第五章 WRS の概説

作成：Weiwei Wan，修正：Weiwei Wan, Takuya Kiyokawa

2024 年度 秋 冬学期

1 WRS のシステム構成とそれに基づくロボットプログラミング

WRS のソースコードは以下の複数のフォルダあるいはパッケージによって構成されます.

名前	用途
0000_xxxx	xxxx は任意の文字です．この名前のフォルダはただのフォルダを示しており，関数を提供するためのパッケージをではありません．本書籍のサンプルプログラムがこれらのフォルダに保存されています.
_future	将来的に追加される予定のパッケージです．将来は正式な独立したパッケージとして生成するか既存のパッケージと結合する予定です.
basis	計算や表示のために共通的に使用するパッケージです．下に trimesh という子パッケージと constant.py, data_adapter.py, robot_math.py, trimesh_generator.py など四つのファイルを含んでいます. ◇ constant.py には，色の値などのよく使われる変数が定義されています．こちらの内容の多くは，色の値の定義において matplotlib ライブラリを参考にしており，いくつかのカラーマップの名前も matplotlib と一致しています. ◇ data_adapter.py には WRS が依存する異なるパッケージ間のデータ型を変換するための関数を用意しています．例えば，data_adapter.npmat3_to_pdmat3(npmat3) 関数は引数として受けた numpy の 3×3 の行列 npmat3 を panda3d の Mat3 に変換します．また，たとえば，data_adapter.gen_colorarray(n_colors=1, alpha=1, seed_rgb=None) 関数は n_colors 個の 1×4 のリストを生成します．1×4 の各リストの四つの要素は red, green, blue, alpha の強さを示します．data_adapter.pdgeom_from_vfnf(vertices, face_normals, triangles, name='auto') 関数は引数として与えられた numpy の n×3 の行列 vertices，m×3 の行列 face_normals，m×3 の行列 triangles を用いて，panda3d の Geom を生成します. ◇ robot_math.py では三次元回転用の関数，例えば回転の軸－角度形式の表現から行列に変換する rotmat_from_axangle(axis, angle) 関数（ロドリゲスの回転公式に基づく変換）などが実装されています. ◇ trimesh_factory.py には basis.trimesh.Trimesh 型の各種メッシュを生成する関数を用意しています．trimesh_factory.py に実装された関数は一般ユーザーが使用するものはほとんどありません．modeling パッケージから呼び出され，GeometricModel や CollisionModel 型のオブジェクトを生成する際に使用されます.
bench_mark	このフォルダは，テスト用にさまざまなモデルを保存するために使用されています．現在，YCB（Yale-CMU-Berkeley）ロボットグリップライブラリの一部の物体が格納されています．このライブラリは，ロボットの把持，操作，物体認識などに広く使用される標準化データセットで，食品，工具，容器など日常生活で使用されるさまざまな物体をカバーしています．異なる形状，サイズ，材質の物体を含むことで，ロボットの把持アルゴリズムの性能を評価・比較できるようになっています.

（続き）

名前	用途
drivers	周辺機器のドライバー，RPC 通信用のサーバーとクライアント，ロボットのドライバーなどがこのパッケージに保存されています．ドライバーはメーカーから提供されている API を Python で利用できるように変換したり，簡略化したりしています．特に，ロボットのドライバーは robot_con の下にあるロボットとの通信制御用の関数に利用されます．ユーザー側に直接呼び出されることはありません．
grasping	把持をユーザ自身が定義するあるいは自動計画するための関数はこのパッケージに収められています．また，reasoner.py には複数の目標把持姿勢に対する共通的な非衝突の把持姿勢を計算用のクラスが定義されています．
helper	仮想世界やロボットを簡単に定義するための補助クラスや関数などがこのパッケージに収められています．これらの補助クラスや関数を使うことで，プログラムは簡単に記述できるようになります．例えば，ur3_dual_helper.UR3DualHelper クラスのオブジェクトを定義すると，そのオブジェクトが初期化される際に，自動的にロボットシミュレーション用のメンバ変数，計画用のメンバ変数，逆運動計算用のメンバ変数，ピックアンドプレースの動作生成用のメンバ変数，仮想世界のメンバ変数などが定義されます．また，与えた引数によってロボットとの通信制御用のメンバ変数なども定義されます．helper を使用しない場合は，以上のメンバ変数は自作したファイル内で各パッケージをインポートして細かく定義しないといけません．現時点で，このパッケージは現在，詳細な開発は行っておらず，拡張用途として残してあります．
manipulation	grasping の把持姿勢の定義用の関数に対して，manipulation パッケージでは操作に関する関数，特に対象物を把持してからの作業動作を生成する関数を保存しています．例えば，ピックアンドプレースでは一連の動作，ハンドを物体に近づけ，把持して，持ち上げて，目標姿勢に運んで，そしてハンドを物体から離す動作を生成するための関数などを用意しています．
modeling	このパッケージでは仮想世界に表示できるオブジェクトを定義するためのクラスや関数が実装されています．特に，geometric_model.py と collision_model.py は WRS の他の多くのパッケージに利用されています．geometric_model.py には幾何的なモデルオブジェクトを定義するためのクラス GeometricModel と各種幾何形状のモデルオブジェクトを生成する関数を用意しています．collision_model.py には，衝突検出用のプリミティブの定義や衝突検出用のクラス CollisionModel などが実装されています．geometric_model.py と collision_model.py においては，いずれでも basis.trimesh.Trimesh や直接にハードディスクから読み込んだ.stl，.dae モデルなどで初期化して利用できます．WRS はメートル・ラジアン・キログラムを寸法の単位としてメッシュモデルや物体を取り扱っています．ただし，様々な方法で取得したモデルはメートル・ラジアン・キログラムの単位以外で作成されている場合も多いです．mesh_tool.py にはモデルの寸法を調整するための関数を用意されているため，それを利用して単位を統一しておくと良いでしょう．モデルをハードディスクから読み込んで，寸法を変更した上で，保存することができます．GeometricModel と CollisionModel 以外にも，このパッケージにはダイナミックシミュレーション用のオブジェクトを定義するクラスや関数なども用意されています．ダイナミックシミュレーションは現状 Mujoco に切り替えしていますので，これから廃棄する予定です．

（続き）

名前	用途
motion	ロボットの運動経路生成から運動軌跡生成までに必要なクラスや関数を含んでいます．運動経路は，時間に依存しない一連の関節角度の変化のシーケンスです．一方，運動軌跡は，時間に関連する関節角度の変化のシーケンスです．probabilistic フォルダーには，確率に基づく運動経路計画アルゴリズムが含まれており，例えば rrt_connect.py などがあります．これらを使用することで，衝突のない運動経路を生成できます．primitives フォルダーの interpolated.py には，関節空間または作業空間の 2 つの点間でさまざまな補間を行い，スムーズで連続的な運動経路シーケンスを生成するための補間アルゴリズムが含まれています．approach_depart_planner.py はより高レベルのクラスで，インスタンスは複数の目標位置と回転を受け取り，ロボットがこれらの位置に移動し，その後離れるための動作経路シーケンスを生成します．このファイルで定義されたメソッドは，ボタンを押すなどの動作を生成する際に使用されることがあります．trajectory フォルダーには運動軌跡を生成するためのさまざまなアルゴリズムが含まれており，現在は速度が速く，比較的使いやすいのが topp_ra.py ファイルの時間最適プランナーです．この軌跡プランナーは，運動経路シーケンスを制御周期，最大速度，最大加速度の制約に基づいて補間し，時間に関連する関節角のシーケンスを出力します．
robot_con	こパッケージは，ロボットの名前ごとにさらに多くのサブパッケージに分かれており，それぞれにロボットを制御するためのクラスや関数が含まれています．マニピュレータの制御は主に，ネットワークを介してコントローラにバッファをアップロードし，順次実行する形が一般的です．一方で，エンドエフェクタの制御はより多様で，マニピュレータの末端の IO ポートを介して透過的に制御できるものもあります．この制御は，Modbus や RS485 などの産業用プロトコルに基づくことが多いですが，Dynamixel モーターのプロトコルのようなカスタムプロトコルも使用される場合があります．また，マニピュレータによっては，エンドエフェクタを透過的に制御できず，WRS をインストールした PC に直接接続する必要がありますが，この場合の制御通信の原理は同じで，物理的な接続は通常 USB シリアル通信や RJ45 ケーブルと変換器を組み合わせて行います．
robot_sim	ロボットの定義やシミュレーション用のクラスや関数がこのパッケージに実装されています．このパッケージの詳しい構成については次の節で詳しく説明しますので，ここでは省略します．
vision	カメラキャリブレーション，センサーデータの処理などのための関数がこのパッケージに実装されています．特に ar_marker フォルダー内の make_pdfboard.py ファイルには，AR マーカーの PDF ファイルを生成するための各種関数が含まれており，これを使用して必要なサイズの AR マーカー PDF ファイルを直接生成し，印刷して使用できます．各関数は，境界線を追加する機能も提供しており，その後の裁断が容易になります．また，レーザー切断機と組み合わせることで，高精度なマーカーボードを制作することができます．depth_camera および rgb_camera フォルダーには，それぞれ深度カメラと RGB カメラで一般的に使用されるレジストレーションアルゴリズムが含まれており，詳細については関連する章で説明します．
visualization	二次元や三次元画面を書き出すための関数がこのパッケージに実装されています．現在，matplotlib と panda3d しか使えません．将来的には他のツール，例えば WebGL への拡張も考えています．

0000_xxxx のフォルダには WRS のパッケージを活用した複数のサンプルプログラムを用意しています.「第二章 Python 統合開発環境」に説明された 0000_examples/ur3e_dual_left_planning.py はその一つです．ここで，類似なプログラム（リスト 5.1-1）の中身を見てみましょう．ur3e_dual_left_planning.py と比べて簡単に記述されています．障害物の読み込みやロボットの初期化，画面に書き出す方法については，このファイルを参考にできます.

リスト 5.1-1: UR3 ロボットの定義と表示

```
1  import visualization.panda.world as wd # 三次元の仮想環境や表示画面の定義用
2  import modeling.geometric_model as mgm # 各種幾何形状（例えば矢印や座標系など）の定義用
3  import modeling.collision_model as mcm # 衝突検出可能な各種幾何形状の定義用
4  import robot_sim.robots.ur3_dual.ur3_dual as ur3d # ロボットシミュレーションの定義用
5  import numpy as np # 行列計算用
6  import basis.robot_math as rm # 座標計算用
7
8  if __name__ == '__main__':
9      base = wd.World(cam_pos=[7, 2, 4], lookat_pos=[0, 0, 1])
10     mgm.gen_frame(ax_length=.7, ax_radius=.01).attach_to(base) # グローバル座標系
11     obstacle = mcm.CollisionModel("./objects/milkcarton.stl") # 衝突検出用モデルの定義
12     obstacle.pos = np.array([.55, -.3, 1.3]) # 位置の設定
13     obstacle.rotmat = rm.rotmat_from_euler(-np.pi/3, np.pi/6, np.pi/9) # 回転の設定
14     obstacle.rgba = np.array([.5, .7, .3, .5]) # 色の設定
15     obstacle.attach_to(base) # 画面への表示
16     mgm.gen_frame(ax_length=.3, ax_radius=.007).attach_to(obstacle) # 障害物のローカル座標系
17     # ロボットシミュレーション関連
18     robot_s = ur3d.UR3Dual() # シミュレーション用のロボットの定義
19     robot_model = robot_s.gen_meshmodel(alpha=.5) # 現在の姿勢を用いてメッシュを生成
20     robot_model.attach_to(base) # 生成したメッシュを画面に表示
21     base.run() # 仮想環境を実行
```

上記ファイルの最初の 1~5 行目では，プログラム実行に必須のパッケージをインポートしています．各パッケージの用途は，各行末尾に付与しているコメントを参考にしてください．6 行目は空行です．これは，Python の慣習的なルールで，全ての import が終わった後に 1 行空けてインポート部分とそれ以外の部分の判別がつきやすくなる工夫です．7 行目以降はプログラムのメイン部分です．7 行目は，このファイルが直接実行された場合のみに実行する部分であることを示す記述です．そのため，この記述以下（インデントされている部分）は，このファイルが直接実行されていない場合は実行されません．実際には，このファイルを指定して Python を実行した場合，___name___ という固有変数は文字列 “___main___” の値を持ちます．7 行目では___name___と “___main___” が同じ値を持つかどうかを判断し，同じの場合，この記述以下を実行します．リスト 5.1-1 には実行したいコードしか書いていないので，7 行目を削除しても構いません．ただし，他のファイルに使われる関数やクラスも定義した場合，別のファイルからインポートされる場合には実行したくないコードがあれば，この記述以下にコードを書いてください．

リスト 5.1-1 の 8~21 行目についても，各行末尾のコメントを参考にしてください．ここで，8 行目で定義した仮想環境は関数 run() を呼び出さないとプログラムが実行されないので注意してください．また，一旦 base.run() を呼び出すと，run() 以降のコードは無視されます．この特性を活用すれば，コードの任意行に base.run() を挿入して，その行までの内容を一旦画面に表示することも可

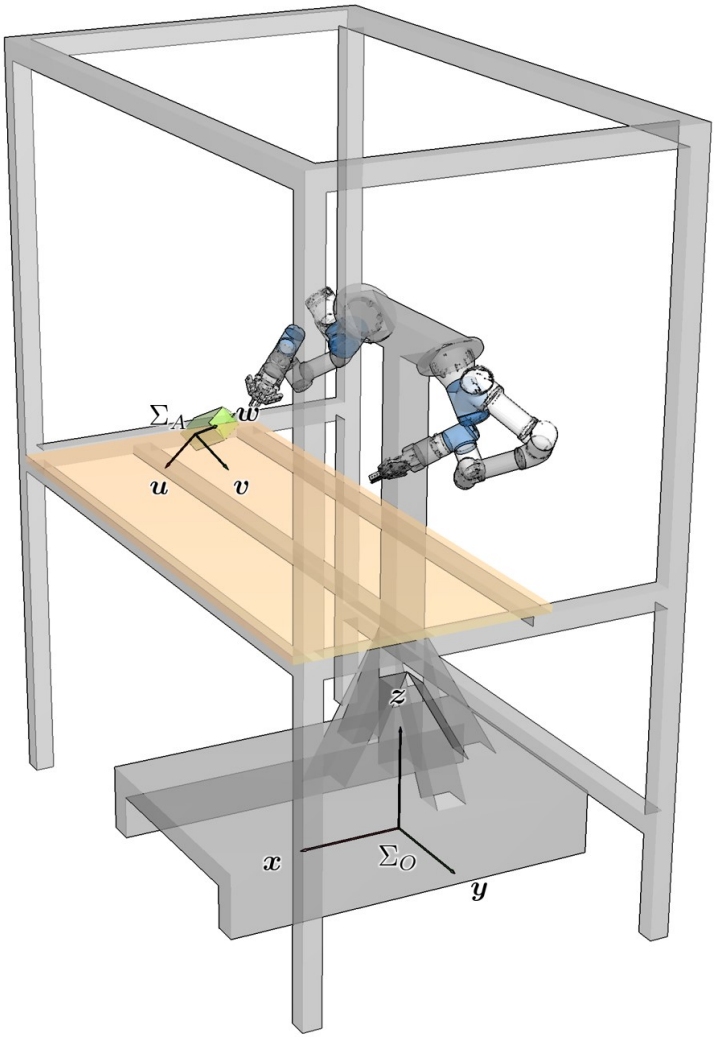

図 1: リスト 5.1-1 の結果．Σ_O はグローバル座標系，Σ_A は対象物のローカル座標系．

能です．プログラムのディバッグにも非常に役立ちます．

図 1 はリスト 5.1-1 のプログラム実行結果である表示画面を示しています．Σ_O はグローバル座標系で，デフォルト設定では，ロボットのローカル座標と重なります．WRS で定義したロボットのローカル座標系は，ロボット前方が $\boldsymbol{x}$，左手側の方向が $\boldsymbol{y}$，鉛直上方向が $\boldsymbol{z}$ という一定のルールに基づいて定義されます．Σ_A は対象物のローカル座標系で，Σ_O から rm.rotmat_from_euler(-np.pi/3, np.pi/6, np.pi/9) の回転行列をかけた姿勢となる座標系です．対象物のローカル座標系を画面に表示するためには，プログラムの 16 行目に示すように，mgm.gen_frame() によって取得した座標系の GeometricModel を対象物の Model に attach_to することで実現できます．

WRS の使用開始時には，新たに「0001_個人の英記苗字」という名前のフォルダを作成し，そのフォルダ内にプログラムを実装していくことをお勧めします．そうすることで，WRS 側のソースコードを更新する際に，github から新しいバージョンをダウンロードして，自作したフォルダをその新しくダウンロードしたフォルダにコピーするだけで済み，他の部分を触らずに作業を続けることができます．一方，WRS 側のソースコードは，そのような更新を行う場合に消してしまう可能性があるため，既存のパッケージ内のファイルの直接編集は避けるようにするといいでしょう．

2 モデリング

2.1 CAD モデルの作成

第 1 節で示したプログラムの 11 行目ではハードディスクにある “./objects/milkcarton.stl” という CAD モデルを読み込んで CollisionModel を初期化しています．“./objects/milkcarton.stl” という表記は，プログラムファイルが入っているフォルダ 0000_book の下の objects フォルダの下の milkcarton.stl を指しています．そのため，あらかじめ 0000_book/objects の下に milkcarton.stl を用意しないといけません．添え字の.stl はファイル形式を示しています．stl は汎用のメッシュモデルを保存する形式の一種です．ここでは詳しい説明は割愛します．.stl 以外には，.obj, .dae, .ply など色んな形式があります．大体点，面，法線，色，テクスチャ，材質などのデータが含まれていて，異なる形式間に微妙な差が存在します．WRS ではモデルの点，面，法線しか用いず，それ以外の情報を無視しています．また，現在の WRS では，.stl, .dae, .ply の形式の読み込みをサポートしています．.stl, .dae, .ply 形式のファイルを用いて GeometricModel や CollisionModel を初期化することは可能です．WRS は.stl 形式のファイルをデフォルトのモデルファイルとして取り扱います．.stl ファイルはバイナリ形式と ASCII 形式という二種類の保存方式があります．バイナリ形式で保存された.stl ファイルのサイズが小さい・読み込みやすいためおすすめです．

.stl, .dae, .ply 形式の CAD ファイルを作成するのは OnShape というオンラインのツールを使うことをおすすめします．公式ホームページ（https://www.onshape.com）から，大学のメールで無料の登録および利用ができます．図 2（左）では OnShape のインターフェースを示しています．ウェブブラウザを通ってオンラインで操作可能ですので，PC へのソフトのインストールが不要です．詳しい使い方は，ホームページのチュートリアルを参考にしてください．オンラインで作成したファイルをパソコンに保存する際，バイナリ形式の.stl で保存することが望ましいです．なお，対象物のローカル座標系は保存する時に調整できないので，モデルの作成前に適当に設定してください．OnShape を使う一つの利点はモデルの共有と共同編集が可能である点です．他のアカウント（メール）に対して，権限付きで共有できます．図 2（右）が共有する際の画面を示しています．

2.2 基本幾何形状の定義と表示

CAD モデルファイルを読み込む以外の方法として，basis.trimesh.Trimesh 型の変数を引数として，GoemetricModel と CollisionModel に渡すことで初期化することもできます．basis.trimesh_factory.py には色んな basis.trimesh.Trimesh 型のメッシュモデルを定義する関数が用意されています．これらの関数の戻り値は basis.trimesh.Trimesh 型の変数ですので，引数として GeometricModel と CollisionModel に渡して色んな基本幾何形状を初期化でき

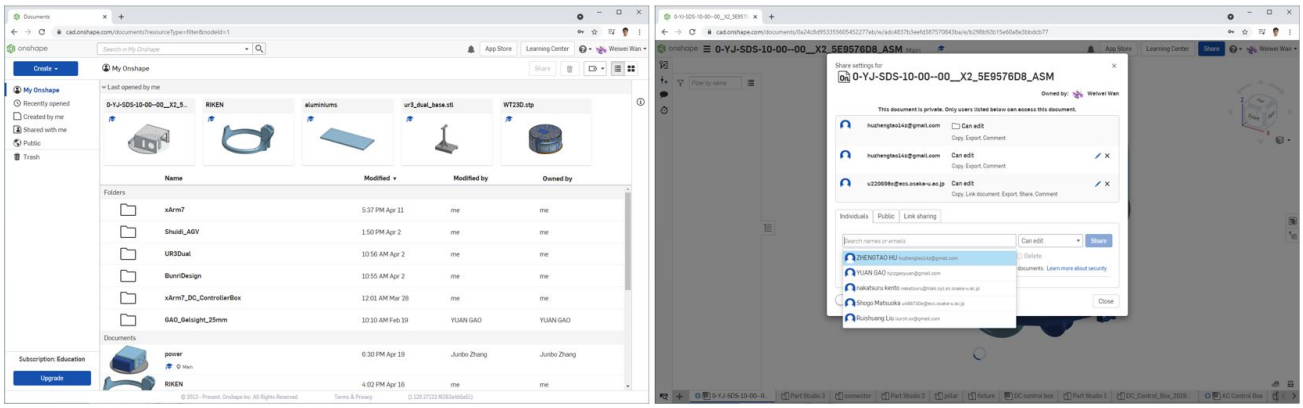

図 2: オンライン CAD モデル作成用ツールである OnShape のインターフェース（左）と作成したモデルを共有する際の操作画面（右）.

ます．例えば，リスト 5.2-2 のプログラムでは basis.trimesh_factory.gen_sphere() で定義した Trimesh 型の球を GeometricModel に渡して GeometricModel 型のモデルに変換して画面に表示します[1].

リスト 5.2-2: GeometricModel 型のモデル生成と表示

```
import numpy as np
import basis.trimesh_factory as trf
import modeling.geometric_model as mgm
import visualization.panda.world as wd

if __name__ == '__main__':
    base = wd.World(cam_pos=np.array([.3, .3, .3]), lookat_pos=np.zeros(3))
    tg_sphere = trf.gen_sphere(pos=np.zeros(3), radius=.05)  # Trimesh型の球を定義
    gm_sphere = mgm.GeometricModel(tg_sphere)  # GeometricModel型のモデルに変換
    gm_sphere.attach_to(base)  # 画面に表示
    base.run()
```

WRS では，リスト 5.2-2 のプログラムの機能を簡単に実装するために，geometric_model.py と collision_model.py の中でいくつかの関数を提供しています．球を定義する場合には geometric_model.py の 483~499 行目に実装されているリスト 5.2-3 の関数を呼び出すことで簡単に実装できます.

リスト 5.2-3: 球の幾何モデルを生成する関数

```
def gen_sphere(pos=np.array([0, 0, 0]),
               radius=0.0015,
               rgb=np.array([1, 0, 0]),
               alpha=1,
               ico_level=2):
    """この関数の説明を省略します"""
    sphere_trm = trm_factory.gen_sphere(pos=pos, radius=radius, ico_level=ico_level)
    sphere_sgm = StaticGeometricModel(initor=sphere_trm, rgb=rgb, alpha=alpha)
    return sphere_sgm
```

ここで注意してもらいたいのは gen_sphere 関数の返り値が GeometricModel 型ではなく，StaticGeometricModel 型である点です．StaticGeometricModel 型は位置姿勢を調整することができない幾何モデルで，GeometricModel クラスと同じく geometric_model.py に定義されています．geometric_model.py と collision_model.py には計四つの XXXXModel のクラスを定義しています．それぞれの用途と継承関係は下の表にまとめています.

[1]現在，WRS の三次元環境は panda3d を用いて構築したものです．Trimesh 型のオブジェクトは直接に画面に表示できません．画面に表示するためには GeometricModel（衝突検出機能なし）或いは CollisionModel（衝突検出機能あり）に変換する必要があります.

クラス名	用途
StaticGeometricModel	geometric_model.py ファイルに定義された最も基礎的な幾何モデルに関するクラスです．静的（Static）なものですので，位置姿勢の調整や衝突検出などの機能がついていません．画面表示用です．
WireFrameModel	StaticGeometricModel から派生したクラスです．geometric_model.py ファイルに定義されています．StaticGeometricModel と同じく静的（Static）なものですが，画面に表示するときに対象の幾何モデルのメッシュのエッジのみがレンダリングされます．
GeometricModel	StaticGeometricModel から派生したクラスです．geometric_model.py ファイルに定義されています．位置姿勢を調整する機能を追加したものです．
CollisionModel	collision_model.py ファイルに定義されています．geometric_model.GeometricModel から派生したクラスです．位置姿勢の調整や衝突検出などの機能が含まれています．

2.3 CollisionModel の定義と表示

collision_model.py ファイルに定義された CollisionModel クラスも geometric_model.GeometricModel クラスから派生したものですが，衝突検出の機能が付いており，初期化する際には，細かい衝突用の引数の設定が可能です．リスト 5.2-4 は collision_model.py の 34 行目 ~44 行目の CollisionModel クラスの初期化関数に記載される引数です．

リスト 5.2-4: CollisionModel の初期化関数に記載される引数

```
    def __init__(self,
                 initor=None, # データ，ファイルやTrimeshなど
                 name="collision_model",
                 cdprim_type=mc.CDPType.AABB, # 大まかな衝突検出用のモデルタイプの設定
                 cdmesh_type=mc.CDMType.DEFAULT, # 精密な衝突検出用のモデルタイプの設定
                 ex_radius=None, # 大まかな衝突検出用のモデル拡張半径
                 userdef_cdprim_fn=None, # 本文中で詳しく説明します
                 toggle_transparency=True, # 透明のレンダリングを認めるかどうか
                 toggle_twosided=False, # 面の表側と裏側両方をレンダリングするかどうか
                 rgb=rm.bc.tab20_list[0], # 色，basis.constantに定義された値を参照
                 alpha=1): # 透明度，0は完全透明，1は不透明
```

CollisionModel が取り扱う衝突検出はプリミティブに基づいたものとメッシュに基づいたものの二種類があります．それぞれリスト 5.2-4 の cdprimit_type と cdmesh_type 引数によって設定されます．プリミティブに基づいた衝突検出では，簡単化した形状，例えば元の詳細なモデルを囲むボックス形状を利用し，そのボックス同士の重なりを衝突として検出します．メッシュに基づいた衝突検出は元のモデルのメッシュ間の重なりを計算します．プリミティブに基づいた衝突検出と比べ，メッシュに基づいた衝突検出は色んな面と点の交叉を求めるため，非常に計算負荷が高く，動作計画など頻繁に衝突検出を呼び出す処理には不向きです．一方，把持計画は指と対象物の間の衝突を細かくチェックする必要があるため，メッシュを用いて細かく衝突検出を行わないといけません．リスト 5.2-5 のプログラムでは複数の異なるプリミティブとメッシュを設定したウサギの CollisionModel を作って，元のモデルとその衝突モデルを画面に描き出しています．このコードを実行して，プリミティブとメッシュの差，そしてプリミティブ同士とメッシュ同士の差を確認してみましょう．

リスト 5.2-5: ウサギの CollisionModel の定義と表示

```
import numpy as np
import modeling.collision_model as mcm
import visualization.panda.world as wd

if __name__ == '__main__':
    base = wd.World(cam_pos=np.array([.7, .05, .3]), lookat_pos=np.zeros(3))
    # ウサギのモデルのファイルを用いてCollisionModelを初期化します
```

```
    # ウサギ1~5はこのCollisionModelのコピーとして定義します
    object_ref = mcm.CollisionModel(initor="./objects/bunnysim.stl",
                                    cdprim_type=mcm.mc.CDPType.AABB,
                                    cdmesh_type=mcm.mc.CDMType.DEFAULT)
    object_ref.rgba = np.array([.9, .75, .35, 1])
    # ウサギ1
    object1 = object_ref.copy()
    object1.pos = np.array([0, -.18, 0])
    # ウサギ2
    object2 = object_ref.copy()
    object2.pos = np.array([0, -.09, 0])
    # ウサギ3
    object3 = object_ref.copy()
    # ウサギ3の衝突検出用のプリミティブを表面にサンプリングした球形状のsurface_ballsへ変更します
    object3.change_cdprim_type(cdprim_type=mcm.mc.CDPType.SURFACE_BALLS, ex_radius=.01)
    object3.pos = np.array([0, .0, 0])
    # ウサギ4
    object4 = object_ref.copy()
    object4.pos = np.array([0, .09, 0])
    # ウサギ5
    object5 = object_ref.copy()
    # ウサギ5の衝突検出用のメッシュを凸包convex_hullへ変更します
    object5.change_cdmesh_type(cdmesh_type=mcm.mc.CDMType.CONVEX_HULL)
    object5.pos = np.array([0, .18, 0])
    # ウサギ1の画面表示．元のモデルのみ書き出します
    object1.attach_to(base)
    # ウサギ2の画面表示．元のモデル上に，デフォルトのプリミティブ(box)も表示します
    object2.attach_to(base)
    object2.show_cdprim()
    # ウサギ3の画面表示．元のモデル上に，新たに設定したプリミティブ(surface_balls)も表示します
    object3.attach_to(base)
    object3.show_cdprim()
    # ウサギ4の画面表示．元のモデル上に，デフォルトのメッシュも表示します
    object4.attach_to(base)
    object4.show_cdmesh()
    # ウサギ5の画面表示．元のモデル上に，新たに設定したメッシュ(convex_hull)も表示します
    object5.attach_to(base)
    object5.show_cdmesh()
    base.run()
```

図 3(a) には上記のコードを実行した結果が示されています．(a.1)~(a.5) はコードから画面へ書き出した 1~5 番目のウサギです．(a.1) は元のモデルを示しています．(a.2) と (a.3) は元のモデル上にデフォルトのプリミティブ (modeling.constant.CDPType.AABB) と新たに設定したプリミティブ (modeling.constant.CDPType.SURFACE_BALLS) を表示しています．いずれにしても，大まかに表現した簡単な形状によって，元のモデルを囲んでおり，衝突検出はこれらの大まかな形状同士の重なりを検出します．(a.4) と (a.5) は元のモデルの上でデフォルトのメッシュ(modeling.constant.CDMType.DEFAULT，三角メッシュ) と新たに設定したメッシュ(modeling.constant.CDMType.CONVEX_HULL) を示しています．このメッシュに基づいた衝突検出は，図中に示された各三角メッシュ間の重なりを細かくチェックするため，計算量が多いです．

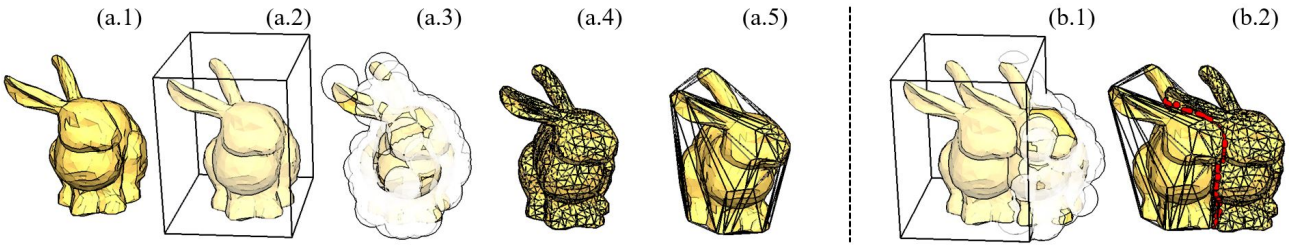

図 3: (a) 異なる cdprim_type と cdmesh_type を設定したウサギの CollisionModel の比較．(a.1) 元のモデル．(a.2) AABB 型のプリミティブ．(a.3) SURFACE プリミティブ．(a.4) 元のモデルの三角メッシュ．(a.5) 凸包の三角メッシュ．(b) CollisionModel 同士の衝突検出の結果．(b.1) プリミティブ (AABB) とプリミティブ (SURFACE_BALLS) 間の衝突検出．(b.2) メッシュ (CONVEX_HULL) とメッシュ (DEFAULT) 間の衝突検出．

2.4 CollisionModel 間の衝突検出

CollisionModel 同士の衝突検出はクラス内の is_pcdwith() あるいは is_mcdwith() メンバ関数によって行われます．is_pcdwith は “is primitive collided with” の略語で，同様に is_mcdwith は “is mesh collided with” の略語です．リスト 5.2-6 のプログラムでは is_pcdwith と is_mcdwith を用いた衝突検出の例を示しております．

リスト 5.2-6: ウサギのモデルを用いた衝突検出の例

```
... # importや__name__の判断を省略
    base = wd.World(cam_pos=np.array([.7, .05, .3]), lookat_pos=np.zeros(3))
    # ウサギのモデルのファイルを用いてCollisionModelを初期化します
    # ウサギ1~5はこのCollisionModelのコピーとして定義します
    object_ref = mcm.CollisionModel(initor="./objects/bunnysim.stl",
                                    cdprim_type=mcm.mc.CDPType.AABB,
                                    cdmesh_type=mcm.mc.CDMType.DEFAULT)
    object_ref.rgba = np.array([.9, .75, .35, 1])
    # ウサギ1，デフォルトのプリミティブ（AABB)のまま使います
    object1 = object_ref.copy()
    object1.pos = np.array([0, -.07, 0])
    # ウサギ2，プリミティブ（SURFACE_BALLS)に変更します
    object2 = object_ref.copy()
    object2.pos = np.array([0, -.04, 0])
    object2.change_cdprim_type(cdprim_type=mcm.mc.CDPType.SURFACE_BALLS, ex_radius=.01)
    # ウサギ1と2とそのプリミティブを画面に表示します
    object1.attach_to(base)
    object1.show_cdprim()
    object2.attach_to(base)
    object2.show_cdprim()
    # ウサギ1と2のプリミティブ間の衝突を検出します
    pcd_result = object1.is_pcdwith(object2)
    print(pcd_result) # 衝突の結果を出力します
    # ウサギ3，メッシュ（CONVEX_HULL)に変更します
    object3 = object_ref.copy()
    object3.change_cdmesh_type(cdmesh_type=mcm.mc.CDMType.CONVEX_HULL)
    object3.pos = np.array([0, .04, 0])
    # ウサギ4，デフォルトのメッシュのまま使います
    object4 = object_ref.copy()
    object4.pos = np.array([0, .07, 0])
    # ウサギ3と4とそのプリミティブを画面に表示します
    object3.attach_to(base)
    object3.show_cdmesh()
    object4.attach_to(base)
    object4.show_cdmesh()
    # ウサギ3と4のメッシュ間の衝突を検出します．ここで，引数toggle_contactsをTrueにして，衝突
      した点も呼び出し側に戻します
    mcd_result, cd_points = object3.is_mcdwith(object4, toggle_contacts=True)
    print(mcd_result) # 衝突の結果を出力します
    # 検出した衝突点を画面に表示します
    for pnt in cd_points:
        mgm.gen_sphere(pos=pnt, rgb=np.array([1, 0, 0]), alpha=1, radius=.002).attach_to(base)
    base.run()
```

上記のプログラムを実行すると，図 3(b) に示す結果が得られます．また，コンソールの標準出力では True が二回出力されます．それぞれ上記のコードの 27 行目と 41 行目に対応します．特に，関数 is_mcdwith は引数 toggle_contacts を受け取ります．この引数を真に設定した場合，衝突したかどうかのほかにも，衝突する点も呼び出し側に戻します．戻った点を gm.gen_sphere を用いて画面に表示すれば，衝突の細かいチェックが可能となります．図 3(b.2) の赤い点群は 272~273 行目の gm.gen_sphere(pos=pnt, rgba=[1, 0, 0, 1], radius=.002) で書き出した衝突が検出された点です．is_pcdwith 関数は大まかなプリミティブに基づく衝突検出であるため，衝突する点は出力されません．

3 ロボットの定義

robot_sim パッケージにはロボットシミュレーション用のクラスや関数などを定義しています．このパッケージの中身はさらに以下のフォルダに分けられます．

フォルダ名	用途
_data_files	DDIKSolver で使用される KDTree と関節角データを保存するためのフォルダです.
_kinematics	直列リンクの運動学の定義や機構のメッシュの生成，機構の衝突検出，IK 計算などのためのクラスや関数はこのフォルダ内のファイルに実装されています．このフォルダ内で robot_sim の基盤となる機能の実装がなされており，他のフォルダの定義のほとんどは，このフォルダ内の実装に依存します．ハンド，マニピュレータおよびロボットのどれにおいても，一個あるいは複数個の JLChain によって構成されます.
end_effectors	ロボットのエンドエフェクタを格納するためのパッケージです．このパッケージには，ee_interface.py ファイルおよび gripper，multifinger，single_contact という 3 つのサブパッケージが含まれています．例えば，Robotiq 85 のような 2 本指のグリッパーは gripper サブパッケージに保存され，Barrett Hand のような多指のグリッパーは multifinger サブパッケージに保存されます．吸盤やドリル，ねじ締め器などの接触型ハンドの定義は single_contact サブパッケージに保存されています．ハンドの定義は，ee_interface.py 内で定義されている EEInterface インターフェースを継承する必要があります．さらに，gripper サブパッケージには GripperInterface も定義されており，これは EEInterface を基にして，グリッパー専用のインターフェースを追加しています．グリッパー型のエンドエフェクタの定義は，GripperInterface を継承する必要があります.
manipulators	マニピュレータの定義を格納するためのパッケージです．このパッケージ内では，各種または各タイプのマニピュレータに対応した独立のサブパッケージが設けられています．また，manipulator_interface.py というファイルが含まれており，その中で ManipulatorInterface インターフェースクラスが定義されています．すべてのマニピュレータの定義は，このクラスを継承する必要があります.
others	このパッケージは，いくつかの特殊な機械設備の定義を保存するために予約されています．現在は，一つのシールドマシンのみが含まれています.
robots	ロボットはこのフォルダ下のファイル内で定義されます．ロボットはマニピュレータとエンドエフェクタの集まりです．例えば，ur3_dual ロボットは二つの ur3 マニピュレータと各マニピュレータの手先に付けた二つの robotiq85 ハンドによって構成されます．このパッケージには，各ロボットのサブパッケージのほかに，robot_interface.py と single_arm_robot_interface.py の 2 つのファイルが含まれています．単一のマニピュレータとエンドエフェクタで構成される場合は，single_arm_robot_interface.py に定義されている SglArmRobotInterface インターフェースを継承する必要があります．より複雑な定義，例えば双腕などは，2 つ以上の単腕の集合である可能性があり，それらのクラスは robot_interface.py に定義されている RobotInterface インターフェースを継承するべきです.

各ファイルは，図 4 に示すような関係があります．ここで，二種類の矢印を用いて各クラスの関係を整理しました．黒い菱形の矢印が根本に付いたクラスのオブジェクトは矢先についたクラスのメンバ変数であることを示します．黒い三角の矢印が根本に付いたクラスは矢先についたクラスを継承としていることを示します．_kinematics の下に ik_xx.py，jl.py，model_generator.py, collision_checker.py など四つのファイルがあります．それぞれのファイルには Link, Joint, Anchor, JLChain，CollisionChecker, XXIKSolver クラスおよびこれに関わる関数が定義されています．図 4(a) の左部分の黒い菱形の矢印に示すように，JLChain は Joint や Anchor，Link 型のメンバ変数によって構成されます．図 4(a) の右部分と (b) の右上の黒い菱形の矢印に示すように，end_effectors，manipulators，robots フォルダ内のクラスは JLChain によって構成され，JLChain に強く依存します．また，ManipulatorInterface と RobotInterface は CollisionChecker 型のメンバー変数を持ちます.

続いて，end_effectors，manipulators，robots フォルダ内を詳しく覗いてみましょう．どちらのフォルダでもまず XXXXInterface というインターフェースのクラスを定義しています．特定のハンドやロボットのクラスは，このインターフェースのクラスを継承して定義します．例えば，end_effectors の直下には ee_interface.py ファイル

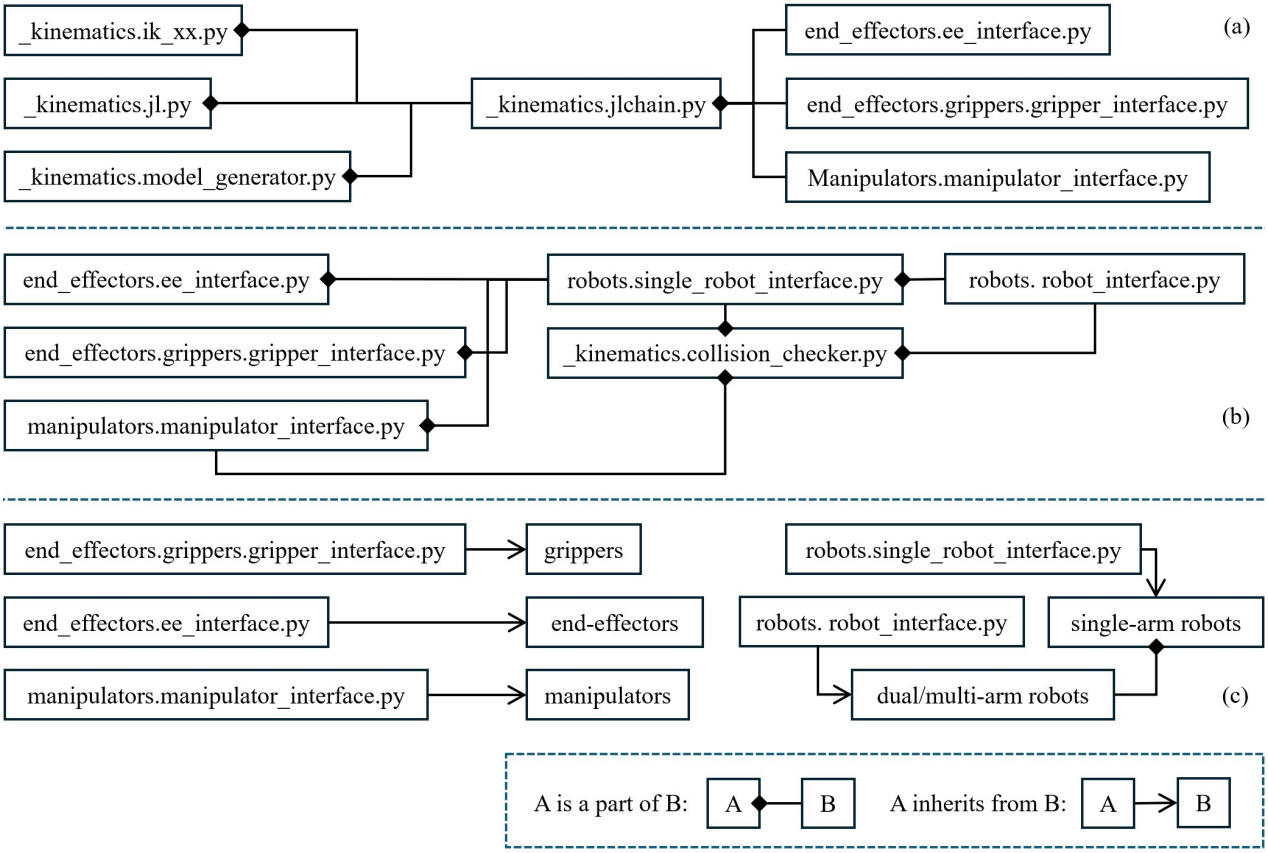

図 4: (a) JLChain クラスは Joint や Anchor，Link 型のメンバ変数によって構成されます．end_effectors，manipulators，robots フォルダ内のクラスは JLChain によって構成され，JLChain に強く依存します．(b, c) ハンドやマニピュレータフォルダの内部で XXXXInterface というインターフェースのクラスを定義して，特定のハンドやロボットのクラスは，このインターフェースのクラスの派生クラスです．

があります．ハンドの定義は，ee_interface.py 内で定義されている EEInterface インターフェースを継承する必要があります．さらに，end_effectors.grippers サブパッケージには GripperInterface も定義されており，これは EEInterface を基にして，グリッパー専用のインターフェースを追加しています．グリッパー型のエンドエフェクタの定義は，GripperInterface を継承する必要があります．end_effectors.grippers は gripper_interface.py 以外，robotiq85，robotiqhe，xarm_gripper，yumi_gripper などのサブパッケージを含めています．各サブパッケージ，例えば robotiq85 には robotiq85.py ファイルとそのハンドの CAD モデルを格納する meshes フォルダが含まれます．robotiq85.py ファイルに Robotiq85 のクラスが定義されており，このクラスは GripperInterface のクラスの派生クラスとなります．manipulators，robots フォルダも大体同じ仕組みですので，ここでの詳細な記述を割愛させます．

3.1 JLChain

3.1.1 JLChain の仕組み

これから，jlchain.py ファイルの具体的な内容について詳しく確認していきます．中身には JLChain クラスの定義しかありません．JLChain クラスの初期化関数はリスト 5.3-7 に示され，四つの引数を受け取ります．name は，この JLChain の名前を指します．pos と rotmat はそれぞれ，JLChain の根ノードの位置と回転を定義しています．n_dof は，JLChain の関節数を定義しています．

リスト 5.3-7: JLChain クラスの初期化関数

```
1 class JLChain(object):
2
```

```
3       def __init__(self,
4                    name="auto",
5                    pos=np.zeros(3),
6                    rotmat=np.eye(3),
7                    n_dof=6): # 関節の数
8           self.name = name
9           self.n_dof = n_dof
10          self.home = np.zeros(self.n_dof)
11           # jlchainの根ノード
12          self.anchor = rkjl.Anchor(name=f"{name}_anchor", pos=pos, rotmat=rotmat)
13           # リンクと関節を初期化
14          self.jnts = [rkjl.Joint(name=f"{name}_j{i}") for i in range(self.n_dof)]
15          self._jnt_ranges = self._get_jnt_ranges() # 可動範囲
16          self._flange_jnt_id = self.n_dof - 1 # 先端とした関節を指定
17          self._loc_flange_pos = np.zeros(3) # 先端のフランジが先端の関節における位置
18          self._loc_flange_rotmat = np.eye(3) # 先端のフランジが先端の関節における姿勢
19          self._gl_flange_pos = np.zeros(3) # global
20          self._gl_flange_rotmat = np.zeros(3)
21          self._is_finalized = False # 最終化関数未呼び出しのフラグ
22          self._ik_solver = None # 逆運動学計算関連
```

JLChain の初期化関数は，与えられたパラメータに基づいて JLChain のインスタンスを初期化します．10 行目では，初期のコンフィギュレーションを設定しており，このコンフィギュレーションはすべて 0 に設定されます．必要に応じて，後続のプログラムで home_conf 属性に値を再設定することで調整が可能です．

JLChain は 2 つの部分で構成されています．第 1 の部分は Anchor 型の根ノードであり，第 2 の部分はこの根ノードを基にした Joint のリストです．Link は明示的に定義されておらず，Joint の一部として存在します．Link は Joint に接続され，その位置と回転は属する Joint の局所座標系内で定義されています．Anchor は本質的に固定されたジョイントで，動作することはありません．Anchor には複数のフランジがあり，他の部品と接続することができます．これらのフランジの位置と姿勢は，Anchor の gl_flange_pose_list 属性を通じて取得できます．図では，Anchor 本体は台形で表現され，Anchor のフランジ面はそれぞれマゼンタ-イエロー-シアン（Magenta-Yellow-Cyan, MYC）色の座標系で表現されています．図 5 に示されているのは，4 つのフランジ面を持つ Anchor で，1 つの台形と 4 つの MYC 色の座標系で表現されています．点線は Anchor 本体から各フランジ面への相対的な関係を示しています．Joint と Link については第 4 章ですでに紹介しているため，ここでは詳細な説明を省略します．注意すべき点として，Link の cmodel 属性には，その Link の CollisionModel が保存されています．2.3~2.4 節で述べているように，各リンクの衝突を検出するためには，プリミティブとメッシュの 2 種類の衝突検出用の形状を設定する必要があります．JLChain の初期化関数では，12 行目と 14 行目でそれぞれ，この JLChain の Anchor と Joint のリストが定義されています．JLChain の Anchor ノードには通常，1 つのフランジ面しかありません．そのため，Anchor の定義時にはフランジの数を特別に設定することはありません．

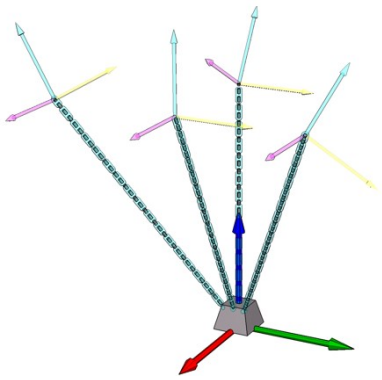

図 5: Anchor とそのフランジ．

初期化関数の 16〜18 行目では，JLChain の先端フランジの位置と回転が定義されています．カップリング，力覚センサー，あるいはエンドエフェクタなどがこの先端フランジに取り付けられます．16 行目では，このフランジが位置する関節の番号が定義されており，デフォルトでは最後の関節が指定されています．17 行目と 18 行目では，それぞれ関節座標系に対するフランジの位置と回転が定義されています．19 行目と 20 行目では，gl_flange_pos と gl_flange_rotmat 属性が定義されていますが，これは一時的な初期化であり，具体的な位置は_compute_gl_flange() 関数によって更新されます．

上記では，Anchor のフランジと JLChain のフランジについて言及しましたが，それぞれ Anchor の末端接続部と JLChain の末端接続部を指しています．フランジは専門用語であり，接続部分を意味し，通常は標準化されたインターフェースやナットを備えており，さまざまな末端装置を取り付けるために使用されます．Anchor のフランジは，後続の Joint を取り付けるためのものであり，JLChain のフランジは，カップリングや末端エフェクタなどのツールを取り付けるために使用されます．この違いを明確に区別してください．

JLChain クラスのインスタンスは初期化後，finalize 関数を呼び出して最終化を行う必要があります．WRS システムでは，この設計パターンを採用することで，JLChain の生成プロセスをより柔軟に制御しています．finalize 関数の内容は以下のリストの通りです．主に，再調整されたパラメータに基づいて関節の可動範囲属性である _jnt_ranges が更新され，go_home() 関数が呼び出されて FK（順運動学）パラメータが更新されます．また，逆運動学ソルバーの設定も行われます．

リスト 5.3-8: JLChain クラスの finalize 関数

```
    def finalize(self, ik_solver=None, identifier_str="test", **kwargs):
        self._jnt_ranges = self._get_jnt_ranges()
        self.go_home()
        self._is_finalized = True
        if ik_solver == 'd':
            self._ik_solver = rkd.DDIKSolver(self, identifier_str=identifier_str)
        elif ik_solver == 'n':
            self._ik_solver = rkn.NumIKSolver(self)
        elif ik_solver == 'o':
            self._ik_solver = rko.OptIKSolver(self)
        elif ik_solver == 'a': # analytical ik, user defined
            self._ik_solver = None
```

go_home 関数をさらに追跡すると，最終的に self.home 属性が self.fk() 関数に渡され，JLChain 内部の各種パラメータが更新されていることがわかります．以下に示すのは self.fk() 関数の具体的な内容です．go_home 関数がこの関数を呼び出す際，update パラメータが True に設定されているため，コード内で update が False となる部分は省略しています．この呼び出しでは，fk 関数が Anchor の根ノードから始まり，順に各 Joint を更新しています（5 行 ~11 行まで）．この関数は，Anchor と Joint の 2 つの部分の関係を統合しています．また，JLChain のグローバル先端フランジの位置と回転属性である gl_flange_pos と gl_flange_rotmat も，この呼び出しの中で更新されています（12 行）．

リスト 5.3-9: JLChain クラスの fk 関数

```
    def fk(self, jnt_values, toggle_jacobian=False, update=False):
        if not update:
            ... # go_home関数とは関係なく，省略します.
        else:
            pos = self.anchor.gl_flange_pose_list[0][0]
            rotmat = self.anchor.gl_flange_pose_list[0][1]
            for i in range(self.n_dof):
                motion_value = jnt_values[i]
                self.jnts[i].update_globals(pos=pos, rotmat=rotmat, motion_value=motion_value)
                pos = self.jnts[i].gl_pos_q
                rotmat = self.jnts[i].gl_rotmat_q
            self._gl_flange_pos, self._gl_flange_rotmat = self._compute_gl_flange()
            if toggle_jacobian:
                ... # go_home関数とは関係なく，省略します.
            else:
                return self._gl_flange_pos, self._gl_flange_rotmat
```

3.1.2 JLChain インスタンスの定義と運動学・逆運動学の計算

リスト 5 では，JLChain の定義と逆運動学（IK）の具体的な実例が示されています．コードの第 4 行では，JLChain が初期化され，関節数が 4 に設定されています．初期化後，この JLChain の関節長，回転軸，動作範囲などにはデフォルト値が割り当てられます．具体的な数値については，jl.py ファイル内の各クラスの初期化関数を参照することができます．第 5 行から第 24 行までの間では，コードは各関節のパラメータを順に調整し，JLChain のフランジ面の位置をリセットしています．これらの変更は，第 25 行で finalize 関数を呼び出すことによって最終化されます．この finalize 関数では，DDIKSolver を逆運動学の解法として指定しています．

リスト 5.3-10: JLChain クラスのインスタンス化

```
... # importや__name__の判断を省略
    base = wd.World(cam_pos=[1.5, .2, .9], lookat_pos=[0, 0, 0.3])
```

```
3     mgm.gen_frame().attach_to(base)
4     jlc = rkjlc.JLChain(n_dof=4) # 四自由度のJLChainを定義
5     # Linkのモデルの定義，コメントを外すとgen_meshの結果にメッシュモデルの表示ができるようになります．そうでない場合，メッシュモデルは空になります．
6     # jlc.anchor.lnk_list[0].cmodel = mcm.gen_box(np.array([.05, .05, .1]))
7     # jlc.anchor.lnk_list[0].loc_pos = np.array([0, 0, 0.05])
8     jlc.jnts[0].loc_pos = np.array([0, 0, .1])
9     jlc.jnts[0].loc_motion_ax = np.array([1, 0, 0])
10    # jlc.jnts[0].lnk.cmodel= mcm.gen_box(np.array([.05, .05, .1])) # 6〜7行と同様
11    # jlc.jnts[0].lnk.loc_pos = np.array([0, 0, 0.05])
12    jlc.jnts[1].loc_pos = np.array([0, 0, .1])
13    jlc.jnts[1].loc_motion_ax = np.array([0, 1, 0])
14    # jlc.jnts[1].lnk.cmodel= mcm.gen_box(np.array([.05, .05, .1])) # 6〜7行と同様
15    # jlc.jnts[1].lnk.loc_pos = np.array([0, .0, 0.05])
16    jlc.jnts[2].change_type(type=rkjl.rkc.JntType.PRISMATIC, motion_range=np.array([-.1, .1]))
17    jlc.jnts[2].loc_pos = np.array([0, 0, .1])
18    jlc.jnts[2].loc_motion_ax = np.array([0, 0, 1])
19    jlc.jnts[3].loc_pos = np.array([0, 0, .1])
20    jlc.jnts[3].loc_motion_ax = np.array([0, 0, 1])
21    # jlc.jnts[3].lnk.cmodel= mcm.gen_box(np.array([.05, .05, .1])) # 6〜7行と同様
22    # jlc.jnts[3].lnk.loc_pos = np.array([0, 0, 0.05])
23    jlc._loc_flange_pos = np.array([0, 0, .1])
24    jlc.finalize(ik_solver='d') # DDIKSolverを利用
25    # 解を確保するため，正運動学より目標位置と回転を設定する
26    jnt_values = np.array([-np.pi/6, np.pi/3, .05, -np.pi/4])
27    goal_pos, goal_rotmat = jlc.fk(jnt_values=jnt_values, update=False) # 更新しないこと
28    mgm.gen_frame(pos=goal_pos, rotmat=goal_rotmat).attach_to(base) # 目標を画面に表示
29    jnt_values = jlc.ik(tgt_pos=goal_pos, tgt_rotmat=goal_rotmat) # 逆運動学の計算
30    jlc.goto_given_conf(jnt_values=jnt_values) # 更新（fk(update=True)と等価）
31    jlc.gen_stickmodel(toggle_flange_frame=True, toggle_jnt_frames=False).attach_to(base)
32    jlc.gen_meshmodel(alpha=.5).attach_to(base)
33    base.run()
```

コードの第 26 行から第 30 行では，逆運動学の具体的な解法の例が示されています．ここで，逆運動学に解があることを保証するための小技が使われています．まず，第 27 行で各関節の関節角ベクトルを設定します．そして，第 28 行では，その関節角ベクトルに基づいて正運動学を解き，JLChain のフランジ面の姿勢を取得しています．この際，正運動学の計算では update を False に設定しており，内部の各関節の gl_pos や gl_rotmat といったパラメータを更新せず，先端フランジ面の姿勢のみを取得し，その値を返しています．正運動学の結果と関節角ベクトルは対応しているため，これを逆運動学の目標姿勢として設定すれば，必ず解が存在します．第 29 行では，この結果を画面に表示し，図 6(a) の goalpose 座標系として視覚化します．最後に，第 30 行でこの目標姿勢に対して逆運動学の解を求めています．

コードの第 31 行では，逆運動学の解を使用して JLChain の構型（configuration）を更新しています．ここで呼び出されている goto_given_conf 関数は，実質的に update が True に設定された状態の fk 関数を呼び出すものです．WRS システムでは，区別を明確にするため，JLChain の構型を更新する際には goto_given_conf 関数を使用し，フランジ面の位置や回転を取得する際には fk 関数を使用することを推奨しています．fk 関数はデフォルトで update が False になっており，各関節のパラメータを更新せずに位置や姿勢のみを取得します．

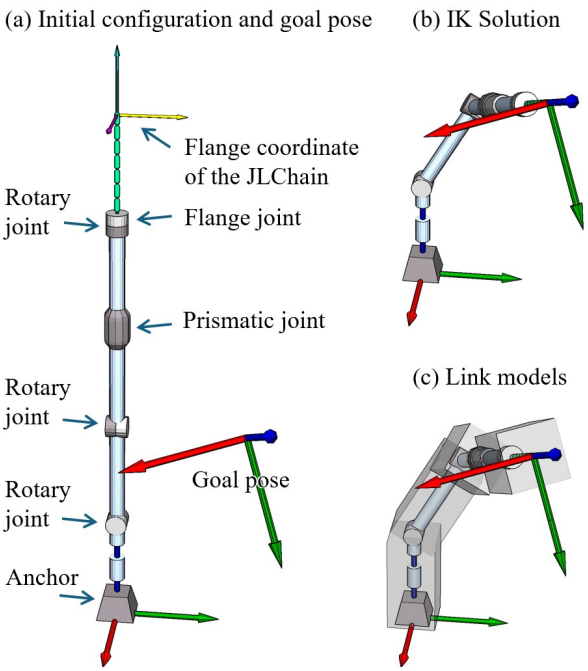

図 6: (a) 初期コンフィギュレーション．(b)IK の解．(c)Link のメッシュモデルを表示．

第 32 行と第 33 行では，それぞれ gen_stickmodel と gen_meshmodel を呼び出して，JLChain のスティックモデルとメッシュモデルを生成しています．もし JLChain の各リンクに cmodel 属性が設定されていない場合，gen_meshmodel は空のモデルを生成します．上記のコードにおいては，6, 7, 10, 11, 14, 15, 21, 22 行のコメントを

解除することで，各関節に cmodel 属性を設定し，空でないメッシュモデルを生成することができます．図 6(b) および (c) には，それぞれスティックモデルと，スティックモデルにメッシュモデルを重ねた状態が表示されています．

3.2 Grippers と Manipulators

3.2.1 Grippers

Grippers と manipulators は一個あるいは複数個の JLChain のインスタンスによって構成されますので，この節でまとめて説明します．まず，grippers について紹介します．grippers は end_effectors/grippers の子パッケージ内で定義されます．grippers 以外もいろいろ他の end-effectors がありまして，それらの end-effectors の定義は end_effectors の他のフォルダに格納されています．本節は grippers とその下の小パッケージだけに注目します．例えば，grippers.robotiqhe という子パッケージには Robotiq 社の HandE グリッパの定義やメッシュなどが格納されています．グリッパの定義は.py ファイルとして記述され，メッシュの CAD モデルは meshes フォルダに保存されています．リスト 5.3-11 のプログラムは grippers.robotiqhe のハンドを画面に表示する実例です．

リスト 5.3-11: Rrobotiq 社の HandE グリッパを利用する例

```
1  import numpy as np
2  import basis.robot_math as rm
3  import visualization.panda.world as wd
4  import robot_sim.end_effectors.gripper.robotiqhe.robotiqhe as rtq_he
5  import modeling.geometric_model as mgm
6
7  if __name__ == "__main__":
8      base = wd.World(cam_pos=[1, 1, 1], lookat_pos=[0, 0, 0])
9      mgm.gen_frame(ax_length=.2).attach_to(base)
10     grpr = rtq_he.RobotiqHE()
11     grpr.change_jaw_width(.05) # 開き幅を5センチに設定
12     # メッシュモデルを生成し画面に表示します
13     grpr.gen_meshmodel(rgb=np.array([.3,.3,.3]), alpha=.3).attach_to(base)
14     # 棒モデルも生成し画面に表示します．同時に，TCP座標と各関節の座標も表示します
15     grpr.gen_stickmodel(toggle_tcp_frame=True, toggle_jnt_frames=True).attach_to(base)
16     # 位置を変更します
17     grpr.fix_to(pos=np.array([-.1, .2, 0]), rotmat=rm.rotmat_from_axangle([1, 0, 0], .05))
18     # 変更した位置で改めてメッシュモデルを生成し画面に表示します
19     grpr.gen_meshmodel(toggle_cdmesh=True).attach_to(base) # mesh衝突モデルも表示します
20     # もう一度位置を変更します
21     grpr.fix_to(pos=np.array([.1, -.2, 0]), rotmat=rm.rotmat_from_axangle([1, 0, 0], .05))
22     grpr.gen_meshmodel(toggle_cdprim=True).attach_to(base) # プリミティブ衝突モデルも表示します
23     base.run()
```

リスト 5.3-11 の実行結果を図 7 に示します．真ん中のグリッパは 13 行目と 15 行目で書き出したもので，半透明のメッシュモデル，棒モデルの両方が表示されています．MYC 色の座標系はグリッパの把持中心の座標系，それ以外の RGB 色の座標系は各関節の座標系を指します．右側のグリッパは 17 行目と 19 行目によって表示したもので，左側のグリッパは 21 行目と 22 行目によって表示したものです．それぞれの mesh 衝突モデルとプリミティブ衝突モデルが表示されています．mesh 衝突モデルの本質は，各モデルの三角形ポリゴンによる面モデルです．プリミティブ衝突モデルの本質は，各 Link の cmodel に保存されている CollisionModel 型のインスタンスです．

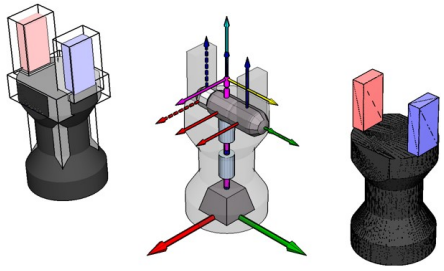

図 7: HandE グリッパ．左から右：プリミティブ衝突モデル；JLChain とその座標系，把持中心の座標系；mesh 衝突モデル．

続いて，上記のコードの 4 行目でインポートしているファイル（robot_sim.end_effectors.grippers.robotiqhe.robotiqhe）を見てみましょう．ファイル内では RobotiqHE のクラスを定義しています．RobotiqHE クラスの初期化関数はリスト 5.3-12 に示します．RobotiqHE クラスの定義は独立なものではなく，gp.GripperInterface から継承したものと分かります．WRS ではさまざまなグリッパを定義するため，グリッパごとに共通の変数や関数を繰り返し定義するのは手間がかかります．そこで，共

通の変数や関数をまとめたインターフェースというクラスを作成し，各グリッパのクラスはこのインターフェースを継承して作成する方法を採用しています．インターフェースは grippers の下の gripper_interface.py ファイルに GripperInterface クラスとして定義されています．GripperInterface クラスはさらに EEInterface クラスを継承しており，これらのインターフェース内では name や pos, rotmat など Gripper の共通のメンバ変数や is_mesh_collided, gen_meshmodel, grip_at_by_xxx などの Gripper の共通のメンバ関数をあらかじめ用意しています．具体的な Gripper，例えば RobotiqHE を定義する際には，GripperInterface クラスを継承します．継承すると，あらかじめ用意されたメンバ変数やメンバ関数が再定義せずにそのまま使えます．また，必要に応じて，継承したメンバ変数の値の変更やメンバ関数のオーバーロードも可能です．

リスト 5.3-12: Robotiq 社の HandE グリッパ用のクラスの初期化関数

```
1  class RobotiqHE(gpi.GripperInterface): # GripperInterfaceクラスから継承します
2
3      def __init__(self,
4                   pos=np.zeros(3),
5                   rotmat=np.eye(3),
6                   coupling_offset_pos=np.zeros(3),
7                   coupling_offset_rotmat=np.eye(3),
8                   cdmesh_type=mcm.mc.CDMType.DEFAULT,
9                   name='rtq_he'):
10         super().__init__(pos=pos, rotmat=rotmat, cdmesh_type=cdmesh_type, name=name)
11         current_file_dir = os.path.dirname(__file__)
12         # 親クラスは必ずカップリングがあることと想定していますから，カップリングを設定する．
13         self.coupling.loc_flange_pose_list[0] = (coupling_offset_pos, coupling_offset_rotmat)
14         # カップリングがある場合（位置はゼロではない），円柱としてモデリングする．
15         if np.any(self.coupling.loc_flange_pose_list[0][0]):
16             self.coupling.lnk_list[0].cmodel = mcm.gen_stick(spos=np.zeros(3), epos=self.
                   coupling.loc_flange_pose_list[0][0], type="rect", radius=0.035, rgb=np.array
                   ([.35, .35, .35]), alpha=1, n_sec=24)
17         self.jaw_range = np.array([.0, .05]) # 開き幅
18         self.jlc = rkjlc.JLChain(pos=self.coupling.gl_flange_pose_list[0][0], rotmat=self.
             coupling.gl_flange_pose_list[0][1], n_dof=2, name=name) # JLChain
19         self.jlc.anchor.lnk_list[0].loc_rotmat = rm.rotmat_from_euler(0, 0, np.pi / 2)
20         self.jlc.anchor.lnk_list[0].cmodel = mcm.CollisionModel(os.path.join(current_file_dir,
              "meshes", "base.stl"), cdmesh_type=self.cdmesh_type, cdprim_type=mcm.mc.CDPType.
             USER_DEFINED, userdef_cdprim_fn=self._base_cdprim)
21         self.jlc.anchor.lnk_list[0].cmodel.rgba = np.array([.2, .2, .2, 1])
22         # 第一関節（左指，y軸の正方向に展開）
23         self.jlc.jnts[0].change_type(rkjlc.rkc.JntType.PRISMATIC, motion_range=np.array([0,
             self.jaw_range[1] / 2]))
24         self.jlc.jnts[0].loc_pos = np.array([.0, .0, 0.11])
25         # self.jlc.jnts[0].loc_rotmat = rm.rotmat_from_euler(0, 0, -np.pi / 2)
26         self.jlc.jnts[0].loc_motion_ax = rm.bc.y_ax
27         self.jlc.jnts[0].lnk.cmodel = mcm.CollisionModel(initor=os.path.join(current_file_dir,
              "meshes", "finger1.stl"), cdmesh_type=self.cdmesh_type, cdprim_type=mcm.mc.CDPType.
             AABB, ex_radius=.005)
28         self.jlc.jnts[0].lnk.loc_rotmat = rm.rotmat_from_euler(0, 0, -np.pi / 2)
29         self.jlc.jnts[0].lnk.cmodel.rgba = np.array([.5, .5, 1, 1])
30         # 第二関節（右指，y軸の負方向に展開）
31         self.jlc.jnts[1].change_type(rkjlc.rkc.JntType.PRISMATIC, motion_range=np.array([0.0,
             self.jaw_range[1]]))
32         self.jlc.jnts[1].loc_pos = np.array([.0, .0, .0])
33         self.jlc.jnts[1].loc_motion_ax = rm.bc.y_ax
34         self.jlc.jnts[1].lnk.cmodel = mcm.CollisionModel(initor=os.path.join(current_file_dir,
              "meshes", "finger2.stl"), cdmesh_type=self.cdmesh_type, cdprim_type=mcm.mc.CDPType.
             AABB, ex_radius=.005)
35         self.jlc.jnts[1].lnk.loc_rotmat = rm.rotmat_from_euler(0, 0, -np.pi / 2)
36         self.jlc.jnts[1].lnk.cmodel.rgba = np.array([1, .5, .5, 1])
37         self.jlc.finalize() # 最終化，IKSolverはディフォルトな値にする（利用しない）
38         # 把持中心
39         self.loc_acting_center_pos = np.array([0, 0, .14]) + coupling_offset_pos
40         # 精密衝突検出時に使われるlinkのメッシュモデルを指定
41         self.cdmesh_elements = (self.jlc.anchor.lnk_list[0], self.jlc.jnts[0].lnk, self.jlc.
             jnts[1].lnk)
```

RobotiqHE クラスの初期化関数の第 13 行では，メンバ変数 coupling が設定されています．coupling は，EEInterface が強制的に要求するメンバ変数であり，デフォルトでは coupling が存在しないため，そのローカル位置と回転はどちらも 0 となります．この関数では，単一の JLChain 型のメンバ変数が定義されています．この JLChain には 2 つ

の直動関節が含まれており，それぞれ左指と右指を駆動します．これらの指のパラメータ，モデル，運動軸，運動範囲などは，コードの第 32 行から第 54 行で詳細に定義されています．実装においては，必要に応じてこれらを設定できます．コードの第 55 行では，JLChain が最終化されています．第 57 行では，グリッパのルート座標系に対する把持中心の位置が設定されています．回転はここでは設定されておらず，デフォルト値（単位行列）が使用されます．第 59 行では，メッシュ衝突検出時に使用されるメッシュモデルのリストが設定されています．

ここで特に触れておくべき点は，第 20 行で定義されている手のひら部分のプリミティブ衝突モデルについてです．この部分では，cdprim_type を USER_DEFINED に設定しています．この設定により，システムは後に定義されている userdef_cdprim_fn で指定された関数を参照して，衝突干渉チェック用のプリミティブを生成します．指定された関数は自由に定義でき，複数の包囲ボックスを組み合わせて複雑な形状をシミュレートすることが可能です．ここで指定されている関数は，メンバ関数である_base_cdprim であり，その実装は以下の通りです．この関数では，2 つの箱型モデルを使用して，手のひらの複雑な形状を表現しています．この関数は静的関数であり，具体的なインスタンスにはバインドされません．また，この関数内では，さらに低レベルの衝突ライブラリに関する知識が関与していますが，ここでは詳細な説明は行いません．必要に応じて，自身で設計する際には，この例を参考にしてコードを作成してください．

リスト 5.3-13: _base_cdprim メンバー関数

```
    @staticmethod
    def _base_cdprim(ex_radius=None):
        pdcnd = CollisionNode("rtq_he_base")
        collision_primitive_c0 = CollisionBox(Point3(0.0, 0.0, 0.1), x=.032 + ex_radius, y
          =.029 + ex_radius, z=.01 + ex_radius)
        pdcnd.addSolid(collision_primitive_c0)
        collision_primitive_c1 = CollisionBox(Point3(0.0, 0.0, 0.05), x=.02 + ex_radius, y=.02
           + ex_radius, z=.03 + ex_radius)
        pdcnd.addSolid(collision_primitive_c1)
        cdprim = NodePath("user_defined")
        cdprim.attachNewNode(pdcnd)
        return cdprim
```

RobotiqHE の他のメンバ関数の実装についても説明します．インターフェースからの継承と実装には 2 つのケースがあります．1 つ目は，インターフェースクラスで実装されていない関数です．例えば，fix_to，gen_meshmodel，change_jaw_width など，この種の関数には統一されたコードがなく，インターフェースクラスで実装されずに NotImplementedError が発生します．そのため，サブクラスはこれらの関数を強制的に実装する必要があります．RobotiqHE の定義では，これらの関数がオーバーロードされ，実装されています．2 つ目は，再定義が必要な関数です．例えば，hold，release，grip_at_by_xxx などが該当します．しかし，RobotiqHE には特別な要件がないため，これらの関数は再定義されていません．

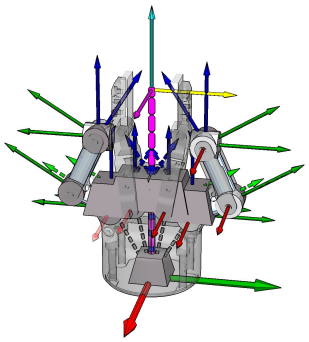

図 8: Robotiq85.

補足として，RobotiqHE に加えて，より複雑なグリッパーである Robotiq85 のクラス定義を紹介いたします．リスト 5.3-14 は，このクラスの初期化関数を示しています．Robotiq85 グリッパーの各指は，アンダードライブの五連杆機構に基づいて構成されています．この初期化関数では，2 つの指の五連杆機構をシミュレーションするために，4 つの JLChain を使用しています．これら 4 つの JLChain は，Anchor 型の手のひらの 4 つのフランジ面に取り付けられています．それらの間で，複数の関節角度を連動させることで，五連杆の動作をシミュレーションしています．リスト 5.3-15 にある change_jaw_width 関数は，関節間の連動関係の設定方法を詳しく示しています．図 8 は，Robotiq 85 グリッパーの JLChain 構造とその各種座標系の定義を示しています．この図では，1 つの Anchor に取り付けられた 4 つのフランジ面上に配置された 4 つの JLChain が明確に区別されています．

リスト 5.3-14: Robotiq 85 グリッパの初期化関数

```
... # importを省略
class Robotiq85(gi.GripperInterface):

```

```
    def __init__(self,
                 pos=np.zeros(3),
                 rotmat=np.eye(3),
                 coupling_offset_pos=np.zeros(3),
                 coupling_offset_rotmat=np.eye(3),
                 cdmesh_type=mcm.mc.CDMType.DEFAULT,
                 name='robotiq85'):
        super().__init__(pos=pos, rotmat=rotmat, cdmesh_type=cdmesh_type, name=name)
        current_file_dir = os.path.dirname(__file__)
        # カップリング
        self.coupling.loc_flange_pose_list[0] = (coupling_offset_pos, coupling_offset_rotmat)
        if np.any(self.coupling.loc_flange_pose_list[0][0]):
            self.coupling.lnk_list[0].cmodel = mcm.gen_stick(spos=np.zeros(3), epos=self.
              coupling.loc_flange_pose_list[0][0], type="rect", radius=0.035, rgb=np.array
              ([.35, .35, .35]), alpha=1, n_sec=24)
        # 開き幅
        self.jaw_range = np.array([.0, .085])
        # 手平を四つのフランジを持つAnchorとして定義する
        self.palm = rkjlc.rkjl.Anchor(name=name + "_palm", pos=self.coupling.
          gl_flange_pose_list[0][0], rotmat=self.coupling.gl_flange_pose_list[0][1], n_flange
          =4)
        self.palm.loc_flange_pose_list[0] = [np.array([0, .0306011, .054904]), np.eye(3)]
        self.palm.loc_flange_pose_list[1] = [np.array([0, .0127, .06142]), np.eye(3)]
        self.palm.loc_flange_pose_list[2] = [np.array([0, -.0306011, .054904]), np.eye(3)]
        self.palm.loc_flange_pose_list[3] = [np.array([0, -.0127, .06142]), np.eye(3)]
        self.palm.lnk_list[0].name = name + "_palm_lnk"
        self.palm.lnk_list[0].cmodel = mcm.CollisionModel(initor=os.path.join(current_file_dir
          , "meshes", "robotiq_arg2f_85_base_link.stl"), cdmesh_type=self.cdmesh_type)
        self.palm.lnk_list[0].cmodel.rgba = rm.bc.dim_gray
        # ======= 左指 ======= #
        # 左指の外側のJLChain
        self.lft_outer_jlc = rkjlc.JLChain(pos=self.palm.gl_flange_pose_list[0][0], rotmat=
          self.palm.gl_flange_pose_list[0][1], n_dof=4, name=name + "_left_outer")
        ... # 左指の外側のJLChainの詳細的な定義を省略
        # 左指の内側のJLChain
        self.lft_inner_jlc = rkjlc.JLChain(pos=self.palm.gl_flange_pose_list[1][0], rotmat=
          self.palm.gl_flange_pose_list[1][1], n_dof=1, name=name + "_left_inner")
        ... # 左指の内側のJLChainの詳細的な定義を省略
        # ======= 右指 ======== #
        # 右指の外側のJLChain
        self.rgt_outer_jlc = rkjlc.JLChain(pos=self.palm.gl_flange_pose_list[2][0], rotmat=
          self.palm.gl_flange_pose_list[2][1], n_dof=4, name=name + "_right_outer")
        ... # 右指の外側のJLChainの詳細的な定義を省略
        # 右指の内側のJLChain
        self.rgt_inner_jlc = rkjlc.JLChain(pos=self.palm.gl_flange_pose_list[3][0], rotmat=
          self.palm.gl_flange_pose_list[3][1], n_dof=1, name=name + "_right_inner")
        ... # 右指の内側のJLChainの詳細的な定義を省略
        # 各JLChainの最終化
        self.lft_outer_jlc.finalize()
        self.lft_inner_jlc.finalize()
        self.rgt_outer_jlc.finalize()
        self.rgt_inner_jlc.finalize()
        # 把持中心
        self.loc_acting_center_pos = np.array([0, 0, .15])
        ... # 精密干渉チェック用Linkリストの設定を省略
```

リスト 5.3-15: Robotiq85 クラスの change_jaw_width メンバー関数

```
    def change_jaw_width(self, jaw_width):
        if jaw_width > 0.085:
            raise ValueError("ee_values must be 0mm~85mm!")
        angle = math.asin((self.jaw_range[1] / 2.0 + .0064 - .0306011) / 0.055) - math.asin(
            (jaw_width / 2.0 + .0064 - .0306011) / 0.055)
        if angle < 0:
            angle = 0
        # 同一のangle値が4つのJLChainを連動し，五連杆機構の運動をシミュレート
        self.lft_outer_jlc.goto_given_conf(jnt_values=np.array([angle, 0.0, -angle, 0.0]))
        self.lft_inner_jlc.goto_given_conf(jnt_values=np.array([angle]))
        self.rgt_outer_jlc.goto_given_conf(jnt_values=np.array([angle, 0.0, -angle, 0.0]))
        self.rgt_inner_jlc.goto_given_conf(jnt_values=np.array([angle]))
```

3.2.2 Manipulators

manipulators サブパッケージと end-effectors サブパッケージは同じ階層にあり，grippers よりも一段上の階層に位置しています．ただし，manipulators サブパッケージの構造は grippers サブパッケージの構造に似ており，その内部にはさまざまなサブサブパッケージが含まれていて，それぞれに違うマニピュレータが定義されています．各マニピュレータサブサブパッケージには，マニピュレータのクラスを定義する.py ファイルと，関節の CAD モデルを保存する meshes フォルダーが含まれています．各マニピュレータクラスは manipulators サブパッケージの直下の manipulator_interface.py に定義された ManipulatorInterface を継承して実装されます．ManipulatorInterface には予め一個の JLChain 型のメンバ変数を用意しています．したがって，各マニピュレータも一つの JLChain 型のメンバ変数を持っています．例えば，IRB14050 マニピュレータの初期化関数をリスト 5.3-16 に示します．その中では，ManipulatorInterface から継承された JLChain タイプの jlc メンバー変数のパラメータを調整し，最終化を行っています．jlc のパラメータを調整する以外に，この初期化関数では，マニピュレータの TCP（Tool Center Point）の JLChain の先端フランジ面座標系に対する局所的な位置と回転も定義しています．また，enable_cc が True の場合には，プリミティブ衝突検出用の CollisionChecker を設定しています．

リスト 5.3-16: ABB 社の IRB14050 マニピュレータクラスの初期化関数

```
1  class IRB14050(mi.ManipulatorInterface): # ManipulatorInterfaceから継承
2
3      def __init__(self, pos=np.zeros(3), rotmat=np.eye(3), name='irb14050', enable_cc=False):
4          super().__init__(pos=pos, rotmat=rotmat, home_conf=np.zeros(7), name=name, enable_cc=
             enable_cc)
5          current_file_dir = os.path.dirname(__file__)
6          # 第一関節の定義
7          self.jlc.jnts[0].loc_pos = np.array([.0, .0, .0])
8          self.jlc.jnts[0].loc_rotmat = rm.rotmat_from_euler(0.0, 0.0, np.pi)
9          self.jlc.jnts[0].loc_motion_ax = np.array([0, 0, 1])
10         self.jlc.jnts[0].motion_range = np.array([-2.94087978961, 2.94087978961])
11         self.jlc.jnts[0].lnk.cmodel = mcm.CollisionModel(os.path.join(current_file_dir, "
             meshes", "link_1.stl"))
12         self.jlc.jnts[0].lnk.cmodel.rgba = rm.bc.hug_gray
13         ... # 第二関節から第七関節までは省略
14         # フランジの局所回転を変更（実際のロボットに参考した結果）
15         self.jlc._loc_flange_rotmat = rm.rotmat_from_euler(0,0,np.pi/2)
16         # 最終化，DDIKSolverに設定
17         self.jlc.finalize(ik_solver='d', identifier_str=name)
18         # TCP（Tool Center Point），JLChainの先端フランジの座標系に定義，End-Effectorが装着さ
             れるときEnd-Effectorに合わせて更新
19         self.loc_tcp_pos = np.array([0, 0, .007])
20         self.loc_tcp_rotmat = np.eye(3)
21         # プリミティブ衝突検出用のCollisionCheckerの設定
22         if self.cc is not None:
23             self.setup_cc()
```

リスト 5.3-17 では，self.setup_cc() メンバー関数の具体的な内容がさらに詳しく示されています．この関数では，各関節に取り付けられた Link を CollisionChecker タイプのメンバー変数 self.cc の CCElement として追加します．これらの CCElement は外部障害物との衝突検出に使用されます．また，この関数は self.cc の set_cdpair_by_ids 関数を呼び出して，追加された CCElement 内部のメンバー間で検出が必要な衝突を設定しています．

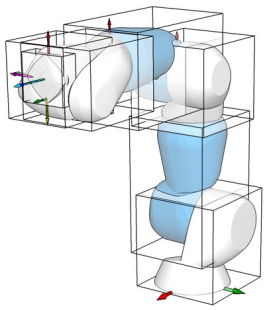

図 9: IRB14050.

リスト 5.3-18 は，IRB14050 マニピュレータのインスタンスを定義し，表示する例です．ここで，enable_cc パラメータは True に設定されており，マニピュレータは衝突検出機能を持っています．コードの 7 行目では，is_collided メンバー関数が呼び出されて衝突検出が行われます．引数は空で，外部障害物との検出は行わず，自身の関節間の衝突検出のみを実施しています．この時，マニピュレータはデフォルトの初期姿勢にあり，衝突は発生しないため，印刷結果は True となります．コード実行後に表示される画面は，図 9 のようになります．メッシュモデルを生成する際に toggle_cdprim を True に設定したため．ロボットアームの各リンクの cdprim が表示されています．それらはすべて AABB タイプです．特定の IK ソルバーが指定されていないため，システムは逆運動学の解法に DDIKSolver を使用します．そのため，初回のプログラム実行時にはデータ構築の進捗バーが表示されます．このロボットアームは 7 自由度を持っ

ているため，サンプリングに時間がかかることがあります．ロボットアームの ik 関数を呼び出す際には，システムがこれらのデータを使用して初期の反復値を見つけ，ニュートン-ラフソン法で逆運動学を解きます．冗長性のあるロボットアームのマルチタスク解法を行う場合は，ヤコビ行列の零空間で計算できるように，独自の IK ソルバーを実装することが可能です．

リスト 5.3-17: setup_cc 関数

```
    def setup_cc(self):
        l0 = self.cc.add_cce(self.jlc.jnts[0].lnk)
        l1 = self.cc.add_cce(self.jlc.jnts[1].lnk)
        l2 = self.cc.add_cce(self.jlc.jnts[2].lnk)
        l3 = self.cc.add_cce(self.jlc.jnts[3].lnk)
        l4 = self.cc.add_cce(self.jlc.jnts[4].lnk)
        l5 = self.cc.add_cce(self.jlc.jnts[5].lnk)
        l6 = self.cc.add_cce(self.jlc.jnts[6].lnk)
        from_list = [l0]
        into_list = [l4, l5]
        self.cc.set_cdpair_by_ids(from_list, into_list)
```

リスト 5.3-18: IRB14050 マニピュレータの描画

```
... # importや__name__の判断を省略
    base = wd.World(cam_pos=[1.5, 1, 0.7], lookat_pos=[0, 0, .2])
    mgm.gen_frame().attach_to(base)
    arm = irb.IRB14050(enable_cc=True)
    arm.gen_stickmodel(toggle_jnt_frames=True, toggle_tcp_frame=True).attach_to(base)
    arm.gen_meshmodel(toggle_cdprim=True, alpha=1).attach_to(base)
    print(arm.is_collided())
    base.run()
```

3.3 Robots パッケージ

3.3.1 単腕ロボット

次に robots パッケージの内容について紹介します．robots パッケージには主に 2 つのインターフェースがあり，それぞれ robot_interface.py に保存されている RobotInterface と，single_arm_robot_interface.py に保存されている SglArmRobotInterface です．RobotInterface は最も基本的な定義であり，位置（pos），回転行列（rotmat），ホームポジション（home_conf）といった基本的なメンバ変数や，衝突判定（is_collided），メッシュモデル生成（gen_meshmodel）といった基本メソッドを含んでいます．SglArmRobotInterface は RobotInterface を継承しており，RobotInterface に加えて，マニピュレータ（manipulator）とエンドエフェクタ（end_effector）というメンバ変数が追加されています．これらを使って，単腕ロボットの唯一のアームと末端エフェクタにアクセスできます．本節ではまず，Yumi ロボットの単腕を例に挙げ，単腕ロボットの実装について説明します．Yumi ロボットの単腕の定義は robots.yumi.yumi_single_arm.py に定義され，そのコードは以下の通りです．

リスト 5.3-19: Yumi ロボットの単腕の定義

```
import numpy as np
import robot_sim.manipulators.irb14050.irb14050 as irb14050
import robot_sim.end_effectors.gripper.yumi_gripper.yumi_gripper as yg
import robot_sim.robots.single_arm_robot_interface as ri


class YumiSglArm(ri.SglArmRobotInterface):

    def __init__(self, pos=np.zeros(3), rotmat=np.eye(3), name="sglarm_yumi", enable_cc=True):
        super().__init__(pos=pos, rotmat=rotmat, name=name, enable_cc=enable_cc)
        # マニピュレータはIRB14050
        self.manipulator = irb14050.IRB14050(pos=self.pos, rotmat=self.rotmat,
                                              name="irb14050_" + name, enable_cc=False)
        # エンドエフェクタはIRB14050
        self.end_effector = yg.YumiGripper(pos=self.manipulator.gl_flange_pos,
                                           rotmat=self.manipulator.gl_flange_rotmat, name="yg_
                                             " + name)
```

```
        # マニピュレータのTCPをエンドエフェクタの把持中心に更新
        self.manipulator.loc_tcp_pos = self.end_effector.loc_acting_center_pos
        self.manipulator.loc_tcp_rotmat = self.end_effector.loc_acting_center_rotmat
        if self.cc is not None:
            self.setup_cc()

    def setup_cc(self):
        # エンドエフェクタの必要なリンクをCollisionCheckerのCCElementとして追加
        elb = self.cc.add_cce(self.end_effector.jlc.anchor.lnk_list[0])
        el0 = self.cc.add_cce(self.end_effector.jlc.jnts[0].lnk)
        el1 = self.cc.add_cce(self.end_effector.jlc.jnts[1].lnk)
        # マニピュレータの必要なリンクをCollisionCheckerのCCElementとして追加
        ml0 = self.cc.add_cce(self.manipulator.jlc.jnts[0].lnk)
        ml1 = self.cc.add_cce(self.manipulator.jlc.jnts[1].lnk)
        ml2 = self.cc.add_cce(self.manipulator.jlc.jnts[2].lnk)
        ml3 = self.cc.add_cce(self.manipulator.jlc.jnts[3].lnk)
        ml4 = self.cc.add_cce(self.manipulator.jlc.jnts[4].lnk)
        ml5 = self.cc.add_cce(self.manipulator.jlc.jnts[5].lnk)
        from_list = [elb, el0, el1, ml4, ml5]
        into_list = [ml0, ml1]
        self.cc.set_cdpair_by_ids(from_list, into_list) # CCElement間の干渉チェックの設定
        # 手持ち部品とCCElementの干渉チェックの設定，リストに含まれた要素のみとチェック
        self.cc.dynamic_into_list = [ml0, ml1, ml2, ml3]

    def fix_to(self, pos, rotmat):
        self._pos = pos
        self._rotmat = rotmat
        # マニピュレータとエンドエフェクタ両方を更新すること
        self.manipulator.fix_to(pos=pos, rotmat=rotmat)
        self.update_end_effector() # 親クラスから継承したものそのまま呼び出す

    def get_jaw_width(self):
        return self.end_effector.get_jaw_width()

    def change_jaw_width(self, jaw_width):
        self.end_effector.change_jaw_width(jaw_width=jaw_width)
```

ロボットは複数のグリッパとマニピュレータから構成されます．robots は grippers や manipulators と同じように子パッケージで各ロボットを定義します．子パッケージはロボットの定義用の.py ファイルとメッシュの CAD モデルを保存するための meshes フォルダなどを格納しています．また，各ロボットの共通のメンバ変数やメンバ関数などを robots.robot_interface.py の RobotInterface インターフェースクラスとして抽出することができ，各ロボットのクラスはこのインターフェースクラスを継承して実装されます．ロボットの定義では，グリッパとマニピュレータ以外に，ロボットの体やベースを JLChain 型を用います．例えば，リスト 5.3-20 は robots/yumi/yumi.py から一部抜粋したものです．Yumi ロボットは二つの JLChain，二つの IRB14050 マニピュレータ，二つの YumiGripper によって組み立てられます．

リスト 5.3-20: robots/yumi/yumi.py の一部抜粋

```
import robot_sim._kinematics.jlchain as jl
import robot_sim.manipulators.irb14050.irb14050 as ya
import robot_sim.grippers.yumi_gripper.yumi_gripper as yg
import robot_sim.robots.robot_interface as ri
# ...他のインポートに関する記述は省略します...


class Yumi(ri.RobotInterface):

    def __init__(self, pos=np.zeros(3), rotmat=np.eye(3), name='yumi', enable_cc=True):
        super().__init__(pos=pos, rotmat=rotmat, name=name)
        this_dir, this_filename = os.path.split(__file__)
        # JLChain型のロボットの身体左側部分で，実際の身体を示します
        # ベースだけではなく，周辺の設備や柱，テーブルもロボットの身体と同じように取り扱います
        self.lft_body = jl.JLChain(pos=pos, rotmat=rotmat,
                                   homeconf=np.zeros(7), name='lft_body')
        # ...リンクと関節の設定を省略します...
        self.lft_body.lnks[1]['collisionmodel'] = cm.CollisionModel(
            os.path.join(this_dir, "meshes", "body.stl"),
            cdprimit_type="user_defined", expand_radius=.005,
            userdefined_cdprimitive_fn=self._base_combined_cdnp)
```

```
22          self.lft_body.reinitialize() # 身体左側のJLChainを再度初期化します
23          # 左腕の定義で，IRB14050型のマニピュレータです
24          lft_arm_homeconf = np.radians(np.array([20, -90, 120, 30, 0, 40, 0]))
25          self.lft_arm = ya.IRB14050(pos=self.lft_body.jnts[-1]['gl_posq'],
26                                     rotmat=self.lft_body.jnts[-1]['gl_rotmatq'],
27                                     homeconf=lft_arm_homeconf, enable_cc=False)
28          # 左腕をロボットの身体左側の先端に装着します.
29          self.lft_arm.fix_to(pos=self.lft_body.jnts[-1]['gl_posq'],
30                              rotmat=self.lft_body.jnts[-1]['gl_rotmatq'])
31          # 左手の定義. YumiGripper型のグリッパ
32          self.lft_hnd = yg.YumiGripper(pos=self.lft_arm.jnts[-1]['gl_posq'],
33                                        rotmat=self.lft_arm.jnts[-1]['gl_rotmatq'],
34                                        enable_cc=False, name='lft_hnd')
35          # 左手をロボットの左腕の先端に装着します.
36          self.lft_hnd.fix_to(pos=self.lft_arm.jnts[-1]['gl_posq'],
37                              rotmat=self.lft_arm.jnts[-1]['gl_rotmatq'])
38          # JLChain型のロボットの身体右側部分で，実際の身体を示さず，右腕の接続用です
39          self.rgt_body = jl.JLChain(pos=pos, rotmat=rotmat,
40                                     homeconf=np.zeros(0), name='rgt_body')
41          # ...リンクと関節の設定や再度の初期化などを省略します...
42          # ...右腕や右手の定義と設定を省略します...
43          # 双腕のTCP座標系をグリッパのTCP座標系として設定します
44          self.lft_arm.tcp_jntid = -1
45          self.lft_arm.tcp_loc_pos = self.lft_hnd.jaw_center_loc_pos
46          self.lft_arm.tcp_loc_rotmat = self.lft_hnd.jaw_center_loc_rotmat
47          self.rgt_arm.tcp_jntid = -1
48          self.rgt_arm.tcp_loc_pos = self.rgt_hnd.jaw_center_loc_pos
49          self.rgt_arm.tcp_loc_rotmat = self.rgt_hnd.jaw_center_loc_rotmat
50          # 把持している対象物のリスト
51          self.lft_oih_infos = []
52          self.rgt_oih_infos = []
53          # 衝突検出用の変数やモデルなどの初期化
54          if enable_cc:
55              self.enable_cc()
56          # ロボットを構成するマニピュレータやグリッパにアクセスするための辞書型変数
57          self.manipulator_dict['rgt_arm'] = self.rgt_arm
58          self.manipulator_dict['lft_arm'] = self.lft_arm
59          self.hnd_dict['rgt_hnd'] = self.rgt_hnd
60          self.hnd_dict['lft_hnd'] = self.lft_hnd
```

コードからわかるように，Yumi ロボットの単腕は，IRB14050 マニピュレータ（12 行目）と YumiGripper（15 行目）エンドエフェクタで構成されています．グリッパーはマニピュレータのフランジに取り付けられています．マニピュレータの TCP は，グリッパーの把持中心に更新されています（18，19 行目）．setup_cc 関数は，衝突検出の要素を設定する際に，マニピュレータとグリッパーの必要なリンクを追加しています（25~27 行目，29~34 行目）．さらに，self.cc の dynamic_into_list メンバ変数を新たに設定し，把持される物体とマニピュレータの間で必要な衝突検出を行うようにしています．この変数については，後続の章でさらに詳しく説明します．fix_to 関数は，単腕ロボットを特定の位置や回転姿勢に設置するために使用され，その内部ではマニピュレータとエンドエフェクタの両方が更新されます．エンドエフェクタの更新には，エンドエフェクタ自体と，そのエンドエフェクタに取り付けられている物体が含まれており，これは親クラスで update_end_effector に実装されているため，ここではそのメソッドを直接呼び出しています（46 行目）．図 12 に示されているのは Yumi の単腕の描画であり，このコンフィギュレーションは，$\boldsymbol{p} = [0.3, 0.1, 0.3]$ m，$\boldsymbol{R} = (0, 2/3\pi, 0)$（回転ベクトル表記）に対して逆運動学を解いた結果です．TCP はすでにグリッパーの把持中心に更新されているため，逆運動学の解も正しく把持中心に対して計算されています．このときに得られた関節角ベクトルは，$[-1.73397097, -1.59037112, 0.40925671, 0.72448174, 1.60718566, 1.75853825, 0.79361618]$ rad です．このベクトルは，図に示されている Jnt0~6 の順に定義されており，この順序は WRS システムのすべてのコードに適用されます．しかし，ABB 社がコントローラを設計する際，Jnt2 を追加の関節として扱い，RAPID

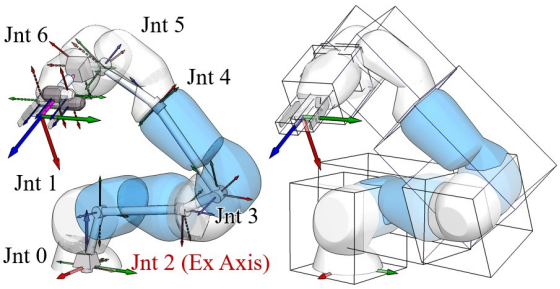

図 10: Yumi ロボットの一つの腕．

言語では 0，1，3，4，5，6，2 の順で各関節を並べ替えています．RAPID プログラムを直接記述する必要がある場合，この違いに注意してください．図中のグリッパーの開閉幅は 0.05m です．図 12 の右半分に表示されているのは，大まかな衝突検出に使用されるプリミティブです．ロボットアームのプリミティブに比べて，手の部分が追加されています．プリミティブはあくまで大まかな衝突検出用であり，それらの衝突は setup_cc によって定義されているため，これらの AABB ボックス同士の重なりについて心配する必要はありません．手指と把持対象物の衝突はメッシュによって検出され，プリミティブは外部障害物との大まかな衝突検出にのみ使用されます．

3.3.2 双腕或いは多腕ロボット

次に，Yumi の双腕ロボットの定義について見ていきます．コードは以下の通りです．双腕ロボットであるため，ここでは SglArmRobotInterface を継承せず，直接 RobotInterface から派生しています．この双腕ロボットは，本質的に 1 つの Anchor インスタンスと，その Anchor インスタンスに取り付けられた 2 つの YumiSglArm インスタンスで構成されています．Anchor インスタンスには，ロボットのボディーや作業台，支柱，ビジョンセンサーなど，多くの周辺構造が付随しています．特にロボットのボディーの cdprim_type は USER_DEFINED に設定されています（18 行目）．この設定により，システムは後に定義されている userdef_cdprim_fn で指定された関数を参照して，衝突干渉チェック用のプリミティブを生成します．

リスト 5.3-21: Yumi 双腕ロボットの定義

```
1  class Yumi(ri.RobotInterface): # 単腕ではないからRobotInterfaceから派生
2
3      def __init__(self, pos=np.zeros(3), rotmat=np.eye(3), name='yumi', enable_cc=True):
4          super().__init__(pos=pos, rotmat=rotmat, name=name, enable_cc=enable_cc)
5          current_file_dir = os.path.dirname(__file__)
6          # 二つの腕を取り付けるため，Anchorを定義（二つのフランジを持つ）
7          self.body = rkjlc.rkjl.Anchor(name="yumi_body", pos=self.pos, rotmat=self.rotmat,
             n_flange=2, n_lnk=9)
8          # 左腕を取付用のフランジ
9          self.body.loc_flange_pose_list[0] = [np.array([0.05355, 0.07250, 0.41492]),
10                                              (rm.rotmat_from_euler(0.9781, -0.5716, 2.3180) @
11                                               rm.rotmat_from_euler(0.0, 0.0, -np.pi))]
12         # 右を取付用のフランジ
13         self.body.loc_flange_pose_list[1] = [np.array([0.05355, -0.07250, 0.41492]),
14                                              (rm.rotmat_from_euler(-0.9781, -0.5682, -2.3155)
                                                   @
15                                               rm.rotmat_from_euler(0.0, 0.0, -np.pi))]
16         # ボディーのモデルをAnchorに設置する．衝突プリミティブを関数で定義する．
17         self.body.lnk_list[0].name = "yumi_body_main"
18         self.body.lnk_list[0].cmodel = mcm.CollisionModel(initor=os.path.join(current_file_dir
             , "meshes", "body.stl"), cdprim_type=mcm.mc.CDPType.USER_DEFINED, userdef_cdprim_fn=
             self._base_cdprim)
19         self.body.lnk_list[0].cmodel.rgba = rm.bc.hug_gray
20         # 作業台と周りの柱もAnchorに設置する
21         self.body.lnk_list[1].name = "yumi_body_table_top"
22         self.body.lnk_list[1].cmodel = mcm.CollisionModel(
23             initor=os.path.join(current_file_dir, "meshes", "yumi_tablenotop.stl"))
24         self.body.lnk_list[1].cmodel.rgba = rm.bc.steel_gray
25         ... # その他の柱の定義を省略する
26         # 視覚センサーをAnchorに設置する
27         self.body.lnk_list[8].name = "phoxi"
28         self.body.lnk_list[8].loc_pos = np.array([.273, 0, 1.085])
29         self.body.lnk_list[8].cmodel = mcm.CollisionModel(
30             initor=os.path.join(current_file_dir, "meshes", "phoxi_m.stl"))
31         self.body.lnk_list[8].cmodel.rgba = rm.bc.black
32         # 左腕は一つのYumiSglArmのインスタンスです
33         self.lft_arm = ysa.YumiSglArm(pos=self.body.gl_flange_pose_list[0][0],
34                                       rotmat=self.body.gl_flange_pose_list[0][1],
35                                       name='yumi_lft_arm', enable_cc=False)
36         self.lft_arm.home_conf = np.radians(np.array([20, -90, 120, 30, 0, 40, 0]))
37         # 右腕も一つのYumiSglArmのインスタンスです
38         self.rgt_arm = ysa.YumiSglArm(pos=self.body.gl_flange_pose_list[1][0],
39                                       rotmat=self.body.gl_flange_pose_list[1][1],
40                                       name='yumi_rgt_arm', enable_cc=False)
41         self.rgt_arm.home_conf = np.radians(np.array([-20, -90, -120, 30, .0, 40, 0]))
42         if self.cc is not None:
```

```
            self.setup_cc()
        # 順運動学の計算でロボットを初期コンフィギュレーション（上記のhome_conf）に更新
        self.goto_home_conf()
```

多腕ロボットを定義する際，重要なメンバ変数の 1 つが，RobotInterface から継承された delegator 変数です．デフォルトでは，この変数の値は None に設定されています．この変数の値を設定することで，関数の呼び出し対象を変更することができます．例えば，Yumi ロボットはリスト 5.3-22 に示される use_both(), use_lft(), use_rgt() などの関数を提供して，delegator を切り替えることができます．具体的なメソッドが呼び出された際には，まず delegator の値を確認し，その後に左腕，右腕，または双腕同時の計算を行うかが決定されます．リスト 5.3-23 に示されている goto_given_conf メソッドの例では，delegator の値に基づいて処理が判断されています．

リスト 5.3-22: use 関数により delegator を設定

```
    def use_both(self):
        self.delegator = None

    def use_lft(self):
        self.delegator = self.lft_arm

    def use_rgt(self):
        self.delegator = self.rgt_arm
```

リスト 5.3-23: delegator の判断により腕を選択

```
    def goto_given_conf(self, jnt_values, ee_values=None):
        if self.delegator is None: # ディフォルトで双腕同時に計算する
            if len(jnt_values) != self.lft_arm.manipulator.n_dof + self.rgt_arm.manipulator.
              n_dof:
                raise ValueError("The given joint values do not match total n_dof")
            self.lft_arm.goto_given_conf(jnt_values=jnt_values[:self.lft_arm.manipulator.n_dof
              ])
            self.rgt_arm.goto_given_conf(jnt_values=jnt_values[self.rgt_arm.manipulator.n_dof
              :])
        else: # そうではない場合delegatorに頼む
            self.delegator.goto_given_conf(jnt_values=jnt_values, ee_values=ee_values)
```

CollisionChecker を設定するための関数は，リスト 5.3-24 に示されている通りです．ここでは，Anchor，各マニピュレータ，および各グリッパーの必要なリンクを CCElement に追加するだけでなく，自分自身の self.cc を各アームに割り当てています．これは，単腕が動作する際の衝突検出が，全体のロボットの CCElement を考慮する必要があるためです．図 11(a) には，Yumi ロボットの衝突プリミティブが示されています．

リスト 5.3-24: delegator の判断により腕を選択

```
    def setup_cc(self):
        # Anchor関連のCCElementの添加
        bd = self.cc.add_cce(self.body.lnk_list[0])
        wb = self.cc.add_cce(self.body.lnk_list[1])
        lc = self.cc.add_cce(self.body.lnk_list[2])
        rc = self.cc.add_cce(self.body.lnk_list[3])
        tbc = self.cc.add_cce(self.body.lnk_list[4])
        tlc = self.cc.add_cce(self.body.lnk_list[5])
        trc = self.cc.add_cce(self.body.lnk_list[6])
        tfc = self.cc.add_cce(self.body.lnk_list[7])
        phx = self.cc.add_cce(self.body.lnk_list[8])
        # 左腕のグリッパーのCCElementの添加
        lft_elb = self.cc.add_cce(self.lft_arm.end_effector.jlc.anchor.lnk_list[0])
        lft_el0 = self.cc.add_cce(self.lft_arm.end_effector.jlc.jnts[0].lnk)
        lft_el1 = self.cc.add_cce(self.lft_arm.end_effector.jlc.jnts[1].lnk)
        # 左腕のマニピュレータのCCElementの添加
        lft_ml0 = self.cc.add_cce(self.lft_arm.manipulator.jlc.jnts[0].lnk)
        lft_ml1 = self.cc.add_cce(self.lft_arm.manipulator.jlc.jnts[1].lnk)
        lft_ml2 = self.cc.add_cce(self.lft_arm.manipulator.jlc.jnts[2].lnk)
        lft_ml3 = self.cc.add_cce(self.lft_arm.manipulator.jlc.jnts[3].lnk)
        lft_ml4 = self.cc.add_cce(self.lft_arm.manipulator.jlc.jnts[4].lnk)
        lft_ml5 = self.cc.add_cce(self.lft_arm.manipulator.jlc.jnts[5].lnk)
        # 右腕のグリッパーのCCElementの添加
        rgt_elb = self.cc.add_cce(self.rgt_arm.end_effector.jlc.anchor.lnk_list[0])
```

```
        rgt_el0 = self.cc.add_cce(self.rgt_arm.end_effector.jlc.jnts[0].lnk)
        rgt_el1 = self.cc.add_cce(self.rgt_arm.end_effector.jlc.jnts[1].lnk)
        # 右腕のマニピュレータのCCElementの添加
        rgt_ml0 = self.cc.add_cce(self.rgt_arm.manipulator.jlc.jnts[0].lnk)
        rgt_ml1 = self.cc.add_cce(self.rgt_arm.manipulator.jlc.jnts[1].lnk)
        rgt_ml2 = self.cc.add_cce(self.rgt_arm.manipulator.jlc.jnts[2].lnk)
        rgt_ml3 = self.cc.add_cce(self.rgt_arm.manipulator.jlc.jnts[3].lnk)
        rgt_ml4 = self.cc.add_cce(self.rgt_arm.manipulator.jlc.jnts[4].lnk)
        rgt_ml5 = self.cc.add_cce(self.rgt_arm.manipulator.jlc.jnts[5].lnk)
        # 腕とボディーの干渉チェックのペア
        from_list = [lft_ml4, lft_ml5, lft_elb, lft_el0, lft_el1, rgt_ml4, rgt_ml5, rgt_elb,
          rgt_el0, rgt_el1]
        into_list = [bd, wb, lc, rc, tbc, tlc, trc, tfc, phx, lft_ml0, rgt_ml0]
        self.cc.set_cdpair_by_ids(from_list, into_list)
        # マニピュレータとグリッパー間の干渉チェックのペア
        from_list = [lft_ml0, lft_ml1, rgt_ml0, rgt_ml1]
        into_list = [lft_elb, lft_el0, lft_el1, rgt_elb, rgt_el0, rgt_el1]
        self.cc.set_cdpair_by_ids(from_list, into_list)
        # 左右腕の干渉チェックのペア
        from_list = [lft_ml1, lft_ml2, lft_ml3, lft_ml4, lft_ml5, lft_elb, lft_el0, lft_el1]
        into_list = [rgt_ml1, rgt_ml2, rgt_ml3, rgt_ml4, rgt_ml5, rgt_elb, rgt_el0, rgt_el1]
        self.cc.set_cdpair_by_ids(from_list, into_list)
        # 自分自身の self.cc を各アームに割り当てます
        self.lft_arm.cc = self.cc
        self.rgt_arm.cc = self.cc
```

リスト 5.3-25 のコードは，左右の手それぞれの IK を求解する例です．use 関数を呼び出して delegator を左手または右手に設定することで，各手ごとに計算を行うことができます．このコードの実行結果は図 11(b) と (c) の通りです．

リスト 5.3-25: 左右の手それぞれの IK を求解する例

```
... # importやインスタンスの定義などを省略
    tgt_pos = np.array([.6, .0, .3])
    tgt_rotmat = rm.rotmat_from_axangle([0, 1, 0], math.pi / 2)
    gm.gen_frame(pos=tgt_pos, rotmat=tgt_rotmat).attach_to(base)
    robot.use_rgt() # 右でを使う，use\_lft()で左に切り替えは可能
    jnt_values = robot.ik(tgt_pos, tgt_rotmat)
    robot.goto_given_conf(jnt_values=jnt_values)
    robot.gen_meshmodel().attach_to(base)
    base.run()
```

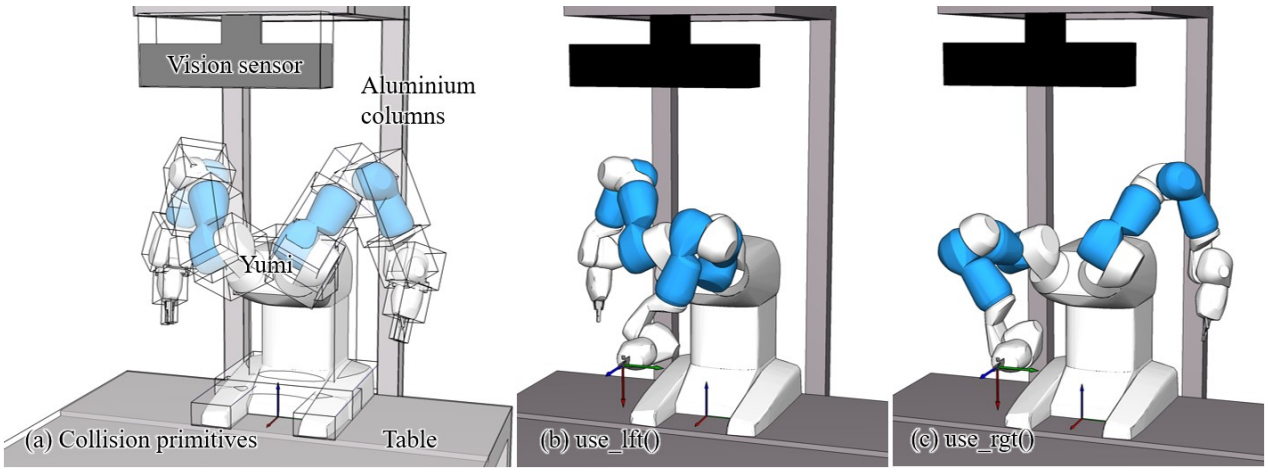

図 11: (a) 衝突干渉チェック用プリミティブ．(b, c) リスト 5.3-25 のコードの結果．(b) 左腕を使う．(c) 右腕を使う．

3.4 実例

これまでの本章では，WRS システムの多くの継承関係や基礎的な構成について紹介してきました．これは，システムの動作方法に対する読者の理解を深めることを目的としています．実際の使用においては，新しいロボット

を定義する場合を除き，これらの基礎コードに触れる必要はなく，それらを基にして二次開発を行うだけで十分です．本節では，Yumi の直線軌道生成を例に，開発事例を紹介します．コードはリスト 5.3-26 の通りです．

リスト 5.3-26: Yumi の右手の動作の直線補完

```
1  import basis.robot_math as rm
2  import robot_sim.robots.yumi.yumi as ym
3  import visualization.panda.world as wd
4  import motion.primitives.interplated as mip
5
6  if __name__ == '__main__':
7      base = wd.World(cam_pos=[3, 1, 1], lookat_pos=[0, 0, 0.5])
8      robot = ym.Yumi(enable_cc=True)
9      robot.use_rgt()
10     interp_planner = mip.InterplatedMotion(robot=robot)
11     start_pos = rm.np.array([.6, -.3, .5])
12     start_rotmat = rm.rotmat_from_axangle([0, 1, 0], rm.np.pi / 2)
13     goal_pos = rm.np.array([.6, .0, .3])
14     goal_rotmat = rm.rotmat_from_axangle([0, 1, 0], rm.np.pi)
15     mot_data = interp_planner.gen_linear_motion(start_tcp_pos=start_pos, start_tcp_rotmat=
         start_rotmat, goal_tcp_pos=goal_pos, goal_tcp_rotmat=goal_rotmat,toggle_dbg=True)
16     for mesh in mot_data.mesh_list:
17         mesh.attach_to(base)
18     base.run()
```

このコードは，motion.primitives.interpolated を使用して，運動軌跡の線形補間を実行しています．コードの第 10 行では，線形補間器のインスタンスが定義されており，その初期化パラメータにはロボットのインスタンスが指定されています．線形補間器は，このインスタンスが単腕ロボットであるか，またはデリゲーターとして単一アームが指定されている多腕ロボットである必要があります．第 15 行では，線形補間器のメンバーメソッド gen_linear_motion が呼び出され，指定された 2 つの位置と回転に対して線形補間を行っています．このメソッドの内部では，点と点の間の逆運動学（IK）を解き，MotionData オブジェクトを返します．gen_linear_motion 関数と MotionData の定義については，motion.primitives.interpolated.py ファイルおよび motion.motion_data.py ファイルを参照してください．特に，gen_linear_motion 関数は，ニュートン・ラフソン法における初期値の選択問題を自動的に考慮しています．最初の位置の逆解を求める際には初期のコンフィギュレーションを指定せず，ソルバーが自動的に決定します（DDIKSolver を使用する場合，KDTree で選択）．その後の位置の逆解を求める際には，前の位置の解を初期値として使用し，運動軌跡の連続性を確保します．この実例の実行結果は図に示します．

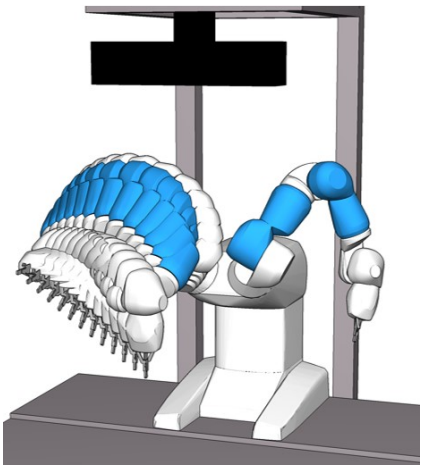

図 12: リスト 5.3-26 の結果．