

手先カメラを用いた双腕ロボットによる
マニピュレーションシステム
操作手順書

東京大学 情報理工学系研究科 稲葉研究室

平成 24 年 2 月 10 日

目次

第 1 章	システム概要	2
1.1	全体のモジュール構成	2
1.2	認識部	3
1.2.1	エッジベース二次元対象物認識モジュール (AppRecog)	3
1.2.2	カメラ共通 I/F 準拠の画像キャプチャモジュール (CameraComp)	5
1.2.3	認識部の動作確認	5
1.3	動作生成部	5
1.3.1	(VPython 版)HiroNX 動作生成モジュール	5
1.3.2	HiroNXInterface	7
1.3.3	動作生成部の動作確認	8
1.3.4	動作生成を対話環境で行う場合	8
1.3.5	動作生成を動作計画モジュールとして分離する場合	8
第 2 章	準備	9
2.1	モデルファイルの修正 (カセンサありとなし)	9
2.2	キャリブレーション	9
2.2.1	カメラキャリブレーション	9
2.2.2	カメラ取り付け位置のキャリブレーション	9
第 3 章	デモの実行	11
3.1	認識部のモジュール起動と接続	11
3.2	動作生成プログラムの起動 (1 つの python プロセス上での実行)	11
付 録 A	VPython 環境でのプログラミング	12

第1章 システム概要

本サービスは、工場での部品整理をイメージしたものである。具体的には手先のカメラを用いて作業台上の部品を認識し、両手で箱に整理して入れる機能を実現する。手を動かすことで複数の対象物を認識、両手の干渉を考慮して同時にアプローチできる対象物を選択する。

1.1 全体のモジュール構成

以下に、本システムで利用するモジュールの一覧を示す。

- app-recog
- CameraComp
- LoadPictureComp
- iv_plan_hironx
- HiroNXInterface

ファイルシステム上の場所は必ずしも重要ではないが、ディレクトリ構成は揃えておくことと本ドキュメントとあわせて理解しやすい。

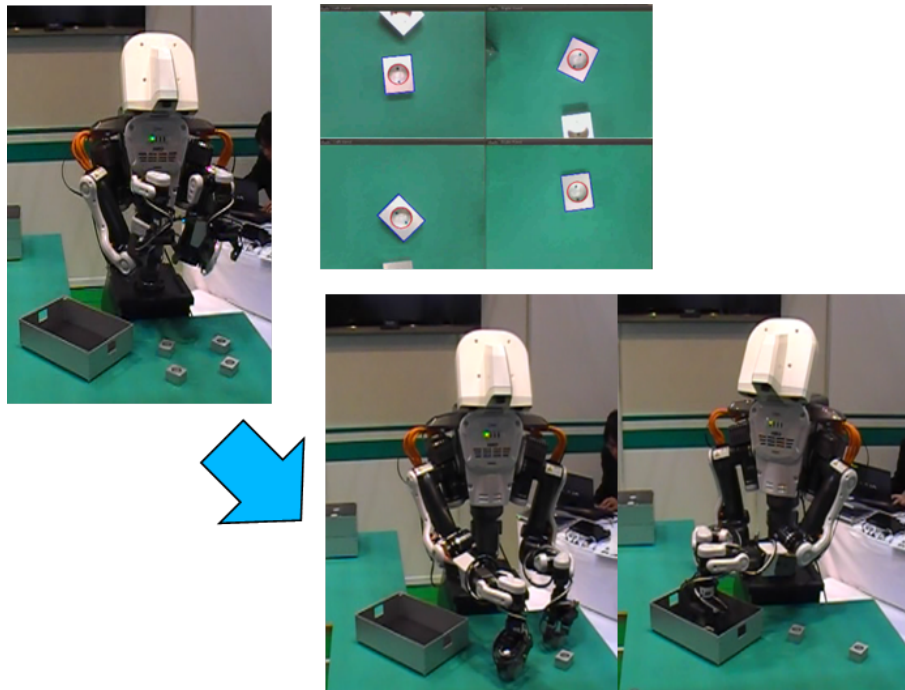


図 1.1: サービスイメージ

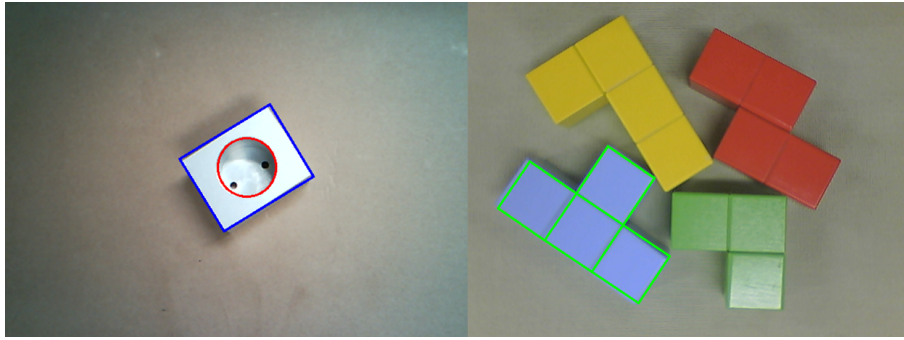


図 1.3: 認識例

インタフェース

• データポート

- 入力: `Img::TimedCameraImage` (`Img.idl`)
画像出力共通インタフェース準拠のカメラモジュールから、画像及び、カメラパラメータを受取ります。
- 出力: `TimedRecognitionResult` (`Vision.idl`)
認識結果共通インタフェースにしたがい、対象物体の位置姿勢を出力します。
`Img::TimedCameraImage` 処理結果を画像として出力します。

• サービスポート

認識対象のモデルを設定するために使います。あらかじめ、`ModelFiles/ModelList.txt` にモデル ID とモデル定義ファイル名を記述し、モデル ID を引数としてサービスコールを行います。`setModelID(i)` は、*i* 番のモデルを使用することを意味します。

認識結果は `TimedRecognitionResult` によって出力されます。具体的な出力内容は以下の通りです。現在、対象物の姿勢以外は入っていません。

```
0: 0, 1: 0, 2: 0, 3: 0, 4: 0
5: 0, 6: 0, 7: 0,
8: R00, 9: R01, 10: R02, 11: Tx
12: R10, 13: R11, 14: R12, 15: Ty
16: R20, 17: R21, 18: R22, 19: Tz
```

カスタマイズ

連続的に送られてくる画像に対して認識を行います。認識結果の時間方向の連続性は考慮せず、各フレームで一番尤度が高い位置を計算し、その尤度が閾値以上であれば検出結果を返します。

モデルと実画像のマッチングは、画像座標系で行われます。検出した位置、姿勢、スケールからカメラパラメータを用いて、カメラ座標系における対象物の位置、姿勢が計算されます。したがって、画像座標での (x, y, θ) の探索範囲、検出の閾値をコンフィグファイル `AppRecog.conf` で指定します。また、モデル定義ファイルは `./ModelFile` の中に置き、ファイルは頂点と辺によって構成されています。

1.2.2 カメラ共通 I/F 準拠の画像キャプチャモジュール (CameraComp)

<http://www-arailab.sys.es.osaka-u.ac.jp/CameraIF/>

大阪大学により開発され画像キャプチャモジュール CameraComp をダウンロード，コンパイルする．ログ画像によるテストを行うため，LoadPictureComp モジュールも同様にダウンロードするとよい．

1.2.3 認識部の動作確認

実際にカメラモジュールと接続し，オンラインでテストを行う．このときのモジュール接続は図 1.4 のようになり，実行手順は，以下の通りである．

1. 認識モジュール AppRecog とキャプチャモジュールをそれぞれ実行する．
2. rtshell で画像の入出力を接続する (system editor 上で操作してもよい) ．
3. 2 つの RTC を activate する ．

```
$ cd CameraComp
$ ./CaptureCameraComp
別端末で
$ cd app-recog/
$ build/bin/AppRecogComp
別端末で ( rtctree でのパスは適当に補完する )
$ rtcon CaptureCamera0.rtc:CameraImage AppRecog0.rtc:InputImage
$ rtact CaptureCamera0.rtc AppRecog0.rtc
```

初期設定で認識範囲のスケールを絞ってあるため，認識できない場合は対象物までの距離をいろいろ変えてみる．また，背景に模様がなく，対象物と異なる色のものを置くと認識しやすくなる．

テストとして，カメラモジュールを LoadPictureComp モジュールに差し替え，あらかじめ撮っておいた画像を用いてテストを行う場合，CaptureCameraComp を LoadPictureComp に置き換えた接続となる．ログ画像の指定は，LoadPictureComp モジュールの LoadPicture.conf で行う．AppRecog モジュール付属の画像 data/parts4.jpg を用いて確認を行うとよい．状態空間における探索範囲は別途指定が可能である．

1.3 動作生成部

1.3.1 (VPython 版)HiroNX 動作生成モジュール

http://openrtm.org/openrtm/ja/project/NEDO_Intelligent_PRJ_HiroAccPrj_5003

Python 対話環境において、幾何モデルを用いた動作生成システムを柔軟に構築するためのスクリプト群です．RtcHandle を用いて対話環境から RTC 構成によるシステムの各モジュールと通信を行うことで，システムの統合を行います．RRT-connect による双腕の干渉を考慮した動作計画機能を提供し，人手による動作記述とプランナによる動作生成をスムーズに統合できます．また，RTC として，作業共通インタフェースを実装する動作生成モジュールとして利用することもできます．

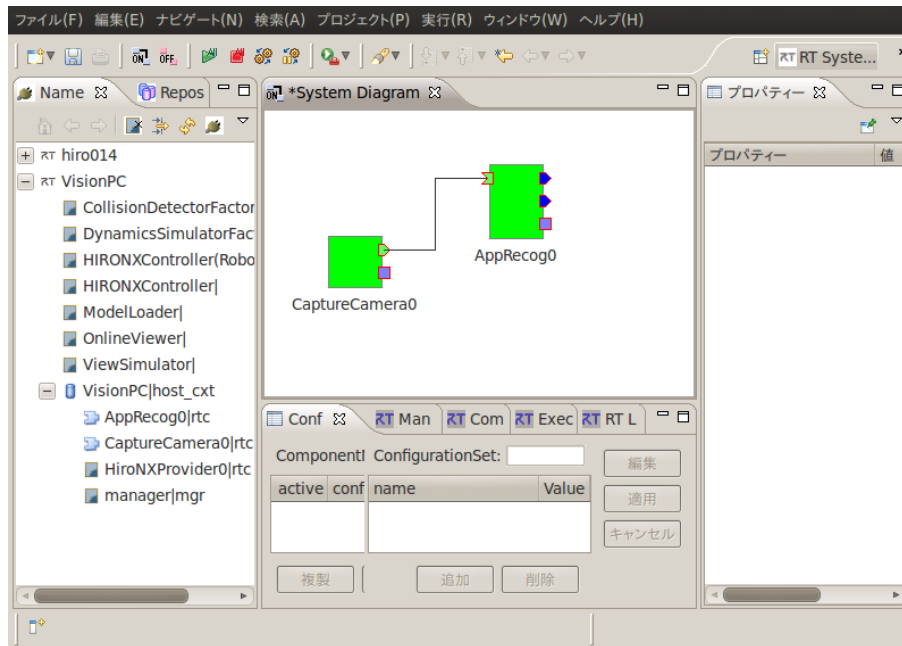


図 1.4: USB カメラを用いたテスト

ダウンロードとコンパイル

```
$ cd iv-plan-hironx/iv_plan/scripts; ./install-debs.sh
```

```
$ cd iv-plan-hironx/iv_plan/; make
```

環境変数PYTHONPATHに `iv-plan-hironx/iv_plan/src` を追加

環境によって VPython が正しく動作しない場合があるため、また、deb パッケージでインストールされるバージョンではサポートされていない機能を利用するため、VPython は Ubuntu 10.04LTS の標準パッケージより新しいものにパッチを当て、コンパイル済みのものを利用します。これは、make 時にダウンロードします。したがって、標準の python-visual パッケージをインストールしている場合は、それが python 上で先にロードされないように注意する必要があります。ikfast により生成された HIRO-NX 用逆運動学計算のソースは同梱されており、PQP のソースは make 時にダウンロードし、それぞれコンパイルされます。それ以外に必要なものは apt コマンドでインストールします (install-deb.sh)。

開発・動作環境

- Ubuntu Linux 10.04 LTS 32bit/64bit
- OpenRTM-aist 1.0.0 (Python 版)

基本的な使い方

まず、タスク記述、動作計画、認識および制御モジュールとの接続を 1 つの Python シェル上で行う方法を説明します。

```
$ cd iv_scenario/src
```

```
$ ipython test.py
```

```
>>> test1()
```

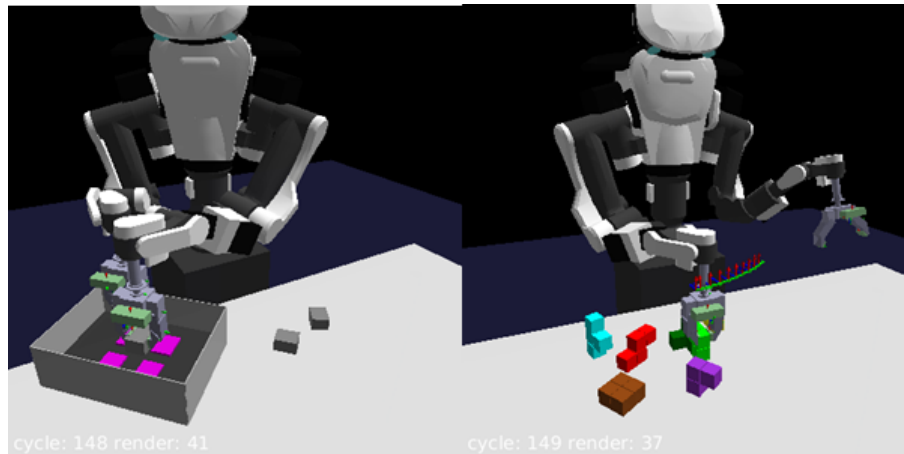


図 1.5: 動作生成モジュール

立体パズルデモ

```
$ ipython puzzle.py
>>> demo() # パズル配置を元に戻すのは reset_puzzle()
```

シミュレーションによる部品の箱詰め

```
>>> 変更中
```

ウィンドウ操作は、右ドラッグで回転、中ドラッグでズームです。プログラムの終了は Ctrl+d でインタプリタを抜けた後、GUIの閉じるボタンを押します。Ctrl+c はトラップされています。GUIの閉じるボタンを押すのが面倒な場合は Ctrl+\ で SIGQUIT を送ることによって終了させることができます。

Python シェル上でロボットの姿勢を変更したり、環境中の物体を移動させたり、動作計画を行う場合の操作に関しては付録を参照してください。

外部RTC との通信には RtcHandle を利用しています。(http://staff.aist.go.jp/t.suehiro/rtm/rtc_handle.html)

基本的に、ロボットクラス、ロボットインスタンスごとのカスタマイズは既存クラスを拡張することで行います。

- robot.py から hironx.py
- hironx_params.py 個体差がある各種 transform と、利用する外部モジュール定義
- 環境モデルの定義
(python のデータ構造で与える。scene_object.py)
- 他の機能として、VRML のローダ、センサ取り付け位置の推定機能があります。

インタフェース

1.3.2 HiroNXInterface

<http://www.openrtm.org/openrtm/en/node/4645>

双腕ロボットの制御コマンドの共通インタフェースに準拠する HiroNX 用制御モジュールです。詳細については、開発元である産業技術総合研究所のドキュメントを参照してください。

1.3.3 動作生成部の動作確認

以下のコマンドでデモプログラムを起動します .

```
$ cd iv_scenario/src
$ ipython demo-wexpo.py
```

シミュレーションでのデモ動作の実行

```
>>> demo(False)
```

外部モジュールとの通信

(AppRecogモジュールを実行しておく)

```
>>> rr.detect(sensor='rhandcam') # 右手ハンドカメラでの認識
```

以下は , HiroNXの起動が完了し , HiroNX体内のモジュールと通信可能な状態であるとする .

```
>>> detect(sensor='rhandcam') # 認識結果を世界座標に直したもの
>>> rr.get_joint_angles() # 実機の関節角度列
>>> r.prepare() # シミュレータ内で姿勢を変更
>>> sync() # 実機の姿勢をシミュレータにあわせる
```

1.3.4 動作生成を対話環境で行う場合

1.3.5 動作生成を動作計画モジュールとして分離する場合

シミュレーション環境外部モジュールとの通信動作コマンド送信 , 状態読み込み

第2章 準備

必要に応じて行うロボットごとに以下の作業を行う。

2.1 モデルファイルの修正（カセンサありとなし）

2.2 キャリブレーション

ロボットの個体差を修正する作業である。デフォルト値はHiroNX16号機のものであり、精度を出すには各機体ごとに行う必要がある。

2.2.1 カメラキャリブレーション

RT-middleware のコンポーネントでも ROS のノードでも何でもよいので単眼カメラのキャリブレーションを行い、CameraComp が読み込めるようにする。

その後、エッジベース二次元対象物認識モジュールにおいて、対象物の位置、距離が正しく出力されているかどうか確認しておくといよい。

2.2.2 カメラ取り付け位置のキャリブレーション

現状では、チェッカーボードの認識に ROS のノードを使用している。キャリブレーションプログラムは、学習データとしてロボットの各姿勢におけるチェッカーボードの姿勢を入力とします。チェッカーボードの姿勢は tf として publish されたメッセージを python プログラムで受信します。したがって、準備として、

- ROS のカメラノードにより画像および上記カメラパラメータがトピックとして publish
- checkerboard detection ノードでそれらを subscribe し、推定した tf が publish

される状態にしておく必要があります。

手順

1, 机の上にチェッカーボードを置く

2, 首を動かして学習データをとる

```
$ ipython hironx_calib.py
```

```
>>> res = record_data()
```

```
# 手先を動かして画像とそのときの関節角度値を取得する。
```

```
# すべての姿勢でチェッカーボードが視野に入り、
```

安定して認識できているかどうかを確認する .

3, 頭リンクから kinectカメラへのtransformを計算する

```
>>> f = calibrate(res, height=960.0)
```

4, transformの書き換え

```
>>> r.Thd_kinectrgb = f
```

5, 確認 (正しい位置でキャリブボードのフレームが止まっていれば成功)

```
>>> play_data(res)
```

6, デフォルト値の更新

hironx_params.pyのThd_kinectdepthを書き換える

- 上のコマンドは頭部 kinect の場合なので, ハンドカメラの場合に変更する
- 原理について

第3章 デモの実行

3.1 認識部のモジュール起動と接続

VisionPC にて, `run.sh` を実行する．これにより, 両手に対応するカメラ, 認識モジュール, `HiroNXInterface` 制御モジュールが起動, 接続され, さらに各モジュールが `activate` されます．`HiroNXInterface` モジュールは起動後 GUI 上の「RTC Status」が緑になった状態で「Set up Robot」ボタンを押し RTC まわりの初期化を行う必要がある点に注意してください．

ここで, `rtc.conf`, 左右カメラの左右のカメラが逆になっていないかどうか確認してください．

3.2 動作生成プログラムの起動 (1 つの python プロセス上での実行)

- テーブル上に対象物を 4 つ配置し, `look_for()` 関数を実行します．手先を動かして机上の対象物を認識してみます．
- このとき, 画像中で対象物が正しく認識されているか, 認識結果がシミュレータ中に正しく表示されているかどうかを確認します．
- これで準備が整ったので, 実際にデモを実行します．実行後, ロボットはテーブルをスキャンし, 対象物を認識 `demo()`

付 録 A VPython環境でのプログラミング

フレームやロボット，環境中の物体操作等を python 上で行う方法について説明します．
以下の情報は古いもので，更新予定です．

0, はじめに

```
$ roscd MotionPlan/src
$ ipython demo.py
```

1, python を使う

```
dir(r) # 関数やメソッドの一覧
r. [TAB] # メソッド名等の補間 (ipython の機能)
help(r) # 引数や説明
```

とりあえずデモ

```
putbox() # 箱を認識して掴み、置き直す
putbox(name='box1')
demo()
```

demo2() # 左手でのアプローチ

```
qs = gen_traj() # 手先で sin カーブを描くための関節角軌道を作る
play_traj(qs) # 作った軌道を実行する
```

コマンドライン

```
# Ctrl+c
# Ctrl+p / Ctrl+n
# Ctrl+r
```

2, 環境中の物体操作

```
env
env.get_objects() # 環境中の物体一覧
[x.name for x in env.get_objects()] # 物体名一覧
```

```
# 物体は名前で管理されている。
# 同じ名前の物体は追加できないので、一度削除するか、名前を変えて追加する。
putbox() # テーブル上に箱を置く
b = env.get_object('box') # 名前で物体取得
put_box(name='box2') # 名前を変えると違う物体
env.delete_object('box2')
```

```
frm = b.where() # 物体の位置と姿勢を取得
```

```
# ロボットの移動、回転 ( world=>basejoint の座標変換の変更 )
r.go_pos(-150,500,0) # 2D 座標, (x,y,theta)
r.go_pos(-150,0,pi/2) # 横を向く
```

3, 自作サンプルの書き方

```
emacs mysample.py
```

```
== mysample.py ==
from demo import *
```

```
... 適当なコード ...
== ==
```

```
ipython mysample.py
```

```
# サンプルを修正後は、以下を実行すると
# mysample.py の修正が反映される
```

```
import mysample # 最初の一回だけ
```

```
reload(mysample)
from mysample import *
```

4, ロボットの姿勢の操作

```
# joint, link の取得
r
r.get_joints()
r.get_links()
```

```
# ジョイント名の取得
j0 = r.get_joints()[0]
```

```

dir(j0)
j0.angle
j0.name
map(lambda x: x.name, r.get_joints())
[x.name for x in r.get_joints()]

# 関節角度の表示
r.get_joint_angles()
# これは下のコードと同じ
[x.angle for x in r.get_joi

# 腕だけ
r.get_arm_joint_angles()
r.get_arm_joint_angles(arm='left')
r.set_arm_joint_angles(angles)
r.set_arm_joint_angles(angles, arm='left')

# ハンドだけ
r.get_hand_joint_angles()
r.get_hand_joint_angles(hand='left')
r.set_hand_joint_angles(angles)
r.set_hand_joint_angles(angles, hand='left')

# 関節1つを変える
r.get_joint_angle(0) # ID
r.set_joint_angle(0, 0.5) # ID と角度 [rad]

# 初期姿勢に戻す
r.reset_pose()

# あらかじめいくつかの姿勢が定義されている
r.poses
r.poses.keys()

# r.reset_pose() は以下と同じ
r.set_joint_angles(r.poses['init'])

# 手のポーズ
r.hand_poses
r.hand_poses['open']
r.set_hand_joint_angles(r.hand_poses['open']) # 手を開く
r.set_hand_joint_angles(r.hand_poses['close']) # 手を閉じる

# アニメーション (首を振ってみる)

```

```

arange(0, 1, 0.2)
for th in arange(0,1,0.2):
    r.set_joint_angle(1, th)
    time.sleep(0.5)

# ちなみに左右の腕の関節書く取得は以下のコードと同じ
r.get_joint_angles()[3:9]
r.get_joint_angles()[15:19]

# 5, 座標系 (frame) について
# 3x3 回転行列と 3 次元ベクトル = 4x4 の同次数行列
# euler 角, 自由軸回転, quaternion

help(VECTOR)
help(MATRIX)
help(FRAME)

# 値の生成 (constructor)
VECTOR()
MATRIX()
FRAME()
v=VECTOR(vec=[100,0,0])
MATRIX(angle=pi/2, axis=VECTOR(vec=[0,0,1]))
MATRIX(c=pi/2) # a,b,c を同時に指定できないので注意
m=MATRIX([[1,0,0],[0,1,0],[0,0,1]])
FRAME()

frm.mat # 姿勢部分
frm.vec # 位置部分

# 演算
v*v # ベクトル積
dot(v,v) # 内積
v+v # 和
2*v # スカラー倍
# 行列は姿勢表現専用 (直行行列)
m*m # 積
-m # 逆行列
# -m はじ実装は転置行列
# スカラー倍、行列和は定義されない (配列の結合解釈される)

# 姿勢表現間の変換

```



```

m.abc() # 行列=>euler
m.rot_axis() # 行列=>自由軸回転
# euler=>回転行列、自由軸回転=>回転行列は上記の MATRIX のコンストラクタ

# 位置と姿勢を合わせた表現（同時行列）
FRAME(mat=m, vec=v)
FRAME(xyzabc=[x,y,z,a,b,c])

f=FRAME(vec=[500,0,1000])
show_frame(f)
f.mat = MATRIX(a=pi/4)
show_frame(f)

f.mat = MATRIX(a=pi/4)*MATRIX(b=pi/4)
show_frame(f)

# 座標系の親子構造
FRAME.affix() # 座標系の親子関係の定義
FRAME.unfix() # 座標系の親子関係の削除
FRAME.set_trans() # 親子間の座標変換の設定
f.rel_trans # 親子間の座標変換の取得

# 物体追加
# putbox() の記述を参照
# 表示用形状と FRAME を作り、適当な親座標系の子 FRAME として、座標系ツリーに挿入する

env.insert_object('box2') # 物体追加
env.delete_object('box2') # 物体削除（子フレームの物体も削除される）

# 6, 逆運動学 (Inverse Kinematics, IK) の利用

# 手首位置，姿勢の取得
r.fk() # 順運動学計算，現在の手先 FRAME を返す．デフォルトは右手．
r.fk(arm='left') # 左手は明示的に引数で指定する

# 目的位置へのアプローチ

putbox() # 手が届きそうなところに置く
objfrm = detect()

# アプローチ姿勢，把持姿勢の計算（2 つずつ求まる）
afrms, gfrms = pl.grasp_plan(objfrm)

```

```

# アプローチ姿勢を表示
# (FRAME から CoordinateObject を作って、環境に insert_object する)
show_frame(afrms[0])
show_frame(afrms[1])

# 目標位置へ手を伸ばすための関節角度を計算する
r.ik(afrms)
help(r.ik)
# - IK は目標手先 FRAME の集合を引数にとり、
#   初期姿勢から関節空間での重み付き距離順に並べた解の列を返す
#   目標手先フレーム 1 つを引数に指定してもよい
# - 解が存在しなければ None
# - ほとんどの場合、最初の解を採用すればよい

asols = r.ik(afrms)
r.set_arm_joint_angles(asols[0]) # 腕の角度のみ

gsols = r.ik(gfrms)
r.set_arm_joint_angles(gsols[0]) # 腕の角度のみ

# 腰を使う
# 腰 yaw 軸を使うと手の届く範囲が大きく広がる

asols = r.ik(afrms, use_waist=True) # 返値の形式が違うので注意 (waist_yaw, arm_angles)
th, js = asols[0]
r.set_joint_angle(0, th) # 腰を回す
r.set_arm_joint_angles(js) # 腕の姿勢を変える

# 他の解も表示してみる
for th, js in asols:
    r.set_joint_angle(0, th)
    r.set_arm_joint_angles(js)
    time.sleep(0.5)

# 物体をハンドに固定する
tgtobj = env.get_object('box0')
handjnt = r.get_joint('RARM_JOINT5')
reltf = (-handjnt.where())*tgtobj.where()
tgtobj.unfix()
tgtobj.affix(handjnt, reltf)

# 手先軌道での動作生成 (収束 IK ではなく、初期姿勢に近い解を明示的に選択している)

```

```

# 軌道の作成
traj = CoordinateObjects('trajectory')
traj.append( ... ) # 適当なフレームを追加する
env.insert_object(traj, FRAME(), env.get_world()) # 世界座標相対で軌道を定義
traj.coords

# 削除したいときは
env.delete_object('trajectory')

# 7, 実機を動かす

# 実機とのインタフェース

rr = RealHIRO()
rr.connect() # 認識処理が始まっていなければ開始指令を送る
rr.get_joint_angles() # tf and joint state by ROS
js = r.get_joint_angles()
# socket(+pickle) to jython script
# ssequencer, wait
duration = 4.0
rr.send_goal(js, 4.0) # blocking
rr.send_goal(js, 4.0, wait=False) # non-blocking
# blocking な呼出しも最大 10[sec] で timeout する仕様

# 実際には、モデルで姿勢を確認してから以下を実行するのが便利
sync() # デフォルトは 4[sec] で実機をモデルに同期させる
sync(duration=3.0) # 時間を変える

# チェッカーボードの認識
detect() # 認識結果が box として表示される
objfrm = detect() # 実機の場合は ROS の識別器から tf を受信
                # simulation の場合は直接、物体位置を読む

# Link に沿った座標変換
# 認識結果は、カメラ=>対象物の座標変換であるので、
# 世界座標=>対象物の座標変換に直す
# detect() 関数の中で行っている処理
r.Thd_leye # 頭リンク => 左目カメラへの変換
Tleye_obj = rr.detect()
Twld_hd = r.get_link('HEAD_JOINT1_Link').where()
Twld_obj = Twld_hd * r.Thd_leye * Tleye_obj

# 実機の場合

```

```

hand_cam_demo()

# シミュレータで実行するときは、先に手が届く場所に箱を置いておく
# 'box0' => marker 0, 'box1' => marker 1 に対応
putbox(name='box0', vaxis='y')
putbox(name='box1', vaxis='y')
hand_cam_demo()

# もう一度実行したいとき
env.delete_object('box0')
putbox(name='box0', vaxis='y')

# ハンドカメラは AR-toolkit による marker 認識（複数物体対応）
# 返り値は（マーカ番号、カメラ=>マーカの座標変換）のリスト
# non-blocking で最新の認識結果を返す
# 過去 thre [sec] 以内に認識に成功していないマーカは返さない
# デフォルトは thre = 0.5[sec]

lupus@ roslaunch Sense sense_lhand_ar.launch
rr.detect(camera='lhand') # 左手カメラによる認識

# 平行グリッパの間隔指定把持
r.grasp(width=65, hand='right') # 指の間隔が width[mm] になるように指を動かす
r.grasp(65) # これでも同じ

# アプローチ距離の指定
pl.reaching_plan(objfrm, approach_distance=80) # 少し遠くからアプローチ

# 動作計画（仕様が変わる可能性大）
traj = test_plan() # RRT-connect による動作計画
show_traj(traj) # 軌道の表示
opttraj = pl.optimize_trajectory(traj) # 軌道の最適化
show_traj(opttraj)
opttraj = pl.optimize_trajectory(opttraj) # さらにスムージング

# 干渉チェックには PQP を利用
# ロボットのモデル VRML そのまま

# 干渉チェック対象物の追加例
r.add_collision_object(env.get_object('table top'))

```