ_L を用いて計算リソースを割り当てる、またはより高速なミドルウェアを選択する。
- L_{total} が (379) に違反する場合、制御帯域幅 f_c を減らすか通信リンクを改善する。
- 無線リンクで p_{loss} が 1-2

エンジニアリングへの影響、トレードオフ、およびリスク。

- センササンプリングレートを上げると推定誤差は減少するが、 $L_{compute}$ およびエネルギー使用量

が増加する。応答性とバッテリー寿命の間でトレードオフする。

- ・遅延バジェットを厳しくすると、より高価なネットワークハードウェアまたはローカル計算へのオフロードが必要になる。
- ・単一環境にテストを過適合すると脆いシステムが生まれる。シナリオの多様性は必須。
- ・運用上のリスクには、フォールバック戦略を厳密にテストしない場合の安全でない動作が含まれる；安全停止、劣化モード、および人間相互作用ガードをすべての障害条件下で検証する。

ロボティクスの将来動向

48 ロボティクスにおける AI の進歩

48.1 制御のための新興 AI モデル

シミュレーション忠実度と知覚モデル移転の先行検討を踏まえ、本小節ではヒューマノイド制御を現在牽引する AI モデルを考察する。焦点は高次元ダイナミクス、接触、安全臨界要件に対処するアーキテクチャ、学習パラダイム、展開パターンである。

ヒューマノイド制御は非線形多体ダイナミクス、断続的接触、高次元センサストリームを扱う必要がある。新興 AI アプローチは補完的なカテゴリに分類される：

- ・モデルフリー強化学習 (RL)：PPO, SAC, TD3 などのアルゴリズムは報酬信号から直接方策を学習する。エンドツーエンド技能には優れるが、多くの相互作用と注意深い報酬設計を要する。
- ・学習済みダイナミクスを用いたモデルベース制御：ニューラルモデルがロボットダイナミクスを近似し、モデル予測制御 (MPC) のような計画アルゴリズムがオンラインで行動を生成し、環境ステップを大幅に削減する。
- ・模倣学習とオフライン RL：行動複製とオフライン方策抽出は記録済みデモンストレーションを活用してコントローラをブートストラップし、ハードウェア上の危険な探索を削減する。
- ・シーケンス・基盤モデル：Transformer と拡散モデルは軌道をシーケンスとして扱う。目標や高レベルコマンドを条件として安定した運動プリミティブを生成できる。
- ・ハイブリッドアーキテクチャ：残差 RL (古典コントローラの上で補正を学習) や安全フィルタ (学習済み方策を監視する保証付き低レベルコントローラ) のような組合せ。

工学問題を正確に述べると：ダイナミクスと安全制約を尊重しながらタスクコストを最小化する制御方策または行動シーケンスを合成せよ。微分可能な学習済みダイナミクスモデル f_θ が利用可能な場合、反復最適化は

$$\begin{aligned} \min_{u_{0:H-1}} \quad & \sum_{t=0}^{H-1} c(x_t, u_t) + c_T(x_H) \\ \text{s.t.} \quad & x_{t+1} = f_\theta(x_t, u_t), \quad x_0 = x_{\text{meas}} \\ & g(x_t, u_t) \leq 0 \quad \forall t \end{aligned} \tag{381}$$

を解く。ここで x はベース姿勢、関節角度、速度、接触インジケータを含む。学習済みモデル f_θ は完全状態予測器または低次元潜在モデルであり得る。制約 g は関節限界、接触摩擦円錐、安全マージンを符号化する。

主要なモデリング・学習上の考慮事項：

1. 表現選択：物理情報組み込みネットワークまたは構造化予測器（例：リンク上のグラフニューラルネットワーク）は保存則を保持し、形態間の一般化を改善する。
2. 不確実性と頑健性：アンサンブルまたは確率モデルは予測分散を推定する。不確実性を意識したコスト（リスク感受性 MPC）での計画は接触が曖昧な場合に脆い計画を防ぐ。
3. データ効率：デモンストレーションデータセットとオンライン微調整を組み合わせることでハードウェアリスクを削減。オフライン RL 手法は分布外逸脱を避けるための保守的な目的関数を用いる。
4. 時間信用と長期推論：Decision Transformer のようなシーケンスモデルは所望リターンを条件として時間的に一貫した方策を生成し、明示的な報酬設計を不要にする。

研究・プロトタイピングの実装パターンは、ダイナミクスモデルをオフラインで学習し、ハードウェア試験前にシミュレーションでシューティングベース MPC を実行することである。以下のコードスニペットは、短いホライズンのダイナミクスネットワークを学習し、学習済みモデル上で候補行動シーケンスを評価する簡潔な PyTorch 風ループを示す。コメントは工学読者向けに簡潔に絞られている。

コードサンプル 159 1 ステップダイナミクスモデルを学習し、単純なシューティング MPC を実行（プロトタイプ）。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Tuple, Optional
import numpy as np

class DynamicsModel(nn.Module):
    """
    確定的ダイナミクスモデル： $s_{t+1} = f(s_t, a_t)$ 
    正規化・デノイズ機能を内蔵し、安定学習を支援
    """
    def __init__(self,
                  state_dim: int,
                  action_dim: int,
                  hidden_dim: int = 512,
                  drop_prob: float = 0.1):
        super().__init__()
        self.state_dim = state_dim
        self.action_dim = action_dim

        # 入力統計量（推論時に使用）
        self.register_buffer('s_mean', torch.zeros(state_dim))
        self.register_buffer('s_std', torch.ones(state_dim))
```

```

self.register_buffer('a_mean', torch.zeros(action_dim))
self.register_buffer('a_std', torch.ones(action_dim))
self.register_buffer('ds_mean', torch.zeros(state_dim))
self.register_buffer('ds_std', torch.ones(state_dim))

self.net = nn.Sequential(
    nn.Linear(state_dim + action_dim, hidden_dim),
    nn.LayerNorm(hidden_dim),
    nn.ReLU(),
    nn.Dropout(drop_prob),
    nn.Linear(hidden_dim, hidden_dim),
    nn.LayerNorm(hidden_dim),
    nn.ReLU(),
    nn.Dropout(drop_prob),
    nn.Linear(hidden_dim, state_dim)
)

def forward(self, state: torch.Tensor, action: torch.Tensor) -> torch.Tensor:
    # 正規化入力
    s_norm = (state - self.s_mean) / (self.s_std + 1e-6)
    a_norm = (action - self.a_mean) / (self.a_std + 1e-6)
    x = torch.cat([s_norm, a_norm], dim=-1)
    ds_norm = self.net(x)
    # デ正規化して残差を加算
    ds = ds_norm * self.ds_std + self.ds_mean
    return state + ds

def update_stats(self,
                 states: torch.Tensor,
                 actions: torch.Tensor,
                 deltas: torch.Tensor):
    """データセット全体で統計量を更新"""
    self.s_mean.copy_(states.mean(0))
    self.s_std.copy_(states.std(0))
    self.a_mean.copy_(actions.mean(0))
    self.a_std.copy_(actions.std(0))
    self.ds_mean.copy_(deltas.mean(0))
    self.ds_std.copy_(deltas.std(0))

```

```

class ModelTrainer:
    def __init__(self,
                  model: DynamicsModel,
                  lr: float = 1e-3,
                  weight_decay: float = 1e-4,
                  device: str = 'cuda'):
        self.model = model.to(device)
        self.device = device
        self.opt = torch.optim.Adam(self.model.parameters(),
                                     lr=lr,
                                     weight_decay=weight_decay)
        self.scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            self.opt, patience=10, factor=0.5, verbose=False)

    def train_epoch(self,
                   states: torch.Tensor,
                   actions: torch.Tensor,
                   next_states: torch.Tensor) -> float:
        self.model.train()
        states, actions, next_states = [x.to(self.device) for x in
                                         (states, actions, next_states)]

        deltas = next_states - states
        pred = self.model(states, actions)
        loss = F.mse_loss(pred, next_states)

        self.opt.zero_grad()
        loss.backward()
        # 勾配クリップ
        torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)
        self.opt.step()
        return loss.item()

    def fit(self,
           states: torch.Tensor,
           actions: torch.Tensor,
           next_states: torch.Tensor,
           epochs: int = 500,
           batch_size: int = 256,
           val_split: float = 0.1) -> None:
        n = states.size(0)

```

```

n_val = int(n * val_split)
perm = torch.randperm(n)
train_idx, val_idx = perm[n_val:], perm[:n_val]

ds = next_states - states
self.model.update_stats(states, actions, ds)

best_val = float('inf')
for epoch in range(epochs):
    # ミニバッチ学習
    batch_idx = train_idx[torch.randperm(len(train_idx))[:batch_size]]
    tr_loss = self.train_epoch(states[batch_idx],
                                actions[batch_idx],
                                next_states[batch_idx])

    # 検証
    with torch.no_grad():
        val_pred = self.model(states[val_idx], actions[val_idx])
        val_loss = F.mse_loss(val_pred, next_states[val_idx]).item()

    self.scheduler.step(val_loss)
    if val_loss < best_val:
        best_val = val_loss
    else:
        if self.opt.param_groups[0]['lr'] < 1e-6:
            break

```

```

class ShootingMPC:
    """
    シンプルなランダムシャーティングMPC
    カーネル融合版コスト計算を含む
    """
    def __init__(self,
                  model: DynamicsModel,
                  cost_fn,
                  horizon: int = 20,
                  num_samples: int = 1024,
                  device: str = 'cuda'):
        self.model = model

```

```

        self.cost_fn = cost_fn
        self.horizon = horizon
        self.num_samples = num_samples
        self.device = device
        self.action_bounds = (-1.0, 1.0) # 正規化済みアクション範囲

def __call__(self, state: np.ndarray) -> np.ndarray:
    with torch.no_grad():
        x0 = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
        # 一様ランダムアクション候補
        A = torch.empty(self.num_samples, self.horizon, self.model.action_dim,
                        device=self.device).uniform_(*self.action_bounds)
        X = x0.repeat(self.num_samples, 1)
        total_cost = torch.zeros(self.num_samples, device=self.device)

        for t in range(self.horizon):
            a = A[:, t, :]
            X = self.model(X, a)
            total_cost += self.cost_fn(X, a)

        best_idx = torch.argmin(total_cost)
        best_act = A[best_idx, 0, :].cpu().numpy()
        return best_act

# -----
# 使用例（別ファイルで定義するcost_fnを渡す）
# -----
if __name__ == "__main__":
    # ダミーデータ
    states = torch.randn(10000, 50)
    actions = torch.randn(10000, 12)
    next_states = states + torch.randn(10000, 50) * 0.1

    model = DynamicsModel(state_dim=50, action_dim=12)
    trainer = ModelTrainer(model)
    trainer.fit(states, actions, next_states)

    # コスト関数例：状態ノルム＋アクション正則化
    def cost_fn(x, a):

```

```
return x.norm(dim=-1) + 1e-3 * a.norm(dim=-1)
```

```
mpc = ShootingMPC(model, cost_fn)
```

```
x0 = np.zeros(50)
```

```
u = mpc(x0)
```

ヒューマノイド制御のモデル評価では、性能と運用制約の両方を定量化する：

- ・レイテンシ：方策またはモデルの推論は制御ループタイミングを満たす必要がある。ニューラル積分器は実時間使用のための量子化または剪定を要する場合がある。
- ・安全性：学習済み方策は常にフォールバックコントローラまたは検証層と組合せ、安全でない関節コマンドを防ぐ。
- ・シムツリーリアルギャップ：ドメインランダムマイゼーションとシステム同定が移行ミスマッチを削減する。無 tether 試験前に tethered ハードウェアで接触ダイナミクスとトルクリミットを検証する。
- ・解釈可能性：構造化モデル（例：リンク単位グラフ）は故障診断を可能にし、保守・認証に重要である。

工学上の意味、トレードオフ、運用リスク：

- ・サンプル効率と最終漸近性能のトレードオフ：モデルベース手法は収束が速いが、モデルバイアスが残存すれば最終性能が制限される場合がある。
- ・計算量対頑健性：大きなシーケンスモデルは一般化を改善するが、バッテリー制約のあるヒューマノイドでは推論レイテンシとエネルギー消費を増大させる。
- ・故障モードには、長いロールアウトでの予測誤差の累積、ハードウェア上の安全でない探索、新規接触配置近傍での脆い挙動が含まれる。
- ・緩和戦略は冗長性を要する：アンサンブル、安全モニタ、保守的アクションクリッピング、シミュレーションからガード付きハードウェアへの段階的展開。

48.2 生成 AI とロボティクスの統合

制御のための新興 AI モデルに関する議論を踏まえ、本小節ではヒューマノイドシステムへの生成 AI の実践的な統合パターンを検討する。焦点は、安全性とリアルタイム制約を満たしながら、生成モデルを知覚、計画、低レベル制御に結びつけるエンジニアリングワークフローにある。

問題定義. ヒューマノイドロボットは非構造化タスクに対して柔軟な行動生成を必要とする。生成 AI（文脈を条件としてシーケンス、テキスト、連続出力を合成するモデル）は、タスクレベル計画、運動プリミティブ合成、意味解釈のための豊富な事前知識を提供する。しかし、これらのモデルは幻覚を生じ、厳格な安全性保証を欠き、可変遅延を示すことがある。エンジニアリングの問題は、動力学制約、リアルタイム制御、または認定要件を損なうことなく、生成出力を組み込むことである。

システムアーキテクチャと技術分析. 実践的な統合は層状パイプラインを用いる：

- ・知覚層: センサ融合と状態推定により信念状態 x_t を生成する。
- ・意図層: 生成モデル（テキストまたは運動）がタスク手がかりと信念状態を高レベル計画 π_{gen} に

写像する。

- 運動合成層: 条件付き生成運動モデルが軌道 τ_{gen} または運動プリミティブを生成する。
- 制御層: 認定済みコントローラ（MPC、QP ベース全身コントローラ）が動力学と安全性制約を実施し、 u_t を生成する。

重要な設計ルール：

1. 生成出力を提案として扱い、権威あるコマンドとはしない。
2. モデルの不確実性を定量化し、融合に重み付けする。
3. 幾何学的、運動学的、衝突制約を常に制御層で適用する。

簡潔な数学的融合モデルは、コントローラを生成提案 u_{gen} に近い許容制御 u を見つけ、標準制御 u_{nom} からの逸脱を最小化する最適化として表現する：

$$[H]u^* = \arg \min_u \|u - u_{\text{gen}}\|_R^2 + \lambda \|u - u_{\text{nom}}\|_Q^2 \quad \text{s.t.} \quad g(u, x_t) \leq 0, h(u, x_t) = 0, \quad (382)$$

ここで、 g は不等式制約（衝突、トルク限界）を符号化し、 h は動力学等式制約を符号化する。行列 R, Q は 2 つの提案に対する信頼と剛性を符号化する。単純な凸代理は線形ブレンドを用いる

$$[H]u = (1 - \alpha)u_{\text{nom}} + \alpha u_{\text{gen}}, \quad \alpha = f(\text{conf}), \quad (383)$$

ここで、 conf はモデル logit、予測エントロピ、またはアンサンブル分散から導かれる信頼度指標である。高不確実性を低 α に写像するように f を選ぶ。

不確実性定量化. 言語モデルには、トークンレベル \log 確率を用いて驚きとエントロピーを計算する。連続生成運動モデルには、アンサンブルモデルまたはモンテカルロドロップアウトを用いて共分散 Σ_{gen} を推定する。原則に基づく融合は逆共分散で重み付けする：

$$[H]u = (\Sigma_{\text{nom}}^{-1} + \Sigma_{\text{gen}}^{-1})^{-1}(\Sigma_{\text{nom}}^{-1}u_{\text{nom}} + \Sigma_{\text{gen}}^{-1}u_{\text{gen}}). \quad (384)$$

リアルタイムと安全性制約. 合成と作動の間に「安全シールド」を実装する。シールドは以下を実行する：

- 掃引体積を用いた高速衝突チェック。
- 関節限界とトルク実現可能性チェック（解析的限界経由）。
- 全身逆運動学のための到達可能性チェック。

いずれかのチェックが失敗した場合、保守的なフォールバック動作に縮退する。最悪遅延を維持し、制御スレッドにデッドラインを設定する。

実装スケッチ. 以下は、生成提案を購読し、統計チェックを実行し、安全な関節軌道を公開する ROS2 ライク Python ノードである。コードは、 u_{gen} と信頼度スコアを返すローカル推論クライアントを想定する。

コードサンプル 160 ROS2 ノード: 生成提案を受け入れ、安全性を実施し、安全な軌道を公開.

```
import rclpy
```

```

from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
from std_msgs.msg import Float32
from typing import List, Optional
import threading
import time

class GenIntegrator(Node):
    def __init__(self) -> None:
        super().__init__('gen_integrator')

        # QoS: ロボットハードウェア向けに信頼性を確保
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.VOLATILE,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        self.sub_proposal = self.create_subscription(
            JointState, '/gen/proposal', self.cb_prop, qos)
        self.sub_conf = self.create_subscription(
            Float32, '/gen/conf', self.cb_conf, qos)
        self.pub_safe = self.create_publisher(
            JointState, '/robot/joint_cmd', qos)

        # スレッドセーフな共有変数
        self._lock = threading.Lock()
        self.nominal: Optional[List[float]] = None
        self.last_conf: float = 0.0

        # パラメータ読み込み
        self.declare_parameter('joint_names', ['joint1', 'joint2'])
        self.joint_names: List[str] = (
            self.get_parameter('joint_names').get_parameter_value().string_array_value

        self.declare_parameter('fallback_position', [0.0] * len(self.joint_names))
        self.fallback_position: List[float] = (

```

```

        self.get_parameter('fallback_position').get_parameter_value().double_array

self.get_logger().info("GenIntegrator initialized.")

def cb_prop(self, msg: JointState) -> None:
    with self._lock:
        conf = self.last_conf

        alpha = self._map_conf_to_alpha(conf)
        u_nom = self._get_nominal()
        u_gen = msg.position

        # 次元チェック
        if len(u_gen) != len(self.joint_names):
            self.get_logger().warn("Joint dimension mismatch.")
            return

        u_safe = self._blend_and_check(u_nom, u_gen, alpha)
        out = JointState()
        out.header.stamp = self.get_clock().now().to_msg()
        out.name = self.joint_names
        out.position = u_safe if u_safe is not None else self.fallback_position
        self.pub_safe.publish(out)

def cb_conf(self, msg: Float32) -> None:
    with self._lock:
        self.last_conf = msg.data

def _map_conf_to_alpha(self, conf: float) -> float:
    # 信頼度を[0,1]にクランプ
    return max(0.0, min(1.0, (conf - 0.2) / 0.8))

def _get_nominal(self) -> List[float]:
    # 暫定: 静止姿勢を返す
    return list(self.fallback_position)

def _blend_and_check(self,
                    u_nom: List[float],
                    u_gen: List[float],
                    alpha: float) -> Optional[List[float]]:

```

```

        u = [(1.0 - alpha) * n + alpha * g for n, g in zip(u_nom, u_gen)]
        if not self._joint_limits_ok(u):
            return None
        if self._collision_check(u):
            return None
        return u

def _joint_limits_ok(self, q: List[float]) -> bool:
    # 簡易リミットチェック ( $\pm \pi$ )
    return all(-3.1416 <= v <= 3.1416 for v in q)

def _collision_check(self, q: List[float]) -> bool:
    # 実機ではFCL/MoveIt等を使用
    return False

def main(args=None):
    rclpy.init(args=args)
    node = GenIntegrator()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

エンジニアリング実践と検証. 以下の手順を用いる：

1. シミュレーションインザループ: ハードウェア試験前に Isaac Sim で生成提案をテストする。
2. シャドーモード: 実機で作動なしに生成スタックを並行実行する。
3. 人間監視: 高リスクタスクには監督者の確認を要求する。
4. 継続的ロギング: 提案、信頼度、シールド上書きをオフライン解析のため保存する。

運用上のトレードオフとリスク. 生成 AI の統合は適応性を向上させ、手書きルール負荷を軽減する。コストとリスクは以下を含む：

- 遅延予算: 生成モデルは推論遅延を追加する；リアルタイム需要にはモデル蒸留を用いる。

- 安全性保証: 生成出力の形式的検証は現時点で実行不可能；制御レベルで厳格な制約を実施する。
- データセットバイアス: モデル事前知識は安全でない動作を符号化する場合がある；学習データを選定し拡張する。
- 過度依存: モデル出力を盲目的に信頼すると運用上リスクが増大；保守的フォールバックを維持する。

ヒューマノイド設計への具体的影響には、低遅延推論のためセンサ近傍に計算資源を割り当て、提案を分離するモジュラー制御パイプラインを設計し、シールド動作を認定するためのエンジニア時間を予算配分することが含まれる。堅牢な展開には、保守的な信頼関数、厳格なシミュレーション検証、人間上書き手順が必要である。

48.3 ケーススタディ：予知保全

前の小節では、生成モデルが訓練データを拡張し、最新の制御モデルがより豊かな時間表現を提供する方法を示した。このケーススタディでは、これらの進歩をヒューマノイドロボットの予知保全ワークフローに適用し、物理を意識したモデルとデータ駆動型の系列学習を組み合わせる。

問題定義と運用上の関連：産業・サービス用ヒューマノイドはアクチュエータ密度が高く、複雑な運動学を持つ。ヒップまたはアングルアクチュエータの予期しない故障は、ミッション中止や人の怪我を引き起こす可能性がある。目的は次の2つである：

1. 異常なコンポーネント挙動を早期に検出し、
2. ジョイントおよびアクチュエータの残存有用寿命（RUL）を予測して、メンテナンスを先行的にスケジュールする。

利用可能なセンサは、ジョイントエンコーダ、モータ電流、モータ温度、IMU 振動スペクトル、トルクセンサ出力である。保全戦略は、偽陽性（不要なダウンタイム）と偽陰性（壊滅的故障）の両方を最小化しなければならない。

技術分析：モデル構造とデータエンジニアリング

- データソースとラベル付け：
 - 制御ループ用に 500 Hz でサンプリングされた encoder 読み値と motor_current を使用する。
 - IMU から短時間フーリエ変換（STFT）ウィンドウを用いて振動特徴を抽出する。
 - 歴史的メンテナンスログまたは Isaac Sim 内のシミュレートされた劣化から RUL をラベル付けする。
- 信号処理と特徴抽出：
 - 統計特徴：スライディングウィンドウ上の平均、分散、尖度を計算する。
 - ベアリング摩耗署名のためのスペクトルエネルギーバンドおよびエンベロープ特徴を計算する。
 - ジョイントごとに標準化を適用して単位スケール差を除去する。
- モデリングアプローチ：
 - 単純な物理情報を含む状態モデルとニューラル系列予測器を組み合わせる。
 - 状態空間摩耗モデル：

$$[H]x_{t+1} = x_t + \Delta(x_t, u_t) + w_t, \quad y_t = h(x_t) + v_t \quad (385)$$

ここで x_t は潜在摩耗状態、 u_t は制御入力、 y_t は観測センサ特徴を表す。

- データ駆動型推定器は最近の履歴から RUL を予測する： $\hat{r}_t = f_\theta(\mathbf{y}_{t-w+1:t})$ 。
- 教師あり RUL 学習のための損失：

$$[H]L(\theta) = \frac{1}{N} \sum_{i=1}^N (\hat{r}_{t_i} - r_{t_i})^2 + \lambda \mathcal{R}(\theta), \quad (386)$$

ここで \mathcal{R} はモデル複雑さを正則化する。

- アーキテクチャの選択：
 - 中規模データセットには LSTM または時間畳み込みネットワーク (TCN)。
 - 大規模かつラベル付きデータセットが存在する場合、または合成データが現実を拡張する場合は Transformer ブロック。
 - 認識的不確実性を定量化するベイズまたはアンサンブル垂種は、安全なスケジューリングに不可欠。

実装ブループリント (実践的ステップ)

1. データ収集：エンコーダ、電流、温度、IMU、指令トルクの同期トピックをログする。
2. ラベル付け：温度、トルクリップ、または追従誤差に故障閾値を定義し、ラベル付き故障から逆算して RUL を計算する。
3. 拡張：Omniverse / Isaac Sim で故障モードを生成し、稀な故障クラスを拡張する。
4. 訓練：ジョイントごとに正規化し、時系列で分割し、転移可能性を評価するために未見のロボットで評価する。

代表的なモデル訓練スニペット (PyTorch)。この例は、スライディングウィンドウ特徴から RUL 推定のための LSTM 回帰器を訓練する。

コードサンプル 161 LSTM training loop for joint RUL prediction

```
import os
import random
from typing import Tuple

import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter

# 再現性確保
def seed_everything(seed: int = 42) -> None:
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
```

```

torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

```

```

class LSTMRegressor(nn.Module):

```

```

    """

```

```

    LSTM 回帰モデル

```

```

    """

```

```

    """

```

```

    def __init__(self, feat_dim: int, hidden_dim: int = 64, num_layers: int = 2, dropout: float = 0.5):
        super().__init__()
        self.lstm = nn.LSTM(
            feat_dim,
            hidden_dim,
            num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0.0,
        )
        self.fc = nn.Linear(hidden_dim, 1)

```

```

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: (B, T, F)
        _, (h_n, _) = self.lstm(x)
        out = self.fc(h_n[-1]) # 最終層の隠れ状態を使用
        return out.squeeze(-1) # (B,)

```

```

    def train_one_epoch(
        model: nn.Module,
        loader: DataLoader,
        criterion: nn.Module,
        optimizer: torch.optim.Optimizer,
        device: torch.device,
    ) -> float:
        model.train()
        total_loss = 0.0
        for x, y in loader:
            x, y = x.to(device, non_blocking=True), y.to(device, non_blocking=True)
            optimizer.zero_grad()
            pred = model(x)

```

```

        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * x.size(0)
    return total_loss / len(loader.dataset)

```

```

@torch.no_grad()
def validate(
    model: nn.Module,
    loader: DataLoader,
    criterion: nn.Module,
    device: torch.device,
) -> float:
    model.eval()
    total_loss = 0.0
    for x, y in loader:
        x, y = x.to(device, non_blocking=True), y.to(device, non_blocking=True)
        pred = model(x)
        total_loss += criterion(pred, y).item() * x.size(0)
    return total_loss / len(loader.dataset)

```

```

def main():
    seed_everything(42)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # データローダの作成は外部で定義済みと仮定
    # train_loader, val_loader = ...

    model = LSTMRegressor(feet_dim=32).to(device)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=5, fact

    writer = SummaryWriter(log_dir="runs/lstm_rul")
    best_val_loss = float("inf")
    epochs = 50

    for epoch in range(1, epochs + 1):

```

```

train_loss = train_one_epoch(model, train_loader, criterion, optimizer, device)
val_loss = validate(model, val_loader, criterion, device)
scheduler.step(val_loss)

writer.add_scalar("Loss/train", train_loss, epoch)
writer.add_scalar("Loss/val", val_loss, epoch)

if val_loss < best_val_loss:
    best_val_loss = val_loss
    torch.save(model.state_dict(), "best_lstm_rule.pt")

print(f"Epoch_{epoch:03d}: train_loss={train_loss:.4f}, val_loss={val_loss:.4f}")

writer.close()

if __name__ == "__main__":
    main()

```

評価指標と運用閾値：

- RUL 回帰には RMSE と MAE を使用する。
- 2 値異常検出には F1 スコアと ROC-AUC を使用する。
- アラームが遅れることを早期より重く罰するタイムリネススコアを組み込む。
- 不確実性区間を報告し、上限リスクが閾値を超える場合は保守的なメンテナンストリガーを使用する。

エンジニアリングへの影響、トレードオフ、リスク：

- データ不均衡：故障は稀である。シミュレーション拡張とコスト感受性訓練を使用する。
- ドメインギャップ：合成動力学は実世界の摩擦と異なる。複数のハードウェアユニットで検証する。
- 偽陽性は不要なメンテナンスを強制し、稼働率を低下させる。アラーム閾値を運用リスク許容度に較正する。
- 偽陰性はミッション故障と安全インシデントをリスクする。不確実性が高い場合は保守的スケジューリングを優先する。
- センサ故障はコンポーネント故障を模倣する。センサヘルスマニタとセンサ間一貫性チェックを含める。
- 計算コスト：オンボード推論は遅延と電力予算を満たさなければならない。エッジ量子化またはロボット上アクセラレータを検討する。

運用推奨事項：

1. 予測出力をメンテナンス計画ツールおよびヒューマンインザループ検証と統合する。
2. スケジューリングには不確実性を意識したモデルを使用し、モデル信頼度が低い場合は検査をエスカレーションする。
3. フィールドデータで継続的に再訓練し、プライバシーを保護しながら fleet 用に federated 更新を使用する。

49 宇宙と探査におけるロボティクス

49.1 宇宙ミッションにおけるヒューマノイド

本章で議論した AI を活用した制御と高忠実度シミュレーションの概念を踏まえ、本小節では宇宙ミッションにおけるヒューマノイドの運用上の課題にそれらを適用する。焦点は、ヒューマノイドを効果的かつ頑健なミッション資産にする実用的な設計、制御、センシング、検証手法にある。

ヒューマノイドの役割とミッション要求

- 典型的な役割：船外保守、船内物流、惑星表面での科学サンプリング、宇宙飛行士支援、テレプレゼンス。
- 主要な設計要因：質量効率、フォールトトレランス、低消費電力アクチュエーション、耐放射線電子機器、人間または繊細なハードウェアとの作業時の予測可能な相互作用力。
- 運用上の制約：長い通信遅延、極端な熱サイクル、真空、研磨性レゴリス、修理の機会が限られること。

問題定義微小重力および部分重力環境でコンタクトリッチなタスクを実行できるヒューマノイドを設計せよ。ロボットは体接触による安定化時の全身ダイナミクスを管理し、宇宙船の姿勢を変えずに大きな反力を扱い、乗組員およびハードウェアとの安全な相互作用を維持しなければならない。

技術分析：ダイナミクスと制御ロボットの関節ダイナミクスとコンタクトレンチは、制御および計画の基礎でなければならない。標準的な剛体方程式が適用される：

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J_c(q)^T f_c, \quad (387)$$

ここで q は関節構成、 M は質量行列、 C はコリオリ行列、 g は重力（自由落下では小さいまたはゼロ）、 τ はアクチュエータトルク、 J_c はコンタクトヤコビアン、 f_c はコンタクトレンチである。

主要な制御上の考慮事項：

1. 自由浮遊セグメントでは運動量保存が支配的である。宇宙船との角および線運動量の結合を管理する。重心運動量 H を用いて $\dot{H} = \text{external_wrenches}$ を強制する。
2. 微小重力では ZMP（ゼロモーメントポイント）定式化は意味を失う。代わりにコンタクト力の実現可能性と運動量制御に依存する。
3. 全身制御はアクチュエータ限界と宇宙船反力制約を満たさなければならない。制御を制約付き二

次計画問題（QP）として定式化する：

$$\begin{aligned}
 & \min_{\tau, f_c} \|W_\tau(\tau - \tau_{\text{des}})\|^2 + \|W_f(f_c - f_{\text{des}})\|^2 \\
 & [H] \quad \text{s.t.} \quad M\ddot{q} + C\dot{q} + g = \tau + J_c^\top f_c, \\
 & \quad \quad A_{\text{att}}(J_{\text{base}}\ddot{q} + \dots) \leq b_{\text{att}}, \\
 & \quad \quad \tau_{\min} \leq \tau \leq \tau_{\max}, \quad f_{\min} \leq f_c \leq f_{\max}.
 \end{aligned} \tag{388}$$

この定式化はダイナミクス、姿勢または反力運動量制約、アクチュエータ／コンタクト限界を強制する。

アクチュエーションと機械設計

- ・アクチュエータの選択：シリーズエラスティックアクチュエータ（SEA）はコンプライアントなコンタクト制御と衝撃耐性を提供する。ハーモニックドライブは高減速を提供するが、耐放射線潤滑と注意深い熱設計が必要である。
- ・構造：荷重経路には軽量カーボン複合材およびチタンを用いる。軌道上交換を可能にするため、モジュール化された冗長リムセグメントを用いる。
- ・熱と真空：熱伝導経路と放射のみの放熱を用いる。対流依存の冷却は避ける。

知覚とセンシング

- ・ビジョンは高コントラスト照明とダストに対処しなければならない。HDR カメラと偏光フィルタを用いる。
- ・絶対位置決め：IMU、スタートラッカー手がかり、視覚的地標を組み合わせる頑健な姿勢推定を行う。
- ・指および手のひらの触覚センシングは安全なマニピュレーションおよび力制御にとって重要である。

自律性、テレオペレーション、ヒューマンインザループ制御

- ・通信遅延は予測ディスプレイ、共有自律性、監督コマンドを要求する。モデル予測制御（MPC）は遅延ウィンドウにわたって計画できる。
- ・階層化された自律性を実装する：低レベルインピーダンス制御、ミドルレベルコンタクトプランナ、ハイレベルタスクプランナおよびフォールトマネージャ。
- ・テレオペレーションでは、遅延を補償するために人間オペレータにシミュレートされた予測状態を提供する。

シミュレーションと検証

- ・物理ベースシミュレーション（例：Isaac Sim）を、訓練および検証のための正確な微小重力、コンタクト、ダストモデルとともに用いる。デジタルツインは複雑な保守シーケンスの飛行前検証を可能にする。
- ・フライトテスト：微小重力検証のための放物線飛行、手順リハーサルのための中性浮力、長期挙動のための ISS 実績実験。

実用的な制御実装（例）以下のスニペットは、所望のエンドエフェクタ力を維持しながらヌル空間

姿勢を保持するための全身トルク分割を計算する。擬似逆行列とヌル空間射影を使用し、QP ソルバが利用できないオンボード監督制御に適している。

コードサンプル 162 所望エンドエフェクタ力のためのヌル空間トルク分配

```
import numpy as np
from typing import Optional, Tuple

def compute_whole_body_torque(
    J: np.ndarray,
    M: np.ndarray,
    q_dot: np.ndarray,
    f_des: np.ndarray,
    tau_posture_des: np.ndarray,
    torque_limits: Optional[Tuple[float, float]] = None,
    damping: float = 1e-4
) -> np.ndarray:
    """
    エンドエフェクタ力とポーズ制御トルクを統合し、関節トルク指令を生成
    """
    if torque_limits is None:
        torque_limits = (-100.0, 100.0)

    # タスク空間トルク:  $J^T * f_{des}$ 
    tau_task = J.T @ f_des

    # ダンピング擬似逆行列によるヌル空間射影
    JJt = J @ J.T
    JJt += damping * np.eye(JJt.shape[0])
    J_pinv = np.linalg.solve(JJt, J).T # (n, 6)
    N = np.eye(J.shape[1]) - J_pinv @ J # ヌル空間射影行列

    # 最終トルク指令
    tau_cmd = tau_task + N @ tau_posture_des

    # 飽和处理
    tau_cmd = np.clip(tau_cmd, *torque_limits)

    return tau_cmd
```

設計のトレードオフと運用上のリスク

- ・質量対能力：マニピュレータおよび放射線遮蔽の追加は質量と打ち上げコストを増加させる。ミッション固有のタスクのためのモジュールペイロードを優先する。
- ・自律性対検証：高い自律性は乗組員負荷を減らす、検証負担を増加させる。段階的な自律性認証を用いる。
- ・反力結合：大きなマニピュレーション力は宇宙船姿勢を攪乱できる。反力制御戦略と安全なコンタクトエンベロープを要求する。
- ・故障モード：アクチュエータ焼き付き、放射線によるセンサ劣化、ダスト摩耗。グレースフルデグラデーション、冗長センシング、安全なデフォルト挙動を設計する。

エンジニアリングへの影響

- ・運動量制約を伴う全身コンタクト認識制御は、安全な宇宙ヒューマノイド運用に不可欠である。頑健なシミュレーションと階層化された自律性はミッションリスクを低減する。トレードオフはミッションクリティカルなタスク、質量予算、長期ミッションの保守性を優先すべきである。

49.2 ロボティック探査の課題

前節で議論したヒューマノイドプラットフォームの運用役割とミッションプロファイルを踏まえ、本小節では惑星および深宇宙探査における有効性を制限する具体的な技術的課題を検討する。焦点は、実際のミッションにおける機械設計、センシング、自律性、およびミッション計画に直接影響を与える制約である。

問題の定義と運用制約

- ・ヒューマノイド探査機は、厳格な質量およびエネルギー予算、劣化した通信、極限環境、保守の機会が限られるという条件下で動作しなければならない。
- ・主要な性能指標には、ミッションあたりの移動可能距離、地図の完全性、資産キャッシュまたは居住施設への安全な帰還確率が含まれる。

技術分析

1. エネルギーと質量のトレードオフ。アクチュエータ、計算、熱制御がエネルギー消費を支配する。簡略化されたエネルギー予算は次のように表される：

$$[H]E_{\text{total}} = \int_0^T (P_{\text{mot}}(t) + P_{\text{compute}}(t) + P_{\text{comm}}(t) + P_{\text{thermal}}(t)) dt, \quad (389)$$

ここで T はミッション期間である。初期設計サイジングでは、平均電力の近似により $E_{\text{total}} \approx P_{\text{avg}}T$ となる。必要なバッテリー質量は

$$[H]m_{\text{batt}} = \frac{E_{\text{total}}}{\rho_{\text{energy}}\eta}, \quad (390)$$

で与えられ、 ρ_{energy} はセルのエネルギー密度、 η はシステムレベルの変換効率である。この関係は、自律性と構造質量の直接的な結合を定量化する。

2. ドリフトとスパース特徴下での位置推定とマッピング。ビジュアル慣性オドメトリ（VIO）および SLAM は 2 つの相互作用する故障モードに直面する：センサ劣化（塵、グレア、放射線誘起ノ

イズ)と積分ドリフト。離散時間共分散伝播は推定の増大を捉える：

$$P_{k+1} = F_k P_k F_k^T + Q_k,$$

ここで F_k は運動ヤコビアン, Q_k はプロセスノイズである。定加速度誤差として扱われる IMU バイアス b に対して, 位置ドリフトは $\frac{1}{2}bt^2$ に比例する。この 2 次増大は, ループクロージャ, 外部ビーコン, または断続的な絶対修正の必要性を強調する。

3. 低重力および不規則地形における移動と接触計画。移動ダイナミクスは重力 g とともに著しく変化し, スタンス支持に必要な関節トルクは $\tau \propto mgr$ に比例する。低重力では重量が減少するが, 衝撃的な擾乱に対する感度が増加し, 接地制御および回復を複雑化する。コンプライアンス制御は, 精度のための剛性と低重力衝撃からのリバウンド回避のための減衰とのバランスを取らなければならない。
4. 遅延通信下での自律性。往復遅延はオンボード意思決定を強制する。自律アーキテクチャは：
 - 新規危険を検出し局所的に再計画する。
 - ダウンリンクのためにテレメトリを圧縮し優先付けする。
 - 不確実性下で安全に動作を劣化させる。

確率的风险指標, たとえば期待ミッション終了故障率は, 自律性切替えの閾値選択に情報を提供しなければならない。

5. 放射線および熱的極限下での信頼性。電子部品の故障率は線量および温度サイクルとともに増加する。冗長性および投票論理は単一点故障を減少させるが, 質量およびエネルギー消費を増加させる。

実装 — 実用的アルゴリズムの考察

- モジュール自律性を優先する：低レベル反射を長期ホライズンプランナから分離する。反射モジュールは高周波で境界付き計算を実行；熟考プランナは機会的に実行する。
- エネルギー認識計画を含める：コスト関数にバッテリー状態を組み込む。ミッション終了の電力枯渇を回避するため保守的マージンを用いる。

Python による最小エネルギー認識プランナループは, 制約チェックと単純な貪欲ウェイポイント選択を例示する。これはオンボードで実行され, 移動するか充電のために待機するかを決定する。

コードサンプル 163 実地運用のためのエネルギー認識ウェイポイントセクタ

```
import math
from typing import List, Tuple, Optional, NamedTuple
import logging

# ROS 2 (rclpy) 対応
try:
    import rclpy
    from geometry_msgs.msg import Pose, Twist
except ImportError:
    rclpy = None
```

```

    Pose = None
    Twist = None

# ログ設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class RobotState(NamedTuple):
    """ロボットの状態（位置・姿勢・速度）"""
    pose: Tuple[float, float, float] # x, y, yaw [rad]
    velocity: Tuple[float, float, float] # vx, vy, omega [rad/s]

class Waypoint(NamedTuple):
    """ウェイポイント情報"""
    pose: Tuple[float, float, float]
    science_value: float # 科学的価値（0.0-1.0）

def estimate_overhead(wp: Waypoint) -> float:
    """
    センサ・通信オーバーヘッドを推定（単位：ジュール）
    """
    # 簡易モデル：距離に応じた通信コスト + 観測対象の複雑さ
    base_comm = 5.0 # 基本通信コスト
    obs_complexity = 1.0 + wp.science_value # 科学的価値が高いほど観測コスト増
    return base_comm * obs_complexity

def select_next_waypoint(
    state: RobotState,
    waypoints: List[Tuple[Waypoint, float]],
    battery_joules: float,
    safety_factor: float = 0.15
) -> Optional[Waypoint]:
    """
    バッテリ残量と科学的価値を考慮して次のウェイポイントを選択
    """
    if battery_joules <= 0:

```

```

        logger.warning("バッテリー残量が0以下です")
        return None

    safe_margin = safety_factor * battery_joules
    candidates: List[Tuple[Waypoint, float]] = []

    for wp, cost in waypoints:
        overhead = estimate_overhead(wp)
        total_cost = cost + overhead

        # 安全マージンを考慮
        if total_cost + safe_margin <= battery_joules:
            # 科学的価値で重み付け（価値が高いほどコストを下げる）
            weighted_cost = total_cost / max(wp.science_value, 0.01)
            candidates.append((wp, weighted_cost))

    if not candidates:
        logger.info("実行可能なウェイポイントがありません")
        return None

    # 重み付きコスト最小を選択
    best_wp, _ = min(candidates, key=lambda x: x[1])
    logger.info(f"次のウェイポイントを選択: {best_wp.pose}")
    return best_wp

```

センサおよび知覚設計推奨

- 相補的モダリティを組み合わせる：ステレオまたはイベントカメラ，短距離ライダ，地下レーダを地下特徴用に。
- 適応露光およびフィルタリングを実装し，極端なコントラストに対処する。
- スパース地図表現および特徴ハッシュを用い，メモリおよび計算コストを制限する。

テストおよび検証戦略

- 放射線および熱モデルを含むフルスタックシミュレーションを，Isaac Sim のような物理リッチ環境で実施する。
- 劣化通信シナリオを注入し，遠隔監督を絞り込み，自律性閾値を検証する。
- ハードウェアインザループサイクルを用い，モータおよびバッテリーへの熱ソーク効果を検証する。

工学への影響，トレードオフ，およびリスク

- トレードオフは質量－エネルギー予算の緊張によって支配される：より多くの計算は自律性を改

善するが、式 (401) を通じて m_{batt} を増加させる。

- 冗長性はミッションリスクを減少させるが質量を増加させ、収穫逓減最適化を生じる。
- 高忠実度知覚への過度の依存は、塵または低特徴地形で脆さを増加させる；保守的動作が必要となる場合がある。
- 運用上のリスクには、電力枯渇による不可逆ミッション終了、累積ドリフトによる位置推定喪失、放射線による部品故障が含まれる。緩和には、機械設計、センサ冗長性、およびエネルギー認識自律ポリシーを統合した層横断的戦略が必要である。

49.3 宇宙ヒューマノイドの将来の可能性

前述の運用上の制約とミッション役割を踏まえ、これらの制限が宇宙におけるヒューマノイドシステムの具体的な技術方向をどう可能にするかを考察する。焦点は、将来の探査タスクで測定可能な能力をもたらす具体的な設計選択、制御パラダイム、ミッションアーキテクチャにある。

問題定義. 長期・遠隔ミッションでは、ヒューマノイドが継続的な人間の監視なしに複雑な巧緻タスクを実行する必要がある。タスクには、大型構造物のオンビット組立、惑星表面での居住施設建設、船外活動（EVA）の拡張、現地資源利用（ISRU）、危険環境での科学的現地調査が含まれる。主要な技術的問いは、能力との質量トレードオフ方法、放射線と真空における信頼性確保方法、検証可能な自律アーキテクチャの設計方法である。

技術分析.

- ハードウェアモジュール性と再構成性: モジュール性は修理時間を短縮し、ミッション適応ツーリングを支援する。駆動手首、交換可能なハンドエンドエフェクタ、センサペイロードポッド、電源モジュールといった設計モジュールを用意する。重要アクチュエータに対する並列冗長アプローチはシステム信頼性を高める。独立モジュール信頼性 r_i に対し、並列システム信頼性 R_{parallel} は

$$[H]R_{\text{parallel}} = 1 - \prod_{i=1}^n (1 - r_i). \quad (391)$$

この式は、質量制約下で含める複製モジュールの数と品質を導く。

- 電力・エネルギー予算: エネルギーは運用デューティサイクルと移動戦略に影響する。ヒューマノイドが使用可能エネルギー E を搭載し、平均運用電力が P の場合、連続運転時間 T は $T = E/P$ を満たす。断続的高電力タスクには、予備コンデンサまたはフライホイール支援運動でピーク電力を抑制する。
- 自律階層と検証: 自律を、リアルタイムの人間入力なしに知覚・計画・行動する能力と定義する。安全性を保証するには階層アーキテクチャが必要：
 1. 低レベル反射（ハードリアルタイム、安全臨界）。
 2. 中レベル運動プランナ（MPC — モデル予測制御 — 安全エンベロープ付き）。
 3. 高レベルタスクプランナ（記号プランナまたは人間上書き付きビヘイビアツリー）。

各層は、シミュレーションおよびハードウェアインザループ試験で検証可能な形式不変条件を公開しなければならない。

- 分散マルチエージェント協調: 複数のヒューマノイドと非ヒューマノイド資産が異種チームを形成する。通信遅延に対処するため、タスク割当てに市場ベースまたは合意アルゴリズムを用い

る。効用 U_j をもつタスクを N 台のエージェントが実行する場合、オークション機構は帯域幅制限下で効率的な割当てを保証する。

実装スケッチ. 以下の Python スニペットは、軌道組立タスクで複数ヒューマノイドのための簡単なエネルギー考慮オークションアロケータを示す。各エージェントが残存エネルギーとタスク効用セットを報告することを仮定する。

コードサンプル 164 複数ヒューマノイド組立のためのエネルギー考慮オークションアロケータ

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from __future__ import annotations
import logging
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Tuple

# ログ設定：本番環境ではINFO以上、開発時はDEBUGに切り替え
logging.basicConfig(level=logging.INFO, format="%(asctime)s_[(levelname)s]_(message)")
logger = logging.getLogger(__name__)

@dataclass(slots=True)
class Agent:
    """ エージェントの状態を保持 """
    aid: int
    energy_wh: float
    reserved: bool = field(default=False, init=False)

    def can_perform(self, cost: float) -> bool:
        return not self.reserved and self.energy_wh >= cost

@dataclass(slots=True)
class Task:
    """ タスクの情報を保持 """
    tid: str
    utility: float

@dataclass(slots=True)
class Bid:
```

```
"""入札情報"""
```

```
agent: Agent
```

```
task: Task
```

```
value: float
```

```
cost: float
```

```
class Auctioneer:
```

```
"""単純な貪欲オークション実装"""
```

```
def __init__(self, agents: List[Agent], tasks: List[Task]) -> None:
```

```
    self.agents = {a.aid: a for a in agents}
```

```
    self.tasks = {t.tid: t for t in tasks}
```

```
def estimate_cost(self, agent: Agent, task: Task) -> float:
```

```
    """タスク実行に必要な推定エネルギー消費[Wh]"""
```

```
    # 実機では経路・動作履歴から推定
```

```
    if task.tid == "A":
```

```
        return 20.0
```

```
    if task.tid == "B":
```

```
        return 10.0
```

```
    return 15.0 # デフォルト
```

```
def compute_bid_value(self, agent: Agent, task: Task, cost: float) -> float:
```

```
    """効用から正規化コストを差し引いた入札値"""
```

```
    return task.utility - (cost / agent.energy_wh) * task.utility
```

```
def run(self) -> Dict[str, int]:
```

```
    """降順貪欲割当てを実行"""
```

```
    bids: List[Bid] = []
```

```
    # 全組み合わせで入札を生成
```

```
    for agent in self.agents.values():
```

```
        for task in self.tasks.values():
```

```
            cost = self.estimate_cost(agent, task)
```

```
            if agent.can_perform(cost):
```

```
                value = self.compute_bid_value(agent, task, cost)
```

```
                bids.append(Bid(agent, task, value, cost))
```

```
    # 入札値の降順でソート
```

```

        bids.sort(key=lambda b: b.value, reverse=True)

    allocation: Dict[str, int] = {}
    for bid in bids:
        if bid.agent.reserved or bid.task.tid in allocation:
            continue
        allocation[bid.task.tid] = bid.agent.aid
        bid.agent.reserved = True
        logger.info("Task_%s->Agent_%d(value=%.2f)", bid.task.tid, bid.agent.aid, bid.value)

    return allocation

# ----- 使用例 -----
if __name__ == "__main__":
    agents = [Agent(aid=0, energy_wh=120.0), Agent(aid=1, energy_wh=80.0)]
    tasks = [Task(tid="A", utility=100.0), Task(tid="B", utility=60.0)]

    auction = Auctioneer(agents, tasks)
    result = auction.run()
    print("Allocation:", result)

```

制御・知覚の進展. 頑強な宇宙運用には以下が必要:

- 放射線硬化 SLAM (同時位置特定とマッピング) スタック、ダスト・低照度に耐性。SLAM を、オドメトリと外感受性センサの融合により一貫した地図を構築するものと定義する。
- カ-トルクフィードバックと学習方策を用いた適応操縦。強化学習と MPC の組合せにより、不確実な固定具との準拠インタラクションを可能にする。
- チーム向け連合学習: エージェントは圧縮モデル更新を交換し、帯域幅を保持しながら現地条件に適応する。

運用アーキテクチャとミッションコンセプト.

- 大型アパーチャ・トラスの組立: ヒューマノイドが微細操縦を行い、より大型のロボットクレーンがマクロ配置を担う。人間監督者が組立チェックポイントを検証し、遠隔操縦ラウンドを最小化する。
- 現地製造・修理: 軽量プリンタと予備モジュール在庫を統合する。ヒューマノイドが検査、材料堆積、モジュール交換を実行する。
- EVA 拡張と人間安全: ヒューマノイドが工具運搬、構造物安定化、高リスク作業中の遠隔存在プロキシとして宇宙飛行士を支援する。

技術的影響、トレードオフ、リスク.

- ・質量対自律: 計算・放射線遮蔽の追加は質量・電力需要を増やす。設計者は現地自律と地上制御依存を釣り合わせる必要がある。
- ・冗長性対複雑性: 冗長モジュールを増やすと (391) に従い信頼性は向上するが、インタフェースと統合リスクが増える。
- ・エネルギー供給: 大型バッテリーや原子炉は運用を延長するが、打ち上げコストと熱管理課題を増やす。
- ・検証負荷: 高自律は応答性を高めるが、検証複雑性を上げる。形式手法はリスクを下げるが開発時間を増やす。
- ・運用リスク: ソフトウェア故障、微小隕石衝突、汚染はモジュールシステムに連鎖する。緩和には故障検出、グレースフルデグラデーション、オンビット修理計画が必要。
- ・法的・倫理的考慮: 不可逆的な行動を行う自律ヒューマノイドには、ミッション交戦規定と遠隔監督プロトコルが必要。

設計者はこれらのトレードオフを早期に定量化し、シミュレーションインザループ検証を用い、運用信頼性に応じて自律を段階的に高める展開を計画すべきである。

50 新たな応用

50.1 エンターテインメントおよびゲーミングにおけるロボット

AI 有効化制御と配備制約についての先行議論を踏まえ、本小節ではエンターテインメントおよびゲーミングに用いられるヒューマノイドロボットの実践的なエンジニアリングを考察する。表現的な動作、インタラクティブな応答性、安全で信憑性のある体験を実現するために必要なシステムレベルのトレードオフに焦点を当てる。

問題定義：ライブショーおよび没入型ゲームにおいて、俳優、インタラクティブな非プレイヤーキャラクター（NPC）、または協調パートナーとして説得力をもって動作するヒューマノイドプラットフォームを設計する。主要な運用要件は、人間の入力への低遅延応答、振付のための高い再現性、表現的な身振りの広いダイナミックレンジ、密集した人間の近接における堅牢な安全性である。

技術分析：

- ・性能指標：
 1. 応答性：知覚されるインタラクティブティには、ループ遅延 L を知覚閾値以下に抑える必要がある。保守的なエンジニアリング目標は、身振り駆動インタラクションについて $L \leq 50 \text{ ms}$ である。
 2. 帯域：標準化周波数 f_s は、最も高い有意人間運動周波数 f_{\max} の 2 倍を超えなければならない（ナイキスト）。表現的上半身動作について $f_{\max} \approx 5 \text{ Hz}$ であるため、

$$[H]f_s > 2f_{\max} \approx 10 \text{ Hz}, \quad (392)$$

と設定するが、トルクレベル制御のために制御およびセンシングは通常 100–500 Hz で動作する。

3. 忠実度 vs エネルギー：より高いアクチュエータ帯域およびトルクは忠実度を高めるが、熱および電力要求を増大させる。

- 制御アーキテクチャ：振付済みセグメントのための軌道再生と、インタラクションのための反応的全身制御を組み合わせる。実用的なトルク指令は、フィードバック項とフィードフォワード項を結合する：

$$[H]\tau = K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}) + \tau_{ff}, \quad (393)$$

ここで q_d は所望関節角度、 q は計測角度、 τ_{ff} は重力および計画ダイナミクスを補償する。

- 動作合成：知覚的信憑性には、階層化された合成が必要である：
 1. モデルベース全身プランナによる基礎移動および姿勢。
 2. 身振りプリミティブをシーケンス化し、手／目ターゲティングのための逆運動学（IK）でブレンド。
 3. マイクロ表情および頭／目動作を確率的に生成し、機械的周期性を回避。
 ブレンド重みは関節限界および衝突制約を尊重すべきである。

実装指針：

- 知覚：オンボードビジョンおよび深度センサを用いて観客の意図および位置を検出。IMU、関節エンコーダ、およびビジョン推定値を拡張カルマンフィルタ（EKF）で融合し、堅牢な状態推定を行う。
- 遅延補償：センサ遅延が存在する場合、単純な線形予測器を用いて短期人間動作を予測する。 Δt を計測遅延とすると、将来目標姿勢を $\hat{x}(t + \Delta t) = x(t) + \dot{x}(t)\Delta t$ と予測する。
- シミュレーション優先検証：物理ベースシミュレータで振る舞いを訓練・反復し、配備前に安全包絡およびバッテリー消費を検証する。シミュレートされたモーションキャプチャ（mocap）データは、多様な身振りセット生成に役立つ。
- 安全層：
 1. 関節トルクおよび速度に対するソフト限界。
 2. 近接センサが優雅な停止またはフォールバック振る舞いをトリガ。
 3. センサ異常または発散状態推定を検出するためのランタイムモニタリング。

表現的ブレンディングおよび遅延補償のための具体的アルゴリズムパターン（ROS2 風の Python 疑似コード）。ループは高レートインナー制御と低レート表現プランナを実行する。

コードサンプル 165 表現的ジェスチャブレンディングと遅延補償（簡略版）

```
import time
import threading
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import Float64MultiArray, Bool
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
import numpy as np
from typing import Tuple, Optional
```

```

class GestureEngine(Node):
    def __init__(self) -> None:
        super().__init__('gesture_engine')

        # QoS設定（リアルタイム制御に適したプロファイル）
        rt_qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        # パラメータ読み込み
        self.declare_parameter('loop_hz', 200)
        self.declare_parameter('joint_names', [
            'shoulder_pan', 'shoulder_lift', 'elbow', 'wrist_1', 'wrist_2', 'wrist_3'
        ])
        self.loop_hz = self.get_parameter('loop_hz').value
        self.dt = 1.0 / self.loop_hz
        self.joint_names = self.get_parameter('joint_names').value

        # パブリッシャ／サブスクライバ
        self.joint_cmd_pub = self.create_publisher(
            JointTrajectory, '/joint_trajectory_controller/joint_trajectory', rt_qos
        )
        self.joint_state_sub = self.create_subscription(
            JointState, '/joint_states', self.joint_state_cb, rt_qos
        )
        self.estop_pub = self.create_publisher(Bool, '/emergency_stop', rt_qos)

        # 内部状態
        self.latest_state: Optional[JointState] = None
        self.lock = threading.Lock()

        # 外部インターフェース（ダミー実装、実機ではROSトピック／サービスに置換）
        self.sensor_interface = SensorInterface()
        self.controller = Controller(self.joint_names)
        self.planner = Planner()

        # 制御ループタイマー

```

```

        self.timer = self.create_timer(self.dt, self.control_loop)

def joint_state_cb(self, msg: JointState) -> None:
    with self.lock:
        self.latest_state = msg

def control_loop(self) -> None:
    t0 = self.get_clock().now()

    # 最新の関節状態取得
    with self.lock:
        if self.latest_state is None:
            return
        state = np.array(self.latest_state.position)

    # センサ状態読み取り
    full_state = self.sensor_interface.read_state()
    audience = self.sensor_interface.estimate_audience()

    # 観客姿勢の遅延補償
    pred_aud = audience.pose + audience.vel * audience.measured_latency

    # 目標関節角度と表情ウェイト取得
    q_d, expr_w = self.planner.get_target(pred_aud)

    # マイクロ表情オーバーレイをブレンド
    micro = self.planner.micro_expression(expr_w)
    q_blend = q_d * (1 - expr_w) + (q_d + micro) * expr_w

    # トルク計算
    tau = self.controller.compute_torque(q_blend, full_state)

    # 指令送信
    traj = JointTrajectory()
    traj.joint_names = self.joint_names
    point = JointTrajectoryPoint()
    point.positions = q_blend.tolist()
    point.time_from_start.sec = 0
    point.time_from_start.nanosec = int(self.dt * 1e9)
    traj.points.append(point)

```

```

self.joint_cmd_pub.publish(traj)

# 安全チェック
if not self.sensor_interface.ok():
    self.estop_pub.publish(Bool(data=True))
    self.get_logger().error('Emergency_stop_triggered!')
    return

# 周期維持
elapsed = (self.get_clock().now() - t0).nanoseconds * 1e-9
sleep_time = self.dt - elapsed
if sleep_time > 0:
    time.sleep(sleep_time)

class SensorInterface:
    def read_state(self):
        # 実機ではROSトピックから状態を取得
        return type('State', (), {
            'encoders': np.zeros(6),
            'imu': np.eye(3),
            'vision': np.zeros(3)
        })()

    def estimate_audience(self):
        # 実機ではVisionトピックから推定
        return type('Audience', (), {
            'pose': np.array([1.0, 0.0, 0.0]),
            'vel': np.array([0.1, 0.0, 0.0]),
            'measured_latency': 0.05
        })()

    def ok(self) -> bool:
        # センサ死活監視
        return True

class Controller:
    def __init__(self, joint_names):
        self.joint_names = joint_names

```

```

        self.kp = np.array([100.0] * len(joint_names))
        self.kd = np.array([10.0] * len(joint_names))

    def compute_torque(self, q_d, state):
        # PD + フィードフォワード (簡易実装)
        q = state.encoders
        dq = np.zeros_like(q) # 実機では速度取得
        return self.kp * (q_d - q) - self.kd * dq

class Planner:
    def get_target(self, pred_aud):
        # ジェスチャプリミティブ選択 (ダミー)
        q_d = np.array([0.0, -0.5, 1.0, -1.5, 0.0, 0.0])
        expr_w = 0.3
        return q_d, expr_w

    def micro_expression(self, expr_w):
        # 微小オフセット生成
        return np.random.uniform(-0.05, 0.05, 6) * expr_w

def main(args=None):
    rclpy.init(args=args)
    node = GestureEngine()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

設計トレードオフおよびエンジニアリング含意：

- アクチュエータ選定：直列弾性アクチュエータは安全性および低インパルス接触を改善するが、生の帯域を低下させる。目標性能包絡に基づき駆動を選択する。

- ・知覚忠実度 vs プライバシー：高忠実度センシングはインタラクティビティを改善するが、公共の場での配備においてプライバシーおよび規制上の懸念を提起する。
- ・振付 vs 適応性：純粋にスクリプト化されたルーチンは再現性を最大化するが、予期しないインタラクションでは失敗する。ハイブリッドシステムは複雑さを加えるが堅牢性を高める。
- ・エネルギー供給：長時間ショーにはバッテリー管理またはテザード運用が必要であり、移動性および設置ロジスティクスに影響する。
- ・摩耗および保守：表現的動作は関節および伝達機構にストレスを与える。公共のエンターテインメントで使用する場合、高頻度予防保守をスケジュールする。

運用上のリスク：

- ・アンカニーバレー効果は、動作タイミングまたは顔表情に一貫性がない場合、観客のエンゲージメントを低下させる。
- ・ネットワーク化制御は単一障害点を導入する。保守的な安全振る舞いへ優雅に劣化させる。
- ・群衆ダイナミクスはヒューマノイドプラットフォームに予測不能な力を及ぼす可能性がある。機械的コンプライアンスおよび堅牢な衝突ハンドリングを設計する。

エンジニアは表現性、安全性、および運用コストをバランスさせなければならない。シミュレーション検証、階層化された安全性、および性能モニタリングを優先し、信憑性があり信頼でき保守可能なヒューマノイドエンターティナを配備する。

50.2 農業における自律型ロボット

これまでの表現的動作と安全なプロクセミクスの議論を踏まえると、これらの能力は植物との器用でコンプライアントな相互作用、生産物の繊細な取り扱い、人間の労働者との協働作業を要する農業タスクに直接対応する。ヒューマノイドのフォームファクタは、到達可能性、擬人化された操縦スキル、農場労働者にとって直感的なインターフェースをもたらす。

問題設定と運用上の関連性。農業の作業負荷には、繊細な果実の収穫、剪定、苗の移植、植物の健康状態の検査、農場インフラとの相互作用が含まれる。これらのタスクは、屋外の変変条件での知覚、作物損傷を回避するためのコンプライアントな操縦、凸凹の地形での移動、長時間の自律運用を要求する。エンジニアリングの目標は、収穫スループットを最大化しながら、作物損傷、エネルギー消費、人間へのリスクを最小化するヒューマノイドシステムを設計することである。

技術的分析。主要なサブシステムは統合的な検討を要する：

- ・知覚：ビジョンシステムは遮蔽、鏡面ハイライト、葉のクラッタに対処しなければならない。センサフュージョンは RGB、多スペクトル、深度データを統合し、堅牢な検出と熟度推定を実現する。実用的な検出信頼度スコア p_c は、下流の決定と自律的な把握試行のためのゲーティング閾値に情報を提供する。
- ・操縦とコンプライアンス：果実は変形可能であり、過剰な力に敏感である。手首と指に能動的コンプライアンスを実装し、接近中の接触を線形スプリング近似でモデル化する：

$$[H]F = K(x_{\text{des}} - x_{\text{contact}}), \quad (394)$$

ここで K はデカルト剛性行列である。制御は $F \leq F_{\text{max}}$ を強制し、損傷を回避し、 F_{max} は作物ごとに経験的に測定される。

- ・移動と安定性：轍のある畝を横断するには、重心の変位を低く抑え、適応的な足配置を要する。全身制御を用いて、難しい地形での操縦精度と安定性をトレードオフする。
- ・タスク計画と最適化：速度、損傷、エネルギーをバランスさせるスカラー目的を定義する：

$$[H]J = \alpha T + \beta D + \gamma E, \quad (395)$$

ここで T は完了までの時間、 D は予想作物損傷、 E はエネルギー消費である。係数は運用上の優先順位を反映する。

実装上の考慮事項。実用的な開発パイプラインは、統合された3つのスレッドに従う：

1. シミュレーションによる学習とドメイン適応
 - ・ Isaac Sim に植物形態と照明変化を含む高忠実度の果樹園を構築する。
 - ・ 検出とセグメンテーションのための多様な合成データセットを生成する。
 - ・ 把持と全身協調のための RL または模倣方策を学習させる。
 - ・ ドメインランダムマイゼーションとシステム同定を適用し、sim-to-real ギャップを削減する。
2. 知覚からアクションまでのスタック
 - ・ 知覚ノードを使用し、セグメンテーションマスク、熟度スコア、6-DoF 把握仮説を出力する。
 - ・ 確率的フィルタリングを用いてマルチセンサデータを統合し、堅牢な状態推定を実現する。
 - ・ ビヘイビアツリー（GROOT）で動作をシーケンスし、安全なオペレータオーバーライドを許可する。
3. 安全性と相互作用
 - ・ 低レベルコントローラでハード制約を強制する：関節トルク制限、デカルト力上限、速度クランプ。
 - ・ リアルタイムポイントクラウド衝突チェックで局所衝突回避を実装する。
 - ・ 指先の触覚センシングと皮膚様被覆を提供し、人間および植物との安全な物理接触を実現する。

計画時に使用される最小限の実用的コスト評価は、以下として実装できる：

$$[H]D = \sum_{i \in \mathcal{G}} P_{\text{bruise}}(i) \cdot w_i, \quad (396)$$

ここで $P_{\text{bruise}}(i)$ は把握 i が損傷を引き起こす確率であり、 w_i は収穫価値の重みである。

ソフトウェア例。以下のコード抜粋は、ROS2 ベースのヒューマノイド向け Python による知覚からコンプライアンスまでのブリッジを概説する。把握提案の読み取り、剛性スケジュールの計算、コンプライアントなデカルトコントローラへの指令を示す。

コードサンプル 166 ROS2 ノード：単純な把握からコンプライアンスまでのブリッジ

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import PointCloud2
from geometry_msgs.msg import PoseStamped, WrenchStamped
```

```

from std_msgs.msg import Float32, Header
import numpy as np
from typing import Optional

class GraspComplianceNode(Node):
    def __init__(self) -> None:
        super().__init__('grasp_compliance')

        # QoS: センサデータ用に信頼性を確保
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )

        self._sub = self.create_subscription(
            PoseStamped, 'grasp_pose', self.on_grasp, qos)

        self._force_pub = self.create_publisher(
            WrenchStamped, 'cmd_cartesian_force', qos)
        self._stiffness_pub = self.create_publisher(
            Float32, 'cmd_cartesian_stiffness', qos)

        # パラメータ宣言&取得
        self.declare_parameter('force_max', 15.0)          # N
        self.declare_parameter('stiffness_min', 0.5)       # kN/m
        self.declare_parameter('stiffness_max', 1.0)       # kN/m
        self.declare_parameter('confidence_topic', '')

        self.F_MAX: float = self.get_parameter('force_max').value
        self.K_MIN: float = self.get_parameter('stiffness_min').value
        self.K_MAX: float = self.get_parameter('stiffness_max').value

        # confidence購読 (オプション)
        conf_topic: str = self.get_parameter('confidence_topic').value
        self._latest_conf: float = 1.0
        if conf_topic:
            self.create_subscription(
                Float32, conf_topic, self.on_confidence, qos)

```

```

self.get_logger().info("GraspComplianceNode_ready.")

def on_confidence(self, msg: Float32) -> None:
    # confidenceをキャッシュ
    self._latest_conf = max(0.0, min(1.0, msg.data))

def on_grasp(self, msg: PoseStamped) -> None:
    # confidenceに応じて剛性をスケジューリング
    c = self._latest_conf
    K = self.K_MAX - (self.K_MAX - self.K_MIN) * (1.0 - c)
    K = float(np.clip(K, self.K_MIN, self.K_MAX))

    # 力指令はゼロ、制御器がF_MAXを守る
    wrench = WrenchStamped(
        header=Header(stamp=self.get_clock().now().to_msg(),
                      frame_id=msg.header.frame_id))
    # すべての成分をゼロに（既初期化済み）

    stiffness_msg = Float32(data=K)

    self._stiffness_pub.publish(stiffness_msg)
    self._force_pub.publish(wrench)

def main(args: Optional[list] = None) -> None:
    rclpy.init(args=args)
    node = GraspComplianceNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

設計上のトレードオフとエンジニアリングへの影響。優先順位を明示的に選択する：

- 高剛性は位置精度を向上させるが、損傷リスクを高める。
- 高い自律性は労働コストを削減するが、エッジケース処理の複雑さを増す。
- バッテリ容量の増加は稼働時間を延長するが、質量とフットプリントが増加し、土壌圧迫に影響する。

運用上のリスクと緩和策：

- ・作物汚染には、洗浄可能な素材と IP 定格筐体が必要である。
- ・作業員の安全には、可視・聴覚的意図信号と認定済み非常停止が必要である。
- ・天候暴露には、密封されたアクチュエータと熱管理が必要である。

展開評価のための実用的メトリクスには、ロボットあたりのスループット、収穫単位あたりの損傷率、停止時間率、タスクあたりの平均エネルギーが含まれる。これらのメトリクスをコスト関数係数 α, β, γ に用い、実世界の農場経済に合わせて動作を調整する。

50.3 ヒューマノイド設計の境界を押し広める

これまでの例では、実地展開可能な自律性とエンターテインメントにおける表現豊かな動作を示し、頑健性とコンプライアントな駆動が共通の実現要因であることを強調した。本小節では、ヒューマノイド設計をこれらの領域から、機械・電気・制御の統合的イノベーションを要する極限、協働、拡張の役割へと押し広めることに焦点を当てる。

問題定義：非構造化環境で動作しながら、長時間耐久性、全身敏捷性、認証済み安全性を兼ね備えたヒューマノイドを設計する。技術者は、機械強度、アクチュエータ帯域、エネルギー制約、知覚忠実度、形式的安全性保証を同時に満たさなければならない。主要な成果物には、再現可能なサイジング手法、運用電力予算、全身コントローラ的设计ルールが含まれる。

技術分析

- ・ダイナミクスとアクチュエータサイジング。機械設計は剛体ダイナミクスから始まる。必要な関節トルクは

$$[H]\tau(q, \dot{q}, \ddot{q}) = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) + J(q)^\top F_{\text{ext}}, \quad (397)$$

を満たさなければならない。ここで M は質量行列、 C はコリオリ項、 g は重力、 J はヤコビアン、 F_{ext} は外力である。(1) 式はモータトルクとギア比のトレードオフを定義する。ギア減速比を小さくするとバックドライバビリティが保たれるが、モータトルクと電流要件が増加する。

- ・エネルギーと耐久性。ミッション継続時間 T に対し、バッテリー比エネルギー e_{batt} は

$$[H]m_{\text{batt}} e_{\text{batt}} \geq \int_0^T P_{\text{loc}}(t) + P_{\text{man}}(t) + P_{\text{compute}}(t) dt \approx P_{\text{avg}} T. \quad (398)$$

を満たさなければならない。これによりバッテリー質量と利用可能な機械搭載質量の直接的スケールリングが得られる。設計者はアクチュエータ効率と歩容選択により P_{avg} を最適化しなければならない。

- ・多目的最適化。実用的な設計では、相反する目的を最適化する必要がある：質量とコストを最小化し、頑健性と耐久性を最大化し、トルク・熱・安定性制約を満たす。コンパクトな定式化は

$$\begin{aligned} \min_x f(x) &= w_m m(x) + w_c C(x) - w_a A(x) \\ [H] \text{s.t. } \tau_{\text{max}}(x) &\geq \max_{t \in T} \tau_{\text{req}}(t), \quad \text{CoM}(x) \in \mathcal{S}_{\text{support}}, \\ E_{\text{batt}}(x) &\geq P_{\text{avg}}(x) T_{\text{mission}}, \quad T_{\text{motor}}(x) \leq T_{\text{rated}}. \end{aligned} \quad (399)$$

ここで x は構成要素選択を符号化し、 $A(x)$ は敏捷性を測定し、重み w_i はプログラム優先度を表す。

実装：ツールとアルゴリズム

1. モジュラー機械サブシステム

- 標準的な電気・機械インタフェースを持つ交換可能な四肢を用いる。これにより開発時間を短縮し、並行的トレードスタディを支援する。
- 安全な相互作用とエネルギー貯蔵が重要な場所では可変剛性アクチュエータ (VSA) を適用し、遠位関節では衝撃耐性のため series-elastic actuators (SEA) を用いる。

2. 統合センシングと知覚

- ボディ IMU、分散力・トルクセンシング、高密度ビジョンを融合し、接触力と支持ポリゴンをリアルタイムで推定する。
- カルマンフィルタまたは因子グラフ推定器を用いて全身状態を計算し、モデル予測コントローラに供給する。

3. 全身制御と安全層

- 階層制御を実装する：上位プランナがタスクを出力し、モデル予測全身コントローラがダイナミクスと接触制約を強制し、下位トルクリープがコンプライアンスを保証する。
- 状態推定が高い転倒リスクを示唆する際に介入する検証済み安全モニタを追加する。

4. シミュレーション駆動設計

- 機構・制御・知覚の反復共設計のための高忠実度シミュレータを用いる。ループ内でコントローラを検証し、sim-to-real ギャップを削減する。

コード：候補モーターおよび質量構成に対するバッテリーサイジングを評価する最小限の Python 例。ミッション電力と搭載質量制約を考慮してバッテリー質量を `scipy.optimize` で選択する。これはシステムレベルトレードスタディの実用的出発点である。

コードサンプル 167 Simple battery-sizing optimization

```
import numpy as np
from scipy.optimize import minimize
from typing import Tuple, Dict, Any
import logging

# ログ設定（本番環境では外部設定ファイルから読み込む）
logging.basicConfig(level=logging.INFO, format='%(asctime)s-%(levelname)s-%(message)s')
logger = logging.getLogger(__name__)

class BatterySizingOptimizer:
    """ バッテリー質量最適化クラス：ミッション要求を満たしながら総質量を最小化 """

    def __init__(self,
                 avg_power_w: float = 150.0,
                 mission_time_s: float = 7200.0,
                 energy_density_wh_per_kg: float = 180.0,
                 base_struct_mass_kg: float = 20.0,
                 struct_mass_factor: float = 0.3,
```

```

        payload_mass_kg: float = 5.0,
        batt_mass_bounds_kg: Tuple[float, float] = (1.0, 200.0)) -> None:
    """
    パラメータはデフォルト値を持たせつつ外部注入可能にして単体テストしやすくする
    """
    self.P_avg = avg_power_w
    self.T_mission = mission_time_s
    self.e_batt_J = energy_density_wh_per_kg * 3600.0
    self.m_struct_base = base_struct_mass_kg
    self.m_struct_factor = struct_mass_factor
    self.m_payload = payload_mass_kg
    self.batt_bounds = batt_mass_bounds_kg

    # 要求エネルギー事前計算
    self.E_req = self.P_avg * self.T_mission

    # 最適化結果キャッシュ
    self._result: Dict[str, Any] = {}

def total_mass(self, m_batt: float) -> float:
    """構造質量をバッテリー質量に比例させて推定"""
    m_struct = self.m_struct_base + self.m_struct_factor * m_batt
    return m_batt + m_struct + self.m_payload

def energy_constraint(self, x: np.ndarray) -> float:
    """エネルギー制約：使用可能エネルギー - 要求エネルギー >= 0"""
    m_batt = x[0]
    E_avail = m_batt * self.e_batt_J
    return E_avail - self.E_req

def objective(self, x: np.ndarray) -> float:
    """目的関数：総質量最小化"""
    return self.total_mass(x[0])

def optimize(self) -> Dict[str, Any]:
    """最適化実行：失敗時は例外を送出して呼び出し側でハンドリング"""
    x0 = np.array([(self.batt_bounds[0] + self.batt_bounds[1]) / 2])
    bounds = [self.batt_bounds]
    cons = ({'type': 'ineq', 'fun': self.energy_constraint})

```

```

try:
    res = minimize(
        self.objective,
        x0,
        bounds=bounds,
        constraints=cons,
        method='SLSQP',
        options={'ftol': 1e-6, 'disp': False}
    )
    if not res.success:
        raise RuntimeError(f"Optimization failed: {res.message}")
except Exception as e:
    logger.error(f"Optimization error: {e}")
    raise

m_batt_opt = float(res.x[0])
m_total_opt = self.total_mass(m_batt_opt)

self._result = {
    'battery_mass_kg': m_batt_opt,
    'total_mass_kg': m_total_opt,
    'struct_mass_kg': self.m_struct_base + self.m_struct_factor * m_batt_opt,
    'energy_margin_J': self.energy_constraint(res.x),
    'success': True
}
logger.info(f"Optimization complete: {self._result}")
return self._result

def main() -> None:
    """CLI エントリーポイント"""
    optimizer = BatterySizingOptimizer()
    result = optimizer.optimize()
    print(f"Optimal battery mass: {result['battery_mass_kg']:.2f} kg")
    print(f"Total system mass: {result['total_mass_kg']:.2f} kg")

if __name__ == "__main__":
    main()

```

設計トレードオフと運用リスク

- トレードオフ:

- ギア比を小さくするとコンプライアンスが向上するが、モータ電流と発熱が増加する。
- 重いバッテリーは航続距離を延ばすが、搭載質量を減らし慣性負荷を増加させる。
- 高性能知覚は計算電力消費と熱設計の複雑さを増大させる。
- リスク：
 - 転倒はセンサとアクチュエータを損傷させる；冗長性とコンプライアント設計がこれを緩和する。
 - 工業サイトでの電磁干渉は状態推定を破損させる；ハードニングとフィルタリングが必要である。
 - 人間環境での認証は文書化された安全事例と故障モード解析を要求する。

技術的影響

- 迅速なイテレーションと現場アップグレードを可能にするためにモジュラ性を最優先する。式(3)の最適化フレームワークを用いて早期にトレードオフを定量化する。電力・熱・ダイナミクス間の結合効果を明らかにするために全システムシミュレーションを重視する。最後に、保守性と形式的安全性証拠を計画し、協働または極限環境での運用展開を確実にする。

ケーススタディ：完全なヒューマノイドロボットの構築

51 ゼロから始める

51.1 要求と目標の定義

先に導入したサブシステムのトレードオフと性能ヒューリスティクスを基に、新規ヒューマノイドプラットフォームの機械・電気・ソフトウェア設計選択を導く、測定可能な要求とミッション目標を本節で形式化する。

設計問題を定義する：ハードウェア選定前にミッションシナリオ、制約、受け入れ試験を明確にする。明確な問題文は手戻りを減らし、コンポーネント選択が運用ニーズを満たすことを保証する。本活動の主要な出力は以下である：

1. 優先順位付けされたミッションシナリオのリスト（産業メンテナンス、倉庫マニピュレーション、人間支援）。
2. シナリオごとの定量化された性能指標（歩行速度、ペイロード、マニピュレーション精度、稼働時間）。
3. ハード制約（最大質量、フォームファクタ、安全クリアランス、規制上限）。
4. シミュレーションおよびハードウェアで検証可能な受け入れ試験。

ミッション目標を測定可能な指標を用いて工学要求に変換する。以下のカテゴリと代表的パラメータを用いる：

- 歩行：持続平地速度 v_{walk} (m/s)、階段能力、段高、静安定余裕 SM。
- マニピュレーション：エンドエフェクタペイロード W_{ee} (kg)、繰返し精度 ϵ (mm)、把持力 F_g (N)。
- エネルギーと耐久：ミッション継続時間 t_{mission} (s)、平均消費電力 P_{avg} (W)、システム効率 η_{sys} 。

- ・計算と知覚：センサ有効距離 R (m)、自己位置推定精度 σ_{pose} (m)、制御ループ周波数 f_{ctrl} (Hz)。
- ・安全とコンプライアンス：最大衝撃力、安全トルク上限、フェイルセーフ応答時間 t_{fail} 。

実現可能性計算を初期段階で実施し、非現実的な目標をフィルタリングする。バッテリーサイズは一般的なドライバである。必要バッテリーエネルギーを

$$[H]E_{\text{req}} = \frac{P_{\text{avg}} \cdot t_{\text{mission}}}{\eta_{\text{sys}}} \quad (400)$$

と推定し、次にバッテリー質量を

$$[H]m_{\text{batt}} = \frac{E_{\text{req}}}{e_{\text{density}}} \quad (401)$$

と近似する。ここで e_{density} は利用可能な質量エネルギー密度 (Wh kg^{-1}) である。これらの見積もりはアクチュエータサイジングと構造質量予算へフィードバックされる。

静的+動的モデルを用いて関節トルク要求を保守的に推定する。重心距離 l_{cm} 、質量 m_{link} 、角加速度 α を持つ単一回転関節について、必要トルクを

$$[H]\tau_{\text{req}} \approx I_{\text{joint}}\alpha + m_{\text{link}}gl_{\text{cm}} \cos \theta \quad (402)$$

と近似する。安全率 k_{sf} (通常 1.5-2.0) を含め、選定モータトルクが $\tau_{\text{motor}} \geq k_{\text{sf}}\tau_{\text{req}}$ を満たすようにする。

競合する要求をトレーサビリティ行列でランク付けする。単純な重み付きスコアで優先度を正規化する：

$$[H]S = \frac{\sum_i w_i s_i}{\sum_i w_i} \quad (403)$$

ここで s_i は要求ごとの正規化充足スコア、 w_i は重みである。この指標により、バッテリー質量とペイロードやセンサ精度を数学的にトレードオフできる。

要求の取得・検証・反復のための実装指針：

1. 各要求を原子形で記述：単一の測定可能パラメータ、閾値、試験方法、合格／不合格基準。
2. 各要求を1つ以上のサブシステムにマッピングし、オーナーシップを割り当てる。
3. Isaac Sim で受け入れ試験自動化のためのシミュレーションテストを作成する。忠実度の降順で検証：ユニットモデル、統合デジタルツイン、次にハードウェアインザループ。
4. ユニット予算（質量・電力・体積）を維持し、上位制約に総和して違反時に警告を発する。

小型 Python ユーティリティでエネルギー・トルクマージンの初期実現性チェックを実装する。コンセプト設計中にこのスクリプトを用いてパラメータ選択を高速に反復する。

コードサンプル 168 バッテリー質量とトルクマージンの実現性チェック

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import math
from typing import Final
```

```

# 定数定義
P_AVG: Final[float] = 250.0          # W: 平均消費電力
T_MISSION: Final[float] = 3600.0     # s: ミッション時間
ETA_SYS: Final[float] = 0.85         # システム効率
E_DENSITY: Final[float] = 200.0     # Wh/kg: 使用可能エネルギー密度

I_LINK: Final[float] = 0.02          # kg・m2: リンク慣性モーメント
ALPHA: Final[float] = 2.0            # rad/s2: 要求加速度
M_LINK: Final[float] = 3.0           # kg: リンク質量
L_CM: Final[float] = 0.12            # m: 重心長さ
G: Final[float] = 9.81               # m/s2: 重力加速度
SAFETY_FACTOR: Final[float] = 2.0    # 安全率
MOTOR_TAU: Final[float] = 50.0       # Nm: モータ最大トルク

def compute_required_energy(p_avg: float, t_mission: float, eta_sys: float) -> float:
    """必要なバッテリーエネルギー [Wh] を算出"""
    return p_avg * (t_mission / 3600.0) / eta_sys

def compute_battery_mass(e_req_wh: float, e_density: float) -> float:
    """バッテリー質量 [kg] を算出"""
    if e_density <= 0:
        raise ValueError("e_density must be positive")
    return e_req_wh / e_density

def compute_required_torque(
    inertia: float, alpha: float, mass: float, length: float, theta: float = 0.0
) -> float:
    """必要な関節トルク [Nm] を算出（重力補償込み）"""
    return inertia * alpha + mass * G * length * math.cos(theta)

def main() -> None:
    # エネルギー・バッテリー質量算出
    e_req_wh = compute_required_energy(P_AVG, T_MISSION, ETA_SYS)
    battery_mass = compute_battery_mass(e_req_wh, E_DENSITY)

    # トルクチェック

```

```

tau_req = compute_required_torque(l_LINK, ALPHA, M_LINK, L_CM)
tau_with_sf = tau_req * SAFETY_FACTOR
torque_ok = MOTOR_TAU >= tau_with_sf

# 結果表示
print(f"E_req_Wh={e_req_wh:.1 f} Wh, battery_mass={battery_mass:.2 f} kg")
print(f"tau_req={tau_req:.2 f} Nm, required_with_sf={tau_with_sf:.2 f} Nm")
print(f"torque_OK={torque_ok}")

if __name__ == "__main__":
    main()

```

要求を運用化するための以下の工学プラクティス：

- 初期は保守的マージンを用い、シミュレーション・ハードウェアデータが揃ったらマージンを絞る。
- 安全とミッション成功に直接影響する要求を優先する。
- 要求を試験可能に保つ。検証のための正確なデータセット、軌道、環境条件を指定する。

主要な設計トレードオフと運用上のリスク：

- バッテリ質量対ペイロードおよび駆動電力は結合した設計ループを形成し、しばしばシステム質量を支配する。
- 高トルクアクチュエータは質量と熱放射を増加させ、ギアボックス選択は逆駆動性と安全に影響する。
- センサ精度は計算要求を高め、電力・熱設計に影響する。
- 安全マージン不足は早期故障を招き、過剰マージンは過重量・低性能設計を生む。

これらの定量化された要求は、システム工学、機械設計、電気設計、制御の間の契約となる。トレサビリティ、初期実現性チェック、自動受け入れ試験は、統合リスクを削減し、反復を加速する。

51.2 設計および製造プロセスの計画

先に確立された目標は範囲、性能目標、および受け入れ試験を決定する；本小節ではそれらの目標を統合リスクを最小化しコストとスケジュールを制御する具体的な設計・製造計画に変換する。

簡潔な問題文から始める：胴体と四肢を備えた 20 自由度駆動のヒューマノイドを搭載知覚機能と共に提供し、2 時間の公称運用と歩行、物体操作、安全な人間相互作用の実証を行う。計画はその文をマイルストーン、成果物、検証ステップ、および対策計画に変換する。

技術的分析と分解

- ロボットを並行作業と明確な責任のためのモジュール化されたサブシステムに分解する：
 1. 機械構造（シャーシ、四肢リンク、ジョイントハウジング）。
 2. 駆動および電源（モータ、ドライブ、バッテリー）。

3. センシングおよび知覚（IMU、ステレオカメラ、力センサ）。
 4. リアルタイム制御および安全レイヤ（低レベルコントローラ、非常停止）。
 5. 高レベルソフトウェア（モーションプランナ、ビヘイビアツリー）。
- 各モジュールに対して以下を作成する：
 - サブシステム要求を全体受け入れ試験にマッピングする要求トレーサビリティマトリックス。
 - CAD から試作、生産への成熟度ロードマップ（TRL ライクなレベル）。

定量的計画指標

- インターフェース数で統合複雑度を見積もる。 n 個のサブシステムに対して最悪ケースのペアワイズ統合試験は

$$[H]N_{\text{tests}} = \frac{n(n-1)}{2}, \quad (404)$$

としてスケールし、これはモジュラバスと明確な API 契約による結合削減を促す。

- 直列サブシステムの信頼性を定義する。サブシステムが独立に故障する場合、システム信頼性は構成要素信頼性の積である：

$$[H]R_{\text{sys}}(t) = \prod_{i=1}^n R_i(t). \quad (405)$$

これは重要サブシステムのわずかな改善が均一な変更よりも大きなシステムレベル利益をもたらすことを示す。

スケジューリングとマイルストーン

- ハード検証ゲートを伴う反復マイルストーンを用いる：
 1. コンセプト検証（週 0–4）：CAD レイアウト、質量および CG 見積もり、モータ選定。
 2. アクチュエータおよび電源サブシステムの試作（週 4–12）：トルク、熱挙動の試験。
 3. 知覚およびシミュレーション統合（週 8–16）：Isaac Sim での合成センサデータ。
 4. 最初のハードウェアインザループ（HIL）試験（週 12–20）：力板による単脚バランス。
 5. 完全統合および安全認証（週 20–36）：監視付き歩行および操作試験。
- クリティカルパス法を適用し最小スケジュールを支配するタスクを特定する。非クリティカルタスクのフロートを追跡し、必要に応じてリソースを再割当てしてパスを圧縮する。

リスク管理およびテストファースト戦略

- リスクを発生確率と結果で分類する。典型的なヒューマノイド高リスク項目：
 - 連続稼働下でのアクチュエータ過熱。
 - 電源配電故障および無制御シャットダウン。
 - 実環境での知覚ドリフト。
 - 遷移中の制御不安定。
- 緩和策：
 - 定格トルクでのモータ早期熱試験。
 - 姿勢推定のための冗長センシング（IMU + ビジュアルオドメトリ）。
 - Isaac Sim におけるドメインランダムマイゼーションを用いたシミュレーションストレステ

スト。

- フルスケール試験前の HIL 試験。
- インターフェースに対してテストファーストアプローチを採用：部品製造前に単体試験、統合試験、受け入れ試験を定義する。

実装ガイダンスおよびサプライヤ戦略

- カスタム対市販品のトレードオフを決定する：
 - トルク密度および熱試験データが存在する市販トルクモータおよびドライブを使用する。
 - 質量分布またはコンプライアンス要求が特異な場合、カスタムハウジングおよびエンドエフェクタは価値がある。
- リードタイムを含む優先付け済み部品表（BOM）を作成する。長納期品を早期にロックしてスケジュール遅延を回避する。
- 機械設計に予備部品および保守アクセスを計画する。

検証インフラ

- シミュレーションアセットを早期に構築する。仮想ツイン忠実度は以下を含む必要がある：
 - 正確な慣性プロパティ。
 - アクチュエータ帯域および飽和。
 - センサノイズおよび遅延モデル。
- シナリオレベル試験のためにビヘイビアツリー（GROOT）をシミュレーションに統合する。
- フルロボット組立前の制御ループ検証のため HIL リグをスケジュールする。

実践的制御および安全考慮事項

- レート別に制御ループを分割する：
 - トルク／位置制御用の高速インナーループ（1 kHz）。
 - 全身制御およびバランス用の中レベルループ（50–200 Hz）。
 - タスクシーケンシング用の高レベルプランニング（1–10 Hz）。
- ジョイント範囲、速度、および許容重心圧逸脱を制限するソフトウェアによる安全エンベロープ制約を定義する。
- グレースフルデグラデーション戦略を検証するための故障注入試験を試作する。

ツーリングおよび再現性

- バージョン管理された CAD およびソフトウェアリポジトリを使用する。ハードウェアリビジョンおよびシミュレーションモデル用にリリースをタグ付けする。
- ファームウェアおよび知覚スタックで可能な限り CI パイプラインを用いてビルドおよび試験を自動化する。
- 生きたリスクレジスタを維持し、各マイルストーンレビューで更新する。

小さな実用的スクリプトがスケジュールおよび信頼性計算を支援する；プロジェクト追跡ツールに適応させる。

コードサンプル 169 単純なシステム信頼性およびペアワイズ試験を計算

```
"""
Production-ready reliability & integration estimator
ROS2 ノードとしても動作可能 (rclpy 対応)
"""

from __future__ import annotations

import math
from typing import Dict, Iterable, List, Optional, Tuple

import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32, Int32

class ReliabilityCalculator(Node):
    """システム信頼度と統合試験数を計算する ROS2 ノード"""

    def __init__(self) -> None:
        super().__init__("reliability_calculator")
        self._declare_parameters()
        self._setup_publishers()

    # -----
    # パラメータ管理
    # -----
    def _declare_parameters(self) -> None:
        self.declare_parameter("modules", rclpy.Parameter.Type.STRING_ARRAY)
        self.declare_parameter("reliabilities", rclpy.Parameter.Type.DOUBLE_ARRAY)

    # -----
    # ROS 2 通信
    # -----
    def _setup_publishers(self) -> None:
        self.sys_rel_pub = self.create_publisher(Float32, "~/system_reliability", 10)
        self.integ_cnt_pub = self.create_publisher(Int32, "~/integration_count", 10)
        self.timer = self.create_timer(1.0, self._publish_results)
```

```

# -----
# 計算ロジック
# -----
@staticmethod
def system_reliability(reliabilities: Iterable[float]) -> float:
    """直列構成のシステム信頼度を返す（各要素の積）"""
    r = 1.0
    for ri in reliabilities:
        if not (0.0 <= ri <= 1.0):
            raise ValueError("信頼度は0～1の範囲である必要があります")
        r *= ri
    return r

@staticmethod
def pairwise_tests(n: int) -> int:
    """完全グラフの辺数=ペアワイズ統合試験数"""
    if n < 0:
        raise ValueError("モジュール数は非負でなければならない")
    return n * (n - 1) // 2

# -----
# ROS 2 出力
# -----
def _publish_results(self) -> None:
    modules = (
        self.get_parameter("modules").get_parameter_value().string_array_value
    )
    reliabilities = (
        self.get_parameter("reliabilities").get_parameter_value().double_array_val
    )

    if len(modules) != len(reliabilities):
        self.get_logger().error("モジュール数と信頼度数が一致しない")
        return

    sys_rel = self.system_reliability(reliabilities)
    integ_cnt = self.pairwise_tests(len(modules))

    self.sys_rel_pub.publish(Float32(data=sys_rel))
    self.integ_cnt_pub.publish(Int32(data=integ_cnt))

```

```

# -----
# スタンドアロン実行
# -----
def main(args: Optional[List[str]] = None) -> None:
    rclpy.init(args=args)
    node = ReliabilityCalculator()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == "__main__":
    # ROS 2未使用時の簡易実行
    try:
        rclpy
    except NameError:
        # ROS 2環境なし → 直接計算
        modules: Dict[str, float] = {
            "mechanical": 0.99,
            "actuation": 0.98,
            "sensing": 0.97,
            "control": 0.995,
        }
        calc = ReliabilityCalculator
        print(calc.system_reliability(modules.values()))
        print(calc.pairwise_tests(len(modules)))
    else:
        main()

```

運用上の影響、設計トレードオフ、およびリスク

- モジュラアーキテクチャは統合作業を削減するが質量およびコネクタを増加させる。モジュラ性と質量予算をバランスさせる。
- シミュレーションおよび HIL への投資は現地故障を減らす初期コストおよびスケジュールを増加させる。その投資は後期再設計リスクを劇的に低下させる。

- ・信頼性解析は直列サブシステムがシステム寿命を短縮することを示す；アクチュエータ、電源、リアルタイム制御の堅牢性を優先させる。
- ・並行開発によるスケジュール圧縮はインターフェースリスクを増加させる；厳格な API 契約および自動試験を適用して統合故障を緩和する。

51.3 コストとタイムラインの管理

要求事項と設計計画を踏まえたうえで、コストとスケジュールの制約が実現可能性と達成可能な機能セットを決定する。以下では、人型ロボットの構築に対して予算を組み、スケジュールを立て、リスクを制御する実践的な手法を、エンジニアリング指標とコンパクトなシミュレーション例を用いて展開する。

問題定義. 予算 B と締切 D 以内に機能要求を満たす人型プロトタイプを納入せよ。一般的な制約：

- ・長納期の機械部品（アクチュエータ、ハーモニックドライブ）およびカスタム PCB がカレンダーリスクを支配する。
- ・ソフトウェアおよび統合タスクはしばしば高いばらつきと隠れた依存関係を示す。
- ・並行作業、人員コスト、手戻りリスクの間でトレードオフが生じる。

技術分析. 問題を 3 つの結合した層に構造化する。

1. 作業分解と先行関係. タスク T_i を期間 d_i 、コスト c_i 、リソース要求 r_i で分解する。有向非巡回グラフ（DAG）で先行関係を捉える。
2. 不確実性下でのスケジュール見積もり. PERT に従い各タスクに三点見積（楽観 a 、最頻 m 、悲観 b ）を用いて期待期間を計算。タスクに対して：

$$[H]E[d_i] = \frac{a_i + 4m_i + b_i}{6}, \quad \sigma_i = \frac{b_i - a_i}{6}. \quad (406)$$

パスの総期間は期待値の和；分散は独立性を仮定して加算。

3. コストー時間トレードオフ（クラッシング）. 期間短縮可能なタスクに対して、通常モードとクラッシュモードを定義する。クラッシュ傾斜は限界クラッシュコストを定量化：

$$[H]s_i = \frac{c_{i,crash} - c_{i,normal}}{d_{i,normal} - d_{i,crash}}. \quad (407)$$

期限 D またはコスト上限 B に達するまで、クリティカルパスタスクのうち傾斜が最小のものを単位時間節約ごとに選択する。

実装指針とヒューリスティクス.

- ・詳細な統合タスク開始前に長納期品の調達を優先する。独特なギアボックスの 12 週納期が 6 か月プロジェクトを停滞させることがある。
- ・可能な限りクリティカルコンポーネントをデュアルソース化する。増加分コストはスケジュールリスクより小さいことが多い。
- ・リソースレベリングを適用し、時間当たりコストやエラー率を増加させる人員ピークを回避する。
- ・ハードウェアとソフトウェアに分けて予備費を配分：ハードウェア予備費は BOM の 15-25；ソフトウェア予備費は計画工数の 10-20。

- ・カスタム製作に対してマイルストーンベースのベンダー契約を用い、支払いを納入ステップに合わせ現金消耗リスクを削減する。

定量的予算編成とリスク. プロジェクト総コストを：

$$[H]C_{\text{total}} = C_{\text{BOM}} + C_{\text{labor}} + C_{\text{tooling}} + C_{\text{contingency}} + C_{\text{risk}}, \quad (408)$$

と表現し、 C_{risk} は手戻りや遅延ペナルティの期待値。スケジュール目標最適化では、重み付き目的関数を最小化：

$$[H]\min_x C_{\text{total}}(x) + \lambda E[T(x)], \quad (409)$$

x は配分決定、 λ は時間コスト（バーン率、ペナルティ）。 λ を選ぶことで請負人雇用とスケジュール遅延容認のバーン率トレードオフが明らかになる。

実践的シミュレーション：スケジュールおよびコスト分布のためのモンテカルロ PERT。以下のスニペットはタスク期間をサンプリングしクリティカルパスを計算し期限 D 達成確率を推定する。タスクパラメータを実際のベンダーリードタイムおよび工数見積もりに置き換えること。

コードサンプル 170 人型ロボット構築のためのモンテカルロスケジュール・コスト推定器.

```
#!/usr/bin/env python3
import random
import math
from typing import Dict, List, Tuple, Optional

# タスク定義：(名前, 楽観a, 最頻m, 悲観b, 通常費用, 短縮費用, 短縮可能週数)
Task = Tuple[str, float, float, float, float, Optional[float], int]

TASKS: List[Task] = [
    ('Frame', 5, 7, 10, 8000, None, 0),
    ('Actuators', 8, 12, 16, 50000, 65000, 0),
    ('PCB', 2, 4, 6, 4000, 7000, 0),
    ('ControlSW', 6, 10, 14, 60000, None, 0),
    ('Integration', 4, 6, 10, 20000, None, 0),
]

# 先行関係 (隣接リスト)
PRECEDENCE: Dict[str, List[str]] = {
    'Frame': [],
    'Actuators': ['Frame'],
    'PCB': ['Frame'],
    'ControlSW': ['Actuators', 'PCB'],
    'Integration': ['ControlSW'],
}
```

```

# タスク名→インデックスの辞書
NAME2IDX: Dict[str, int] = {t[0]: i for i, t in enumerate(TASKS)}

def sample_duration(a: float, m: float, b: float) -> float:
    """PERT近似でタスク期間をサンプリング"""
    mean = (a + 4 * m + b) / 6.0
    sigma = (b - a) / 6.0
    return max(0.1, random.gauss(mean, sigma))

def critical_path_duration(sampled: Dict[str, float]) -> float:
    """動的計画法で最長パス(クリティカルパス)を計算"""
    longest: Dict[str, float] = {}
    for name, _, _, _, _, _ in TASKS:
        start = max((longest[p] for p in PRECEDENCE[name]), default=0.0)
        longest[name] = start + sampled[name]
    return max(longest.values())

def monte_carlo(
    deadline_weeks: int = 40,
    n_samples: int = 5000,
    penalty_rate: float = 0.10,
) -> Tuple[float, float]:
    """モンテカルロシミュレーションで納期遵守率と期待コストを推定"""
    base_cost = sum(t[4] for t in TASKS)
    on_time = 0
    total_cost = 0.0

    for _ in range(n_samples):
        sampled = {t[0]: sample_duration(t[1], t[2], t[3]) for t in TASKS}
        dur = critical_path_duration(sampled)
        penalty = base_cost * penalty_rate * max(0.0, (dur - deadline_weeks) / deadline_weeks)
        total_cost += base_cost + penalty
        if dur <= deadline_weeks:
            on_time += 1

    return on_time / n_samples, total_cost / n_samples

if __name__ == '__main__':
    random.seed(42)

```

```
p_on_time, mean_cost = monte_carlo()
print(f"P(on_time)={p_on_time:.3f}  mean_cost={mean_cost:.0f}")
```

運用管理とガバナンス.

- 主要なソフトウェアプリント開始前に BOM をロックし機械インタフェースを凍結する。インタフェースのずれは高コストな改造を引き起こす。
- 客観的受け入れ試験付きゲート付きマイルストーンを用いる、例：脚統合前にアクチュエータトルクおよび CAN バステスト。
- 四つのライブ指標を週次で追跡：アーンドバリュー、実績コスト、クリティカルパス残期間、長納期発注状況。

エンジニアリングへの影響、トレードオフ、リスク.

- 攻撃的なスケジュール短縮はクリティカルパスハードウェアのクラッシングまたは高コスト請負人雇用を要し、式 (407) に従い C_{total} を増大させる。
- 過度のカスタマイズは BOM リードタイム分散と C_{risk} を増やす。可能な限りモジュール型既製モジュールを優先する。
- 過剰な並行性は統合欠陥と手戻り確率を高め、見かけの並行利得にもかかわらず期待 $E[T]$ を増加させる。
- シングルソーシングは即時調達複雑さを減らす、単一障害点スケジュールリスクを導入する。
- 財務ランウェイとマイルストーン連動資金調達は式 (409) で可能な λ を決定し、機能スコープに直接影響する。

設計トレードオフ：予備費をリードタイム分散と統合リスクに基づきハードウェアとソフトウェア間に配分する。運用リスク軽減は、ベンダー実績とバーン率監視の明示的指標を用いて、追加コストとスケジュール確実性の間でバランスを取らなければならない。

52 段階的な実装

52.1 機械構造の設計

機械構造の設計は、先に決定された要求事項、コスト、スケジュールに従い、それらの目標を具体的な運動学的、慣性的、材料の制約に変換する。本小節では、タスクレベルの仕様を、生産可能なヒューマノイドのための関節サイジング、トポロジ選択、検証ステップにどう変換するかを示す。

問題定義と工学目的。構造は以下を満たす必要がある：

- エンドエフェクタおよび胴体で目標ペイロードを支えながら質量を最小化；
- タスクに必要な自由度 (DoF) と可動域を提供；
- 安定した移動および操作に重心 (CoM) 配置を実現；
- 実世界展開に向けた冷却、配線、保守性を許容；
- 人間との相互作用のための安全・認証制約を満たす。

技術解析：トルク、慣性、帯域。関節アクチュエータの選定は、準静的負荷と動的挙動の両方から

導かれる必要トルクおよび速度見積りから始まる。回転関節において、遠端リンク質量 m 、遠端リンク重心距離 l 、重力加速度 g 、所望角加速度 α が与えられたとき、必要アクチュエータトルクは

$$[H]\tau_{\text{req}} \approx I_{\text{eq}}\alpha + mgl \cos \theta, \quad (410)$$

で上限を評価できる。ここで I_{eq} は関節に見かけた等価慣性、 θ は関節姿勢である。多リンクチェーンでは、リンク慣性を標準の複合剛体方程式で伝播するか、再帰的 Newton-Euler 法を用いて指定運動軌道下での関節トルクを計算する。

設計の経験則：

1. 連続トルク τ_c が $\tau_c \geq \text{FS} \cdot \tau_{\text{cont}}$ を満たすアクチュエータを選ぶ。FS は安全率（通常 1.5-3）。
2. モータのピークトルクと熱時定数が、過熱なく断続的高パワー動作を維持できることを確認。
3. ギア減速比 N を選び、モータ速度をトルクに変換： $\tau_{\text{motor}} \cdot N \geq \tau_{\text{req}}$ としながら、反映慣性 $J_r = J_m N^2$ を制御帯域に許容範囲に保つ。

モード考慮とコンプライアンス。閉ループ安定性と外乱抑制のため、各関節または肢セクションの機械的固有振動数 ω_n は意図する制御帯域の 5-10 倍を超えるべきである。回転バネ剛性 k と集中慣性 J に対し、固有振動数は

$$[H]\omega_n = \sqrt{\frac{k}{J}}. \quad (411)$$

series elastic actuators (SEA) を用いる場合、トルクセンシング精度と低 ω_n ・遅い応答のバランスを考えて k を設計する。SEA は衝撃許容性と力制御性能を高めるが、高周波帯域は低下する。

トポロジと材料のトレードオフ。構造トポロジを質量、剛性、製造性で評価：

- ・モノコック胴体は軽量でクリーンなケーブルルーティングを得るが修理が複雑。
- ・エクソスケルトンフレームは組立とモジュール交換を簡素化。
- ・内部バックボーンとモジュール肢ポッドは、サービス性と剛性のバランスを取る。

材料選択：

- ・アルミニウム合金：剛性-コスト比が良く、加工が容易。
- ・炭素繊維複合材：剛性-質量比が最良だが、金型費用が高い。
- ・高強度鋼：衝撃耐性が必要な局所補強に。
- ・3D プリントポリマー：高速イテレーション、長期疲労寿命は限られる。

実装ワークフロー。以下の反復パイプラインを踏む：

1. CAD で肢ジオメトリと質量分布をパラメータ化。
2. 代表軌道から静トルクおよびピーク動トルクを計算。
3. 熱・慣性制約を考慮しアクチュエータと伝達機構を選定。
4. クリティカル負荷ケース（立位、転倒、重い操作）で FEA を実行。
5. ラピッドファブリケート部品で試作し、実慣性・たわみを測定。
6. トルクマージン、固有振動数、安全率が仕様を満たすまで反復。

実用的計算例。以下のスニペットは、肩振り動作を実行する 2 セグメントアームの必要ピークトルクを計算する。CAD エクスポートの公称パラメータを置き換える。スクリプトはモータ連続トルク

から減速比も見積る。

コードサンプル 171 肩振りのための関節トルクおよびギアボックス比を計算

```
#!/usr/bin/env python3
import math
from typing import Tuple

# リンクパラメータ (CAD 値)
m1, m2 = 2.5, 1.8          # kg
l1, l2 = 0.24, 0.18        # m
I1, I2 = 0.012, 0.006      # kg·m2

# 所望動作
ALPHA_PEAK = 6.0            # rad/s2
THETA_HORZ = 0.0           # rad (水平姿勢 = 重力トルク最大)

# 定数
g = 9.81                   # m/s2
SAFETY = 2.0               # 安全率
MOTOR_TAU_CONT = 0.8       # Nm (モータ連続定格トルク)

def calc_required_torque(m1: float, m2: float,
                          l1: float, l2: float,
                          I1: float, I2: float,
                          alpha: float, theta: float) -> Tuple[float, float]:
    """
    肩関節に必要な最大トルクを計算
    戻り値: (tau_grav, tau_dyn)
    """
    # 等価慣性モーメント (平行軸定理 + リンク2の遠端近似)
    I_eq = I1 + m1 * (l1 ** 2) + I2 + m2 * ((l1 + l2 / 2) ** 2)

    # 重力トルク (水平姿勢で最大)
    tau_grav = (m1 * g * l1 + m2 * g * (l1 + l2 / 2)) * math.cos(theta)

    # 動的トルク (慣性)
    tau_dyn = I_eq * alpha

    return tau_grav, tau_dyn
```

```

def recommend_gearbox(tau_total: float, motor_cont: float, safety: float) -> int:
    """
    必要トルクとモータ定格から最小ギア比を整数で返す
    """
    ratio = (safety * tau_total) / motor_cont
    return math.ceil(ratio)

def main() -> None:
    tau_grav, tau_dyn = calc_required_torque(
        m1, m2, l1, l2, l1, l2, ALPHA_PEAK, THETA_HORZ)
    tau_req = tau_grav + tau_dyn

    gear_ratio = recommend_gearbox(tau_req, MOTOR_TAU_CONT, SAFETY)

    print(f"必要トルク: {tau_req:.2f} Nm")
    print(f"推奨ギアボックス比: {gear_ratio}:1")

if __name__ == "__main__":
    main()

```

統合・保守のための設計。ケーブルハーネス経路、センサ配置、熱放熱孔を早めに計画。標準フランジインターフェースとモジュール電気コネクタを選び、高速交換を可能にする。高質量要素をCoM近くに配置し、肢トルクを低減する。

工学への影響、トレードオフ、リスク：

- ・質量削減はアクチュエータ消費エネルギーを下げるが、積極的な軽量化は座屈リスクを高める。
- ・高減速比はモータ電流を下げるが、反映慣性を増やし逆駆動性を低下させる。
- ・コンプライアンス追加は安全性を高めるが、より高度な制御・チューニングを要求する。
- ・熱限界は連続運用を制約；冷却策は筐体または放熱孔設計変更を強いる可能性がある。
- ・故障モードには応力集中部の疲労、コネクタ摩耗、センサ位置ずれがあり、点検間隔を計画的に設定する。

上記の式を用いてトレードオフを早期に定量化し、実機試作で反復する。設計判断は、意図する運用領域における性能、信頼性、保守性のバランスを取らなければならない。

52.2 制御ソフトウェアのプログラミング

機械設計は、関節可動域、アクチュエータ特性、センサ配置を定め、これらを制御ソフトウェアが尊重しなければならない。本小節では、これらの物理仕様を、フルヒューマノイドシステム向けの決

定的、安全、かつテスト可能な制御スタックに変換することに焦点を当てる。

問題定義. 高レベルタスクをアクチュエータ指令に変換する階層型制御ソフトウェアスタックを実装せよ。要件は以下の通り：

- トルクまたは位置ループ用に 500–1000 Hz のリアルタイム関節レベル制御。
- バランス、マニピュレーション、歩行のための全身協調。
- IMU、関節エンコーダ、外部ビジョンを融合する堅牢な状態推定。
- 関節可動域、トルク可動域、緊急停止を強制する安全モニタ。
- Isaac Sim での開発のためのシームレスなシミュレーションからハードウェアへの移行。

技術分析. 3 層アーキテクチャを用いる：

1. 低レベルコントローラ：トルク／位置／インピーダンスループを実行し、リアルタイム OS または MCU 上で動作。フィードフォワード動力学補償とフォールバック受動コンプライアンスモードを実装しなければならない。
2. ミドルレベル全身コントローラ（WBC）：逆動力学／運動学を解き、複数タスクの関節トルクを制約を満たしながら配分する。
3. ハイレベルタスクプランナとビヘイビアマネージャ：目標エンドエフェクタセットポイントと歩行パラメータを発行し、通常は GROOT 内のビヘイビアツリー経由。

主要な方程式は、モデルと指令の間の忠実性を保証する。逆動力学は、関節トルク τ と運動および外力を結びつける：

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J(q)^T F_{\text{ext}}, \quad (412)$$

ここで M は慣性行列、 C はコリオリ項、 g は重力、 J はタスクヤコビアン、 F_{ext} は外力ワレンチを表す。一般的な制御則は、逆動力学フィードフォワードと PD フィードバックを組み合わせる：

$$\tau = M(q)\ddot{q}_{\text{des}} + C(q, \dot{q})\dot{q}_{\text{des}} + g(q) - K_p(q - q_{\text{des}}) - K_d(\dot{q} - \dot{q}_{\text{des}}).$$

エンドエフェクタのカルテシアンインピーダンスでは、目標ワレンチは

$$F = K(x_{\text{des}} - x) + D(\dot{x}_{\text{des}} - \dot{x}),$$

となり、関節トルクは $\tau = J^T F + N^T \tau_{\text{null}}$ で、 N は二次目的のためのヌル空間へ射影する。

実装ガイドライン. 決定的タイミングと最小遅延を最優先に。代表的なサンプリングレート：

- 低レベルサーボ：500–1000 Hz。
- WBC：100–200 Hz。
- ハイレベルプランナ：10–50 Hz。

状態推定はすべてのセンササンプルにタイムスタンプを付与し、融合アルゴリズムを用いる：

- IMU + 運動学的脚オドメトリを相補または拡張カルマンフィルタでベース姿勢に。
- 直列弾性アクチュエータ向けのヒステリシス補償付き関節エンコーダキャリブレーション。
- ビジョンベースローカリゼーションを非同期補正で低レートで。

安全とフォールトハンドリング：

- ソフトおよびハード関節可動域をソフトウェアで強制。
- トルク変化率制限を実装し、指令スパイクを防止。
- モータ温度とバス電圧を監視し、自動セーフモード遷移。

実用的なコードパターン. 以下の ROS2 スタイル Python コントローラは、トルクレベルインピーダンスコントローラを示し、リアルタイム互換ループを実行し、センサを読み取り、動力学ライブラリを用いてトルクを計算し、安全チェックを適用する。

コードサンプル 172 トルクレベルインピーダンスコントローラ (ROS2 ライク疑似コード)

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
import numpy as np
from builtin_interfaces.msg import Time
from sensor_msgs.msg import JointState, Imu
from trajectory_msgs.msg import JointTrajectoryPoint
from std_msgs.msg import Float64MultiArray
# dynamics_libはM,C,g,J関数を提供; hardware_apiはアクチュエータ／センサを抽象化
from dynamics_lib import compute_M_C_g, compute_Jacobian
from hardware_api import read_joint_states, write_joint_torques, read_imu

NUM_JOINTS = 7 # 実機に合わせて変更

class ImpedanceController(Node):
    def __init__(self):
        super().__init__('impedance_controller')
        self.declare_parameters(
            namespace='',
            parameters=[
                ('kp', [50.0]*NUM_JOINTS),
                ('kd', [2.0]*NUM_JOINTS),
                ('max_torque', 40.0),
                ('control_freq', 500.0),
            ])
        self.Kp = np.array(self.get_parameter('kp').value, dtype=float)
        self.Kd = np.array(self.get_parameter('kd').value, dtype=float)
        self.max_torque = float(self.get_parameter('max_torque').value)
        period = 1.0 / float(self.get_parameter('control_freq').value)

        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
```

```

        durability=DurabilityPolicy.VOLATILE,
        depth=1)

self.joint_cmd_pub = self.create_publisher(
    Float64MultiArray, '/joint_torque_command', qos)
self.joint_sub = self.create_subscription(
    JointState, '/joint_states', self.joint_callback, qos)
self.imu_sub = self.create_subscription(
    Imu, '/imu/data', self.imu_callback, qos)
self.traj_sub = self.create_subscription(
    JointTrajectoryPoint, '/desired_trajectory', self.traj_callback, qos)

self.timer = self.create_timer(period, self.control_loop)

self.q = np.zeros(NUM_JOINTS)
self.q_dot = np.zeros(NUM_JOINTS)
self.imu = Imu()
self.q_des = np.zeros(NUM_JOINTS)
self.qd_des = np.zeros(NUM_JOINTS)
self.qdd_des = np.zeros(NUM_JOINTS)

def joint_callback(self, msg: JointState):
    self.q = np.array(msg.position[:NUM_JOINTS])
    self.q_dot = np.array(msg.velocity[:NUM_JOINTS])

def imu_callback(self, msg: Imu):
    self.imu = msg

def traj_callback(self, msg: JointTrajectoryPoint):
    self.q_des = np.array(msg.positions[:NUM_JOINTS])
    self.qd_des = np.array(msg.velocities[:NUM_JOINTS])
    self.qdd_des = np.array(msg.accelerations[:NUM_JOINTS])

def control_loop(self):
    if self.q.size != NUM_JOINTS or self.q_dot.size != NUM_JOINTS:
        return # 初期化待ち

    M, C, g = compute_M_C_g(self.q, self.q_dot)
    e = self.q - self.q_des
    ed = self.q_dot - self.qd_des

```

```

tau_ff = M @ self.qdd_des + C @ self.qd_des + g
tau_fb = - (self.Kp * e + self.Kd * ed)
tau = tau_ff + tau_fb
tau = np.clip(tau, -self.max_torque, self.max_torque)

cmd = Float64MultiArray()
cmd.data = tau.tolist()
self.joint_cmd_pub.publish(cmd)
write_joint_torques(tau)

```

```

def main(args=None):
    rclpy.init(args=args)
    node = ImpedanceController()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    rclpy.shutdown()

```

統合とテスト戦略：

- ハードウェア試験前に Isaac Sim でソフトウェアインザループ (SITL) を実装。
- 同定実験を用いて動力学パラメータを検証。
- 通信遅延やパケットロスを含む最悪ケース外乱テストをシミュレーションで実行。
- 初期テストではハードウェア安全ハーネスとテザーを使用。

設計トレードオフ：

- 高い制御レートは遅延を減らすが CPU とネットワーク負荷を増加。
- トルクレベル制御は自然なコンプライアンスを与える；位置制御は初期プロトタイプで単純で安全。
- 完全モデル逆動力学は追従性を向上させるが、正確な慣性パラメータを必要とする。
- SE(3) 状態推定器を用いるとベース姿勢精度が向上するが、複雑さが増す。

運用上のリスクと緩和：

- モデルミスマッチは不安定を誘発；ロバストゲインと適応要素を採用。
- センサ故障は安全でない動作をリスクとする；ウォッチドッグとグレースフルリンプモードを実装。
- 遅延スパイクはトルクオーバーシュートを生じ；リアルタイムスケジューリングを監視し、デッドライン対応スレッドを使用。

ヒューマノイド工学への影響：

- ・アクチュエータ能力とミッションタスクに合わせて制御モダリティ（トルク対位置）を選択。
- ・モデルベースコントローラを活用するため、精密なキャリブレーションと同定に投資。
- ・リアルタイム計算と決定的ネットワークをシステム設計の初期に予算し、後段の統合ボトルネックを回避。

52.3 シミュレーションでのロボット訓練

前述の制御アーキテクチャと駆動・センシングの機械的制約を踏まえ、本小節ではシミュレーション上でヒューマノイドを訓練し、実機に適用可能な頑健で転移可能な動作を生成する方法を扱う。焦点は、コントローラソフトウェア、機械設計の限界、導入リスクを結ぶ実践的なエンジニアリング手順にある。

問題定義と運用目標：

- ・アクチュエータトルク限界，関節可動域，センサ遅延を尊重した歩行または操縦ポリシーを訓練する。
- ・環境ばらつきや未モデル化の接触に対して頑健な動作を生成する。
- ・sim-to-real 転移性を最大化し，実機での試行錯誤を最小化する。

技術分析（主要コンポーネントと指標）：

1. 状態，行動，報酬の設計

- ・観測ベクトルはプロプライオセプション，IMU，必要に応じた処理済みビジョンを組み合わせる．文中では q , \dot{q} , imu で関節角，角速度，慣性計測を表す．
- ・アクション空間は低次 PD ループの関節位置目標でもよく，高精度アクチュエータがモデル化されている場合は直接トルクでもよい．
- ・報酬は性能と安全性およびエネルギーをバランスさせる．典型的なスカラー報酬：

$$r_t = w_p r_{\text{progress}} + w_e r_{\text{energy}} + w_s r_{\text{safety}}.$$
- ・客観指標で評価：成功率，平均トルクマージン，ゼロモーメント点（ZMP）安定性，メートルあたりエネルギー，時間あたり故障イベント数．

2. ダイナミクス忠実度と制御階層

- ・ハイブリッドアプローチを用いる：低次アクチュエータダイナミクスを可能な限りシミュレートしながら，制御スタックに PD ループを残して安定性を確保する．ロボットダイナミクスは

$$[H]M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau + J_c(q)^T f_c, \quad (413)$$

で与えられる．ここで M は質量行列， C は遠心・コリオリ項， g は重力， τ はアクチュエータトルク， J_c は接触ヤコビアン， f_c は接触力である．

- ・離地時や足底配置での脆い挙動を避けるため，接触コンプライアンスと摩擦円錐をモデル化する．

3. 学習パラダイムと転移戦略

- ・強化学習（RL）：期待収益目的を定義

$$[H]J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^T \gamma^t r_t \right], \quad (414)$$

ここで π_θ はポリシーパラメータ化, γ は割引率である.

- 模倣学習: モーションキャプチャまたはスクリプト化された軌道を用いてポリシーを初期化し, 探索リスクを低減する.
- ドメインランダムマイゼーションとシステム同定: 質量, 重心オフセット, 摩擦, センサノイズ, 時延をランダムイズし, ポリシー包絡を広げる.
- 特権訓練: 訓練時にポリシーに追加状態 (完全状態アクセス) を与え, 教師・生徒法で観測セットに蒸留して展開可能にする.

実装チェックリスト (ステップバイステップ):

1. Isaac Sim 環境構築

- 衝突ジオメトリとアクチュエータ限界を含む正確な URDF/Xacro モデルを組み込む.
- 接触コンプライアンスとソフト関節ダンピングを実装する.

2. コントローラと行動インタフェース

- 低次 PD コントローラをシミュレーションループでラップし, アクション空間として目標位置またはトルクを公開する.

3. カリキュラムと報酬シェイピング

- 補助環境から開始: 重力摂動を低減, 目標速度を遅くする.
- 徐々に外乱の大きさと複雑さを増やす.

4. 評価ループと安全チェック

- 未モデル化の不整地やセンサドロップアウトを含むホールドアウトシナリオに対して定期的に検証する.
- 関節トルク飽和を監視し, 飽和が頻発する場合は報酬に制約を追加する.

コード例: 軽量 Isaac Sim RL 環境ラッパー. スニペットは PD 制御, ドメインランダムマイゼーション, 報酬コンポーネントを示す.

コードサンプル 173 Isaac Sim 環境ステップ: PD 制御とドメインランダムマイゼーション付き

```
import numpy as np
import gymnasium as gym
from gymnasium import spaces
from typing import Dict, Tuple, Any
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32MultiArray, Bool
from sensor_msgs.msg import JointState
from geometry_msgs.msg import Twist
import torch
import yaml
from pathlib import Path

class HumanoidEnv(gym.Env):
```

```
"""Production-ready humanoid environment with ROS 2 integration."""
```

```
def __init__(self, sim, config_path: str = None):
    super().__init__()

    # 設定ファイル読み込み
    if config_path is None:
        config_path = Path(__file__).parent / "config" / "humanoid_config.yaml"

    with open(config_path, 'r') as f:
        self.config = yaml.safe_load(f)

    self.sim = sim
    self.node = Node('humanoid_env')

    # ROS 2 publishers/subscribers
    self.obs_pub = self.node.create_publisher(Float32MultiArray, '/humanoid/observation', 10)
    self.reward_pub = self.node.create_publisher(Float32MultiArray, '/humanoid/reward', 10)
    self.done_pub = self.node.create_publisher(Bool, '/humanoid/done', 10)

    # Action/Observation spaces
    self.action_space = spaces.Box(
        low=np.array(self.config['action_limits']['low']),
        high=np.array(self.config['action_limits']['high']),
        dtype=np.float32
    )

    obs_dim = self.config['observation']['dim']
    self.observation_space = spaces.Box(
        low=-np.inf, high=np.inf, shape=(obs_dim,), dtype=np.float32
    )

    # PD制御ゲイン
    self.Kp = np.array(self.config['control']['kp'])
    self.Kd = np.array(self.config['control']['kd'])
    self.torque_limit = self.config['control']['torque_limit']

    # 報酬重み
    self.reward_weights = self.config['reward']['weights']
```

```

# 内部状態
self.step_count = 0
self.max_steps = self.config['episode']['max_steps']
self.prev_action = np.zeros_like(self.Kp)

def randomize_dynamics(self) -> None:
    """動的パラメータのランダム化（ドメインランダム化）"""
    # 質量のランダム化
    mass_scale = 1.0 + self.config['randomization']['mass_std'] * np.random.randn()
    self.sim.set_mass_scale(np.clip(mass_scale, 0.8, 1.2))

    # 摩擦係数のランダム化
    friction = self.config['randomization']['base_friction'] + \
        self.config['randomization']['friction_range'] * np.random.rand()
    self.sim.set_friction(np.clip(friction, 0.5, 1.0))

    # 重心位置のランダム化
    com_offset = np.random.uniform(
        -self.config['randomization']['com_range'],
        self.config['randomization']['com_range'],
        size=3
    )
    self.sim.set_com_offset(com_offset)

def reset(self, seed=None, options=None) -> Tuple[np.ndarray, Dict]:
    """環境のリセット"""
    super().reset(seed=seed)

    self.step_count = 0
    self.prev_action = np.zeros_like(self.Kp)

    # シミュレーションリセット
    self.sim.reset()

    # ランダム化適用
    if self.config['randomization']['enabled']:
        self.randomize_dynamics()

    # 初期状態取得
    obs = self._compute_observation()

```

```

# ROS 2メッセージ配信
self._publish_observation(obs)

return obs, {}

def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, bool, Dict]:
    """1ステップ実行"""
    # アクションのクリッピングと平滑化
    action = np.clip(action, self.action_space.low, self.action_space.high)
    action = 0.8 * action + 0.2 * self.prev_action # 低周波フィルタ

    # 現在状態取得
    q, qdot = self.sim.get_prorioception()

    # PD制御によるトルク計算
    tau = self.Kp * (action - q) - self.Kd * qdot
    tau = np.clip(tau, -self.torque_limit, self.torque_limit)

    # トルク適用とシミュレーションステップ
    self.sim.apply_torques(tau)
    self.sim.step()

    # 観測と報酬計算
    obs = self._compute_observation()
    reward, terminated = self._compute_reward(q, qdot, tau, action)

    # ステップ数管理
    self.step_count += 1
    truncated = self.step_count >= self.max_steps

    # ROS 2メッセージ配信
    self._publish_observation(obs)
    self._publish_reward(reward)
    self._publish_done(terminated or truncated)

    self.prev_action = action

    return obs, reward, terminated, truncated, {}

```

```

def _compute_observation(self) -> np.ndarray:
    """観測ベクトルの計算"""
    q, qdot = self.sim.get_proprioception()
    base_orientation = self.sim.get_base_orientation()
    base_angular_vel = self.sim.get_base_angular_velocity()
    base_linear_acc = self.sim.get_base_linear_acceleration()

    # 観測正規化
    obs = np.concatenate([
        q / np.pi, # 関節角度正規化
        np.clip(qdot / 10.0, -1.0, 1.0), # 関節速度正規化
        base_orientation, # ベース姿勢
        np.clip(base_angular_vel / 5.0, -1.0, 1.0), # 角速度正規化
        np.clip(base_linear_acc / 20.0, -1.0, 1.0), # 線形加速度正規化
        self.prev_action # 前回アクション
    ])

    return obs.astype(np.float32)

def _compute_reward(self, q: np.ndarray, qdot: np.ndarray,
                    tau: np.ndarray, action: np.ndarray) -> Tuple[float, bool]:
    """報酬関数計算"""
    # 前進速度報酬
    forward_vel = self.sim.get_forward_velocity()
    target_vel = self.config['reward']['target_velocity']
    progress = -abs(forward_vel - target_vel)

    # エネルギー効率報酬
    energy = -np.sum(np.abs(tau * qdot)) * self.reward_weights['energy']

    # 姿勢安定性報酬
    orientation_error = np.linalg.norm(self.sim.get_base_orientation()[:2])
    stability = -orientation_error * self.reward_weights['stability']

    # 滑らかさ報酬
    smoothness = -np.sum(np.abs(action - self.prev_action)) * self.reward_weights['smoothness']

    # 関節制限ペナルティ
    joint_limits = self.sim.get_joint_limits_violation()
    joint_penalty = -joint_limits * self.reward_weights['joint_limits']

```

```

# 転倒判定
is_fallen = self.sim.is_fallen()
if is_fallen:
    fall_penalty = self.reward_weights['fall']
else:
    fall_penalty = 0.0

total_reward = progress + energy + stability + smoothness + joint_penalty + fa

return float(total_reward), is_fallen

def _publish_observation(self, obs: np.ndarray) -> None:
    """ROS2観測メッセージ配信"""
    msg = Float32MultiArray()
    msg.data = obs.tolist()
    self.obs_pub.publish(msg)

def _publish_reward(self, reward: float) -> None:
    """ROS2報酬メッセージ配信"""
    msg = Float32MultiArray()
    msg.data = [reward]
    self.reward_pub.publish(msg)

def _publish_done(self, done: bool) -> None:
    """ROS2終了メッセージ配信"""
    msg = Bool()
    msg.data = done
    self.done_pub.publish(msg)

def close(self) -> None:
    """環境クリーンアップ"""
    self.node.destroy_node()
    self.sim.close()

```

評価と転移：

- 工場床、階段、ソフトマットを模したランダムイズ済みテストセットを使用する。
- 少数のハードウェア走行データからシステム同定データを収集し、シミュレータパラメータを洗練する。
- トルク、関節限界、圧力中心を監視する保守的な安全監督器を伴って展開する。

エンジニアリングへの影響，トレードオフ，運用リスク：

- ・シミュレータ忠実度を上げると sim-to-real ギャップは縮まるが，計算コストが増しイテレーションが遅くなる．
- ・ランダム化をやりすぎると，過度に保守的なポリシーとなりタスク性能が低下する．
- ・特権訓練は学習を加速するが，限られたセンサセットへの堅牢な蒸留が必要である．
- ・運用リスクには，不現実な摩擦を悪用したポリシーによるアクチュエータ過熱や，ハードウェア限界近傍での高関節トルクが含まれる．トルクペナルティ，安全監督器，制御負荷下での段階的実機テストで緩和する．

52.4 ロボットのテストと展開

シミュレーションで学習したポリシーと事前に調整した制御ソフトウェアは，テストおよび展開時に用いる合格/不合格基準と期待される運用範囲を決定する．本小節では問題、ヒューマノイドを検証するための技術解析、実装チェックリスト、およびその後の運用トレードオフを記述する．

問題の記述．代表的な運用条件下で，フルヒューマノイドシステムが安全・機能・信頼性要件を満たすことを検証する．主要要件は以下の通り：

- ・安全性：予測可能な停止、接触時のコンプライアント挙動、機能する非常停止。
- ・性能：歩容安定性、操縦精度、知覚遅延、バッテリー持続時間。
- ・信頼性：ミッション中の許容失敗率と回復可能性。

技術解析．要件を測定可能なテストにマッピングし，合格閾値を定義するテスト計画を作成する．以下の構造化されたアプローチを用いる：

1. メトリクスと目標を定義する。

- ・安定性マージン：歩行中の最大許容圧力中心逸脱。
- ・知覚精度：対象環境での物体検出の適合率と再現率。
- ・タイミング：制御ループジッタと最悪遅延はタスク固有の閾値以下に留める。
- ・信頼性：ミッション継続時間 t にわたる所望のシステム信頼性 $R(t)$ 。失敗を定常ポアソン過程としてモデル化し，信頼性は

$$[H]R(t) = e^{-\lambda t}, \quad (415)$$

ただし λ は失敗率、 $MTBF = 1/\lambda$ は平均故障間隔。

- ・有病率 p の欠陥クラスを n 回の独立試行後に検出する信頼度は $1 - (1 - p)^n$ 。信頼度 c を達成するには

$$[H]n \geq \frac{\log(1 - c)}{\log(1 - p)}. \quad (416)$$

2. テストモダリティを選択する。

- ・ユニットテスト：アクチュエータドライバ、運動学ライブラリ、知覚パイプラインのエッジケース。
- ・統合テスト：関節限界の実施、トルク飽和処理、ビヘイビアツリーの遷移。
- ・ハードウェアインザループ（HIL）：実アクチュエータにシミュレート環境ダイナミクス。
- ・システムレベル実地試験：制御されたラボシナリオから徐々に非制御環境へ。

- ・フォルトインジェクション：センサ脱落、通信喪失、固着関節、電源低下。
3. リスクとコストでテストに優先順位を付ける。高リスク項目は足滑り下でのバランスとペイロード搬送時の非常停止挙動を含む。

実装：段階的テストと展開。合格基準を満たさないと次段階に進めないゲート付きパイプラインを採用する。

- ・ステージ A — ベンチ検証：
 1. CI で自動ユニット・統合テストを実行。
 2. ソフトウェアタイミングとリソース使用量の静的チェックを実施。
 3. シャーシの電源および EMI 事前チェック。
- ・ステージ B — 制御 HIL：
 1. アクチュエータとセンサを HIL リグに接続。
 2. シミュレート環境擾乱下で閉ループテストを実行。
 3. emergency_stop、ウォッチドッグタイマ、ソフトウェアリミットを検証。
- ・ステージ C — ラボ試験：
 1. テザー歩行・操縦試験に人間監督者を配置。
 2. 予期せぬ接触の有無を力・トルクセンサで監視。
 3. 実世界遅延と再現性を測定。
- ・ステージ D — 実地展開：
 1. 対象環境での限定・監督運用。
 2. テレメトリストリーミングとリモートロールバック機能を有効化。
 3. 自律性とミッション複雑性を徐々に増加。

自動テストとヘルスモニタリングは不可欠である。以下の ROS2 スタイルヘルスモニタは定期的なセンサ・アクチュエータチェックを実施し、ステータス文字列を公開し、致命的な失敗時に安全停止をトリガする。

コードサンプル 174 基本プレフライトチェック用 ROS2 ヘルスモニタ

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from std_msgs.msg import String
from diagnostic_msgs.msg import DiagnosticArray, DiagnosticStatus, KeyValue
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
import time
import threading
from typing import List, Tuple

class HealthMonitor(Node):
    def __init__(self) -> None:
```

```

super().__init__('health_monitor')

# パラメータ定義
self.declare_parameter('heartbeat_timeout', 1.0,
                        ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE,
                                             description='センサ生存タイムアウト'))
self.declare_parameter('motor_temp_limit', 75.0,
                        ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE,
                                             description='モータ温度上限[℃]'))
self.declare_parameter('battery_soc_min', 0.2,
                        ParameterDescriptor(type=ParameterType.PARAMETER_DOUBLE,
                                             description='バッテリー残量下限[0-1]'))

# QoS: 緊急時も到達確実に
qos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    durability=DurabilityPolicy.TRANSIENT_LOCAL,
    depth=1
)

# パブリッシャ
self._diag_pub = self.create_publisher(DiagnosticArray, '/diagnostics', qos)
self._status_pub = self.create_publisher(String, '~/status', 10)

# タイマ
self._timer = self.create_timer(0.5, self._check_all)

# 診断データ保持
self._last_hb_time = time.monotonic()
self._motor_temp = 0.0
self._battery_soc = 1.0
self._sensor_ok = True
self._motor_ok = True
self._battery_ok = True
self._emergency_triggered = False
self._lock = threading.Lock()

def _check_all(self) -> None:
    with self._lock:
        now = time.monotonic()

```

```

hb_elapsed = now - self._last_hb_time
self._motor_temp = self._read_motor_temp()
self._battery_soc = self._read_battery_soc()

self._sensor_ok = hb_elapsed < self.get_parameter('heartbeat_timeout').value
self._motor_ok = self._motor_temp < self.get_parameter('motor_temp_limit')
self._battery_ok = self._battery_soc > self.get_parameter('battery_soc_min')

status_str = "OK" if all([self._sensor_ok, self._motor_ok, self._battery_ok]) else "FAULT"
self._status_pub.publish(String(data=status_str))

# 診断メッセージ生成
diag = DiagnosticArray()
diag.header.stamp = self.get_clock().now().to_msg()
diag.status = [
    DiagnosticStatus(
        level=DiagnosticStatus.OK if self._sensor_ok else DiagnosticStatus.ERROR,
        name='health_monitor:␣sensor_heartbeat',
        message='Heartbeat␣alive' if self._sensor_ok else 'Heartbeat␣lost',
        values=[KeyValue(key='elapsed_sec', value=str(hb_elapsed))]
    ),
    DiagnosticStatus(
        level=DiagnosticStatus.OK if self._motor_ok else DiagnosticStatus.ERROR,
        name='health_monitor:␣motor_temp',
        message='Temperature␣normal' if self._motor_ok else 'Overheat',
        values=[KeyValue(key='temperature_celsius', value=str(self._motor_temp))]
    ),
    DiagnosticStatus(
        level=DiagnosticStatus.OK if self._battery_ok else DiagnosticStatus.ERROR,
        name='health_monitor:␣battery_soc',
        message='SOC␣adequate' if self._battery_ok else 'SOC␣low',
        values=[KeyValue(key='soc_ratio', value=str(self._battery_soc))]
    )
]
self._diag_pub.publish(diag)

if status_str == "FAULT" and not self._emergency_triggered:
    self.get_logger().error('致命的な故障、安全停止を発行')
    self._trigger_emergency_stop()
    self._emergency_triggered = True

```

```

# ハードウェアI/O (オーバーライド推奨)
def _read_sensor_heartbeat(self) -> None:
    # 実機ではセンサから最終受信時刻を更新
    self._last_hb_time = time.monotonic()

def _read_motor_temp(self) -> float:
    # 実機ではCAN等から取得
    return 45.0

def _read_battery_soc(self) -> float:
    # 実機ではBMSから取得
    return 0.85

def _trigger_emergency_stop(self) -> None:
    # 実機ではモータコントローラへ緊急停止コマンド送信
    pass

def main(args=None) -> None:
    rclpy.init(args=args)
    node = HealthMonitor()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

運用テレメトリと飛行後解析。高周波制御トレース、エンドツーエンド知覚タイムスタンプ、電源バスメトリクスをログに記録する。同期タイムスタンプ（NTP または PTP）を用い、ストレージ冗長性を確保する。失敗をオフラインで解析し、経験的 λ を算出し設計 MTBF と比較する。

展開ガバナンスと安全事例。テスト、故障モード、緩和策、人間介入ポリシーを文書化した安全事例を作成する。該当規制および研究所安全規則への適合を確保する。

設計トレードオフと運用リスク：

- 保守性対性能：厳格なトルクリミットは安全性を高めるが敏捷性を劣化。
- シミュレーションへの過適合：sim-to-real ギャップを減らすため訓練シナリオを多様化。

- ・テレメトリ対帯域：大量ログは診断に役立つが制約ネットワーク下でリアルタイム制御を阻害。
- ・単一障害点：冗長センシングとグレースフルデグラデーションはミッション喪失を減らす但し質量・電力を増加。

具体的な工学上の影響：段階的実地試験のためのスケジュール時間を割り当て、予期せぬ負荷に備えてアクチュエータヘッドルームを確保し、安全計測器およびテスト治具の予算を計上する。新たな欠陥有病率推定が得られたら式 (2) を用いて必要試行回数を定期的に再計算する。

53 学んだ教訓と今後の方向性

53.1 開発中に直面した課題

機械設計の選択と制御アルゴリズムを踏まえ、本小節ではプロトタイプ開発および実地試験中に繰り返し遭遇した技術的課題を列挙する。議論では実際の故障を定量的診断と人型プラットフォームで用いた緩和戦略に結びつける。

問題定義：現実環境でバランスを保ち、操作し、知覚する頑健な移動型ヒューマノイドを実現する。主要な課題は、動力学・バランス、駆動・電源、センシング・知覚、ソフトウェア統合という 4 つの相互作用する領域に横断的に生じた。各領域は測定可能な故障モードとトレードオフを生み、設計判断を形作った。

技術分析と主要故障モード

・動力学・バランス：

- 小さな重心高さ誤差とセンサ遅延が歩行中のキャプチャ失敗を引き起こす。線形倒立振子モデル (LIPM) は簡潔な診断を与える。 z_c を公称重心高さ、 g を重力加速度とする。固有角振動数は

$$[H]\omega = \sqrt{\frac{g}{z_c}}. \quad (417)$$

キャプチャ点 x_{cp} はロボットが停止するために踏み出すべき位置を予測する：

$$[H]x_{cp} = x + \frac{v}{\omega}, \quad (418)$$

ここで x は水平重心位置、 v はその速度である。 x または v の誤差は歩行配置に直接伝播する。

- 地面相互作用の不確実性（滑り、コンプライアンス）は支持多角形推定に余裕を要求する。実地試験では少なくとも 20 の幾何学的余裕が転倒を大幅に減少させた。

・駆動・電源：

- 連続高トルク動作がモータと駆動電子機器を加熱する。関節トルク要求は

$$[H]\tau(t) = I\ddot{\theta}(t) + b\dot{\theta}(t) + \tau_{\text{grav}}(\theta), \quad (419)$$

としてモデル化され、ここで I は慣性、 b は粘性減衰、 τ_{grav} は重力項である。不十分なトルク余裕は停動条件を過小評価する。

- バッテリ放電は重心を変化させ、電圧サグはモータ帯域幅を低下させる。試験では電圧降下と関節達成速度の最大 15 減少が相関した。

- センシング・知覚：
 - 視覚パイプラインは極端な照明と反射面下で故障する．深度センサは艶あり床近傍で非対称誤差分布を報告し，足場検出を破綻させた．
 - IMU バイアスドリフトと構造コンプライアンスが徐々に増大する姿勢誤差を生じた．補償には頻繁なキャリブレーションと推定器融合が必要だった．
- ソフトウェア統合とリアルタイム制御：
 - ネットワーク化モジュールはジッタを導入する．センサからコントローラへの 10 ms ジッタは高速動作で有意なキャプチャ点予測誤差に変換した．
 - シミュレーション・リアリティギャップ：摩擦，減衰，関節バックラッシュが Isaac Sim と実機で異なり，非物理的動力学を悪用するポリシーを生じた．

実装戦術と適用緩和策

1. 状態推定の改善：
 - IMU，脚オドメトリ，接触センシングを拡張カルマンフィルタ（EKF）で融合する．接触遷移中は過程雑音を適応的に増加させる．
 - IMU と足力・トルクセンサから信頼度重み付けを行うキャプチャ点フィルタを実装する．
2. 駆動の頑健性向上：
 - 連続・ピークトルクを 2 倍安全率で仕様化する．焼損防止のため熱監視と電流制限を追加する．
 - ギアボックスバックラッシュ補償と逐次同定を用いて関節摩擦をオンラインでモデル化する．
3. 知覚の強化：
 - 近距離深度センシング用の能動照明と床検出フォールバック用ライダーを組み合わせる．
 - 照明に対する脆弱性を減らすため，視覚モデルにドメインランダムイズドシミュレーションを用いる．
4. ソフトウェアとスケジューリング：
 - ハード制御ループをリアルタイムハードウェアに移行する．非リアルタイムネットワークは高次計画に留める．
 - 遅延源を測定しモジュールごとに遅延予算を強制するため，エンドツーエンドトレーシングを用いる．

具体的検証指標と受容閾値

- 1000 歩あたりの転倒回数 < 1 （水平屋内床面）．
- 公称速度で 5 分連続歩行時の関節温度上昇 $< \text{環境温度} + 50^{\circ}\text{C}$ ．
- 敏捷な旋回中のキャプチャ点予測誤差 $\text{RMS} < 2 \text{ cm}$ ．
- 指定照明条件下での知覚誤検出足場率 $< 3\%$ ．

代表的コード：キャプチャ点を計算し歩行プランナに公開する ROS2 ノード

コードサンプル 175 キャプチャ点計算用シンプル ROS2 ノード

```
import rclpy
from rclpy.node import Node
```

```

from geometry_msgs.msg import Point, TwistStamped
from rcl_interfaces.msg import ParameterDescriptor, ParameterType
import math

class CapturePointNode(Node):
    def __init__(self) -> None:
        super().__init__('capture_point_node')

        # パラメータ宣言と取得
        self.declare_parameter(
            'com_height',
            0.9,
            ParameterDescriptor(
                description='公称重心高さ[m]',
                type=ParameterType.PARAMETER_DOUBLE,
                read_only=True
            )
        )
        self.zc: float = self.get_parameter('com_height').value
        self.g: float = 9.81

        # パブリッシャ／サブスクライバ
        self._pub = self.create_publisher(Point, '~/capture_point', 10)
        self._sub = self.create_subscription(
            TwistStamped, '~/com_state', self._cb_state, 10
        )

        self.get_logger().info(f'CapturePointNode started (zc={self.zc})')

    def _cb_state(self, msg: TwistStamped) -> None:
        # 重心速度と位置を TwistStamped から取得
        v = msg.twist.linear.x
        x = msg.twist.linear.z # zフィールドをx座標に流用

        omega = math.sqrt(self.g / self.zc)
        x_cp = x + v / omega

        pt = Point()
        pt.x = x_cp
        pt.y = 0.0

```

```

        pt.z = 0.0
        self._pub.publish(pt)

def main(args=None):
    rclpy.init(args=args)
    node = CapturePointNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

```

技術的影響，トレードオフ，運用上のリスク

- 高トルク・センサ冗長性は質量とコストを増加させ、飛行時間と敏捷性を低下させる．トレードオフはミッション主導で最適化する必要がある．
- 制御を決定論的ハードウェアに移行するとジッタは減少するが、開発複雑性と認証負荷が増大する．
- シミュレーションに過適合したコントローラは開発を加速するが、実世界での壊滅的故障のリスクを孕む．
- 運用上のリスクには熱暴走、電気故障、予測不能な人間との相互作用が含まれる．義務的緩和策として熱カットアウト、ウォッチドッグ、保守的安全エンベロープが必要である．

設計者は性能と安全余裕，保守性，コストをバランスさせる必要がある．定量的受容指標は反復を導き、配備されたヒューマノイドシステムにとって最も致命的な故障モードを緩和する．

53.2 スケーラビリティの最適化

スケーラビリティの最適化は、より多くのロボット、より多くの自由度、より大きなデータワークロードをサポートするために、アーキテクチャ、ツール、プロセスを変更することでこれらの制約に対処する。

問題設定とスケーラビリティの軸。ヒューマノイドシステムにおいて「スケーラビリティ」は少なくとも3つの工学軸にまたがる：

- メカニカル・運動学的スケール：ジョイントやセンサを追加すると制御の複雑さが増す。
- フリートスケール：単一のプロトタイプから数十～数百台の展開ユニットへ移行すること。
- データおよび計算スケール：大規模なシミュレーションバッチやニューラルモデルの学習には並列GPUとネットワーク推論が必要。

明確な性能目標は制御ループのレイテンシ予算である。安定歩行および操作用に、エンドツーエンドループは

$$[H]T_{\text{sense}} + T_{\text{est}} + T_{\text{ctrl}} + T_{\text{act}} < T_{\text{period}}, \quad (420)$$

を満たす必要があり、ここで T_{period} はリアルタイム制御周期（例：500 Hz ジョイント制御なら 1/500 s）である。各コンポーネントは測定され上限を設定する必要がある。ジョイント数 n が増えるにつれ、計算ステップの一部は n に比例してスケールするため、レイテンシの余裕は減少する。

計算およびアルゴリズムのスケールリング。計算量が n に対して線形または準線形で増加するアルゴリズムを選ぶ。関節剛体ダイナミクスでは：

- 再帰的ニュートン・オイラー逆ダイナミクス実装は $O(n)$ ランタイムを達成する。
- 単純な密ヤコビアン逆行列は $O(n^3)$ のコストを伴い、数十ジョイントを超えると実用的でない。

したがって、木構造とスパース性を活用するアルゴリズムファミリーを優先する。状態推定および制御では：

- 増分ソルバ（例：iSAM）を持つ因子グラフ推定器を使用して、時間ウィンドウにわたって計算を償却する。
- モデル予測制御（MPC）では、予測ホライズンを制限するか、スパース性を活用する QP ソルバを使用する。

データおよび学習のスケールリング。記録されたセンサストリームのストレージコスト S は概ね

$$[H]S \approx R_{\text{sens}} \cdot B_{\text{per_sample}} \cdot H, \quad (421)$$

でスケールし、ここで R_{sens} はセンサ全体のサンプリングレート、 $B_{\text{per_sample}}$ はサンプルあたりのバイト数、 H は収集時間である。ストレージおよび I/O コストを制御するために、ラベル付きデータの保持、圧縮、部分集合選択を計画する。

実装のためのアーキテクチャパターン。ハードリアルタイム制御をソフトパーセプションおよびクラウドサービスから分離する層化ハイブリッドアーキテクチャを採用する：

- リアルタイムコア：確定的なシングルボードコンピュータまたは RTOS で、ジョイントレベル制御と安全チェックを処理。
- ローカルミドルウェア：リアルタイム拡張付き ROS2、デッドラインセンシティブトピック用に QoS を調整。
- パーセプション/推論：複数のロボットからバッチリクエストを受け付ける GPU バックエンドサービス。
- フリートサービス：クラウドで管理されるオーケストレーション、OTA アップデート、テレメトリ、シミュレーションリプレイ。

実践的な戦略：

1. 制御を四肢ごとのコントローラにモジュール化し、監督歩行マネージャを配置する。これによりプロセスごとの状態サイズが削減され並列化が可能。
2. ロボット間またはシミュレーションインスタンス間でニューラルネットワーク推論をバッチ化し、GPU カーネル起動オーバーヘッドを償却。
3. コンテナ化されたデプロイメントと CI/CD パイプラインを使用して、調整されたアップデートを安全にリリース。
4. Isaac Sim ヘッドレスモードで大規模並列シミュレーションを実行し、GPU クラスタ上でオーケ

ストレーションしてデータ生成。

例：バッチ推論サーバパターン。以下のスニペットは、観測を集約し、シミュレーション内の複数のヒューマノイドエージェントに対して単一の GPU 順伝播を実行する簡単なバッチンググループを実装する。これによりエージェントごとのオーバーヘッドが削減されスループットが向上する。

コードサンプル 176 複数のヒューマノイドエージェント向けの簡単なバッチ GPU 推論。

```
import os
import time
import signal
import logging
from queue import Queue, Empty
from threading import Thread, Event
from dataclasses import dataclass
from typing import Optional, Tuple, List

import torch
import torch.nn as nn
from torch.cuda.amp import autocast

# ログ設定
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s_[(levelname)s]_[(name)s]:_[(message)s]",
    handlers=[logging.StreamHandler()])
logger = logging.getLogger(__name__)

# 環境変数でバッチサイズ・デバイスを上書き可能
BATCH_SIZE = int(os.getenv("BATCH_SIZE", 32))
DEVICE = os.getenv("TORCH_DEVICE", "cuda" if torch.cuda.is_available() else "cpu")

# シャットダウン用グローバルイベント
shutdown_event = Event()

@dataclass
class InferenceItem:
    """推論リクエスト1件"""
    obs_id: int
    obs: torch.Tensor
```

```

class PolicyModel(nn.Module):
    """本番用は外部ファイルからロード"""
    def __init__(self, ckpt_path: str):
        super().__init__()
        # 例: torch.load(ckpt_path, map_location="cpu")
        # self.backbone = ...
        # self.head = ...
        logger.info(f"Model loaded from {ckpt_path}")

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # 実装省略
        return x

class BatchInferenceService:
    def __init__(
        self,
        model: nn.Module,
        device: str,
        batch_size: int,
        max_delay: float = 0.01,
        num_threads: int = 1,
    ):
        self.model = model.to(device)
        self.device = device
        self.batch_size = batch_size
        self.max_delay = max_delay # バッチが埋まらない場合の最大待機[s]

        self.in_queue: Queue[InferenceItem] = Queue()
        self.out_queue: Queue[Tuple[int, torch.Tensor]] = Queue()

        self.threads: List[Thread] = []
        for _ in range(num_threads):
            t = Thread(target=self._worker, daemon=True)
            t.start()
            self.threads.append(t)

    def _worker(self):

```

```

""" バッチ推論ワーカー """
while not shutdown_event.is_set():
    batch: List[InferenceItem] = []
    deadline = time.time() + self.max_delay

    # バッチを溜める
    while len(batch) < self.batch_size and time.time() < deadline:
        try:
            item = self.in_queue.get(timeout=0.001)
            batch.append(item)
        except Empty:
            continue

    if not batch:
        continue

    # 推論
    ids = [it.obs_id for it in batch]
    x = torch.stack([it.obs for it in batch]).to(self.device)
    with torch.no_grad(), autocast(enabled=self.device.startswith("cuda")):
        actions = self.model(x)

    # 結果を返す
    for idx, act in zip(ids, actions.cpu()):
        self.out_queue.put((idx, act))

def submit(self, obs_id: int, obs: torch.Tensor):
    self.in_queue.put(InferenceItem(obs_id, obs))

def get(self, timeout: Optional[float] = None) -> Tuple[int, torch.Tensor]:
    return self.out_queue.get(timeout=timeout)

def stop(self):
    shutdown_event.set()
    for t in self.threads:
        t.join()

# グレースフルシャットダウン
def _signal_handler(signum, frame):

```

```

        logger.info("Shutdown signal received")
        shutdown_event.set()

signal.signal(signal.SIGINT, _signal_handler)
signal.signal(signal.SIGTERM, _signal_handler)

# 初期化
model = PolicyModel("policy.ckpt")
service = BatchInferenceService(model, DEVICE, BATCH_SIZE)

# 使用例
if __name__ == "__main__":
    # シミュレータ側
    dummy_obs = torch.randn(4, 84, 84)
    service.submit(agent_id=0, obs=dummy_obs)

    # 制御側
    aid, action = service.get(timeout=0.1)
    logger.info(f"Agent_{aid}->_action_{action}")

```

シミュレーションのスケールアップとハードウェアへの転送。ドメインランダム化とパラメータアンサンブルを使用し、数百のシミュレートされたヒューマノイドで学習したポリシーがハードウェアに一般化することを保証する。重要な実践：

- Isaac Sim を使用して GPU あたり数千のシミュレーションインスタンスを並列化し、多様な軌道を収集。
- sim-to-real ギャップを早期に検出するためのハードウェア試行の検証セットを維持。
- デバッグのための再現可能な実験メタデータとシード付きランダム性を優先。

運用上のトレードオフとリスク。スケラビリティを最適化する際は以下を考慮：

- レイテンシ vs 集中化：集中推論はアップデートを簡素化するが、ネットワークリスクとレイテンシを増加させる。エッジバッチングはレイテンシを減らす但ハードウェアコストが上昇。
- 複雑さ vs 保守性：マイクロサービスアーキテクチャはスケールするが、堅牢な CI/CD とオペラビリティを必要とする。
- コスト vs 性能：より多くの GPU はデータ生成と学習を加速するが、運用費用が増加。
- 安全性とセキュリティ：OTA アップデートとリモート制御インターフェースは攻撃対象を拡大する。認証、再現可能なロールバック、安全インターロックを実装。

ヒューマノイドプロジェクトへの設計上の影響：

- 計算量が自由度に対して適切にスケールする制御アルゴリズムを選ぶ。

- ハードリアルタイムループをスケーラブルサービスから分離するようアーキテクト。
- シミュレーションスケール機能と自動化パイプラインに早期に投資。
- 式 (1) でレイテンシ予算を定量化し、システムに計測と強制手段を組み込む。

これらの選択により、ヒューマノイド設計が単一のラボプロトタイプから保守可能なフリートへと移行し、予測可能な性能を提供できるかどうかが決まる。

53.3 ヒューマノイドロボティクスの次のステップ

スケーラビリティ戦略と具体的な開発課題を踏まえ、次のステップは実験室プロトタイプと実用ヒューマノイドシステムのギャップを埋めることに集中する。これらのステップは、ハードウェア、制御、知覚、シミュレーション、安全検証にわたる統合ソリューションを重視する。

問題提起と優先順位。実地対応型ヒューマノイドは長時間にわたり信頼性高く動作し、人間と安全に対話し、新しいタスクに適応しなければならない。主要な工学目標は以下の通りである：

- より長いミッションのためのエネルギー効率と熱管理を改善する；
- 非構造化環境での知覚頑健性を高める；
- 物理モデルと学習ベース適応性を融合した制御アーキテクチャを提供する；
- 証明可能な安全性と予測可能な故障モードを確保する。

技術分析：ハードウェアとエネルギー。アクチュエータレベルの効率が耐久性を支配する。Series Elastic Actuators (SEA) および準直接駆動モータは反射慣性を低減しバックドライバビリティを改善し、接触タスク時のコンプライアンスを向上させる。電池のエネルギー密度向上は漸進的であるため、システムレベル戦略が重要である：

1. 回生ブレーキを可能にするためアクチュエータと電力電子機器を共設計する；
2. ホットスワップとミッション計画のためのモジュラー電池パック；
3. 連続トルクスケジューリングとディレーティングのための熱モデル。

知覚とセンシング。頑健な知覚には多モーダル融合が必要である。ビジョン、ライダー、プロプリオセプション、触覚センシングを確率的融合バックエンドで統合する。センサヘルスマニタリングとオンラインで動作する動的キャリブレーションパイプラインを実装する。知覚モデルでは以下を重視する：

- 重度のドメインランダムマイゼーションで学習されたドメイン一般特徴抽出器；
- 意思決定ゲーティングのための不確実性推定；アンサンブルまたはベイズ近似を使用。

制御アーキテクチャ：ハイブリッドモデルベースおよび学習ベース制御。Model Predictive Control (MPC) は制約処理と明示的計画を提供する。学習コンポーネントは残差ダイナミクスと高次ポリシーを供給する。ホライズン N における歩行のための実用的な MPC コストは：

$$[H]J = \sum_{k=0}^{N-1} \|x_{k+1} - x_{k+1}^{\text{ref}}\|_Q^2 + \|u_k - u_k^{\text{ref}}\|_R^2, \quad (422)$$

ダイナミクス $x_{k+1} = f(x_k, u_k)$ と接触制約に従う。MPC と学習済み残差ポリシー π_r を組み合わせてモデルミスマッチを補正する：

$$[H]u_k = u_k^{\text{MPC}} + \pi_r(x_k, \hat{d}_k), \quad (423)$$

ここで \hat{d}_k はモデル化されていない外乱を推定する。

安全性と検証可能性。Control Barrier Functions (CBF) はランタイムで安全制約を強制する。CBF は安全状態に対して $h(x) \geq 0$ を満たす連続微分可能関数 $h(x)$ である。以下を強制する：

$$[H]\dot{h}(x, u) + \alpha(h(x)) \geq 0, \quad (424)$$

クラス K 関数 α を用いる。各制御ステップで小規模な二次計画 (QP) を解き、公称入力を最小限に修正しながら安全性を保持する。

シミュレーションとシムツールリアル転移。高忠実度、GPU 加速シミュレーション (Isaac Sim) を大規模データ生成とテストに使用する。重要なステップ：

- システム同定を実施し $f(\cdot)$ とノイズモデルをフィット；
- ダイナミクス、ビジュアル、レイテンシでドメインランダムマイゼーションを適用；
- プログレッシブネットワークまたは残差学習を用いてハードウェア上でポリシーを適応。

実装ブループリント。段階的工学計画を優先する：

1. 漸進的アップグレードのためハードウェアおよびソフトウェアインターフェースをモジュール化；
2. リッチテレメトリとオンラインシステム ID のためシステムに計測器を装備；
3. ハイブリッドスタックを実装：MPC プランナー、残差 RL ポリシー、CBF 安全フィルタ；
4. シミュレーションとハードウェアテストを接続する継続的インテグレーションパイプラインを自動化。

コンパクトな例として、CBF ベース安全フィルタを二次計画として実装する。このフィルタは制御周波数で動作し、公称コマンドを最小限調整してロボットを安全に保つ。

コードサンプル 177 cvxpy を用いた CBF 安全フィルタ QP 例

```
import cvxpy as cp
import numpy as np
from typing import Optional

class CBF_QP:
    """
    1次元平面ダイナミクス用のCBF-QP安全フィルタ
    """
    def __init__(self,
                 A: np.ndarray,
                 B: np.ndarray,
                 x_safe_min: float = 0.0,
                 k: float = 5.0,
                 u_max: float = 2.0,
                 solver: Optional[str] = None) -> None:
```

```

self.A = A
self.B = B
self.x_safe_min = x_safe_min
self.k = k
self.u_max = u_max
self.solver = solver or cp.OSQP

def filter(self, x: np.ndarray, u_nom: np.ndarray) -> Optional[np.ndarray]:
    h = x[0] - self.x_safe_min
    Lfh = self.A[0, :].dot(x)
    Lgh = float(self.B[0, :]) # 1x1行列をスカラー化

    u = cp.Variable(1)
    # CBF条件:  $h_{\dot{}} + k \cdot h \geq 0$ 
    constraints = [Lfh + Lgh * u[0] + self.k * h >= 0,
                   cp.abs(u) <= self.u_max]

    prob = cp.Problem(cp.Minimize(cp.sum_squares(u - u_nom)), constraints)
    try:
        prob.solve(solver=self.solver)
        return u.value if prob.status == cp.OPTIMAL else None
    except Exception:
        return None

# 使用例
if __name__ == "__main__":
    A = np.array([[0, 1], [0, 0]])
    B = np.array([[0], [1]])
    cbf = CBF_QP(A, B, x_safe_min=0.0, k=5.0, u_max=2.0)

    x = np.array([0.2, -0.1])
    u_nom = np.array([0.5])
    u_safe = cbf.filter(x, u_nom)
    if u_safe is not None:
        print("u_safe=", u_safe[0])

```

運用上の考慮とトレードオフ。あらゆる拡張はトレードオフを伴う：

- 知覚と MPC のためのより多くの計算は重量と消費電力を増加；
- より高いコンプライアンスは安全性を改善するが高速俊敏性を低下；

- 重度のドメインランダムマイゼーションは頑健性を改善するが収束を遅延；
- 形式的検証は信頼性を高めるがコントローラ表現力を制限。

工学への影響とリスク。モジュラーアップグレードと測定可能性能指標を優先する。オンラインシステム同定と安全臨界モニタリングに投資する。現在の電池技術により耐久性は制約されることを受け入れる。運用上のリスクを軽減するため：

- 安全性を層化（ハードウェア停止、CBF、監視下自律性）；
- シナリオ分布にわたる継続的テスト；
- メンテナンスと修理可能性のための明示的計画。

設計チームのための具体的即時アクション：

- 式 (1)(2) の通り MPC-plus-residual パイプラインを実装；
- ランタイムフィルタとして Listing 1st:cbf_{qp} のような CBF 安全 QP を追加；
- エネルギーと熱テレメトリのためアクチュエータに計測器を装備；
- Isaac Sim 実験をハードウェア検証へ接続する CI を構築。

これらのステップは配備準備度を高めるが、規律ある統合、性能測定、保守的安全マージンを要する。