(VPython版)HiroNX動作生成モジュール 使用説明書

東京大学　情報理工学系研究科　稲葉研究室

平成 24 年 2 月 14 日

1　概要

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．

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

2　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 10.04LTC 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ることが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).

3　開発・動作環境

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

4　サンプルプログラムの実行

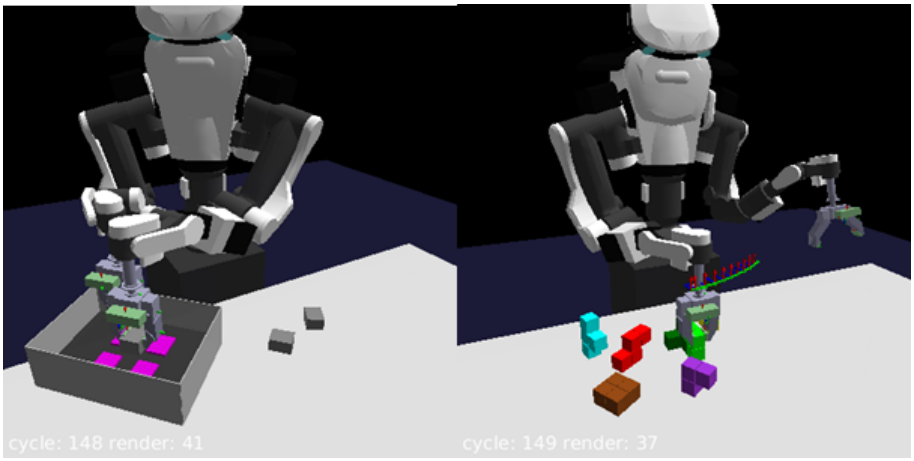

図 1: 動作生成モジュール

```
$ cd iv_scenario/src
$ ipython test.py
>>> test1()

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
$ ipython demo_wexpo.py
>>> demo(False)
```

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 でインタープリタを抜けた後，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 を送ることで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．

5 HiroNX 実機の利用

本モジュールは，双腕ロボットの制御コマンド共通インタフェースにを利用して，実機に動作コマンドを送ります．したがって，実機を利用する場合は，HinoNXInterface モジュールを必要とします．

5.1 HiroNXInterface

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．

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

5.2 動作確認

最初に HiroNXInterface 制御モジュールを起動，activate，接続し，利用可能な状態にします（詳細は HiroNXInterface のドキュメントを参照してください）．添付のスクリプトでも起動できます．

```
$ cd iv_plan/scripts
$ ./run_hironx_interface.sh
```

iv_plan/scripts 以下のスクリプトは，ローカルのファイルシステム非依存で各モジュールのパスを検出するため，ROS のツールである rospack を使います．rospack を利用できない場合は適宜，修正してください．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．

HiroNXIntarface の起動および，ロボットのジョイントキャリブレーションが終了した状態で，実機との通信テストを行います．

```
$ cd iv_scenario/src
demo_wexpo.pyのreal_robotをTrueに変更します.

$ ipython demo_wexpo.py
>>> rr.get_joint_angles() # 実機の関節角度列を取得
>>> 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(実機が動くので注意)
```

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)

動作生成側から HiroNX 体内の RTC や認識などのモジュールにアクセスする必要があります．そのための検索先ネームサーバは Python プログラムを起動するディレクトリにある rtc.conf から取得します．以下のように iv_scenario ディレクトリで python プログラムを起動する場合，iv_scenario/rtc.conf が読み込まれます．外部モジュールと通信ができない場合は，rtc.conf を確認してください．また，接続先 RT コンポーネントのパスは，interface_wexpo.py に定義されています．portdefs を利用している環境にあわせて修正してください．

6 設計方針

7 RT コンポーネントとしての利用

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.
(変更中)

```
# 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

# 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
# scequencer, 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'))
```