GMC-7004, SUJET SPÉCIAL (GÉNIE MÉCANIQUE). SYSTÈME D'EXPLOITATION DE ROBOT, ROS, ENVIRONNEMENT DE SIMULATION.

Philippe Lebel
Université Laval, philippe.lebel.4@ulaval.ca

Résumé

L'objectif de ce sujet spécial est de rendre compte des connaissances acquises pour l'utilisation de différents outils de développement dans le domaine de la robotique. Les différentes problématiques abordées dans ce rapport sont : la communication ordinateur-robot sans l'entremise d'un noeud de calcul, l'installation et utilisation de la plateforme ROS et la mise sur pied d'un environnement de simulation robotique en utilisant des outils tels que Gazebo, V-rep et URSIM. L'objectif étant de démontrer l'acquisition de connaissances mais aussi de permettre à d'autres d'utiliser cette synthèse, le présent rapport tentera d'exemplifier et aura une approche pratique sur l'utilisation des outils mentionnés.

Dans une démarche expérimentale, il est souvent requis d'implémenter les algorithmes développés, dans un environnement tel que Matlab, sur des systèmes robotiques déjà existants. La solution couramment utilisée est d'employer un nœud de calcul QNX sur lequel RTLAB est installé. Ceci requiert un processus d'adaptation autant au niveau de l'utilisation du logiciel qu'au niveau de l'adaptation du code Matlab, python c++ déjà développé. Bien souvent, le processus peut s'échelonner sur plusieurs semaines. Ceci motive alors la recherche d'alternatives permettant l'utilisation directe du code sur un système robotique. L'objectif étant de permettre l'utilisation de différents langage de programmation pour minimiser le temps d'adaptation. Une solution permettant l'utilisation du code Matlab, python, c++ et autre est alors mise sur pied.

1 Mise en contexte

Les solutions présentées dans ce rapport répondent toutes au besoin d'utiliser le code directement sur un montage robotique. Cependant, elle présentent toutes différents avantage et inconvénient. Comme processus de validation, le solutions ont été testées en répondant à la même problématique : l'utilisation d'un bras Jaco de Kinova directement par l'entremise de la connexion usb.

2 Définition des objectifs

Différents critères doivent être rencontrés pour qu'une solution soit acceptable et remplace le système déjà en place.

1. Permettre la communication bidirectionnelle.
2. Être compatible avec plusieurs systèmes robotiques.
3. Permettre une communication avec un délais de moins de 10 ms. (100 Hz)
4. Permettre une grande flexibilité dans le type de commandes qu'un utilisateur peut envoyer. (commandes articulaires, cartésiennes, force, etc.)

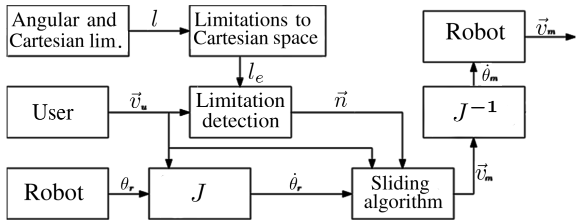

FIGURE 1 – General structure of the algorithm.

3 Solution de contrôle du bras Jaco développée sous Windows

Sachant que Windows est le système d'exploitation majoritairement utilisé au sein du laboratoire de robotique de l'université Laval, le besoin de développer une solution compatible Windows a été identifié. La stratégie employée est représenté dans la Figure 1. Du point de vue global, la structure du système se résume en une chaine de communication UDP entre différents programmes et le robot. Cependant, dans le cas de l'implémentation avec le robot Jaco, une API, *Applications Programming Interface*, gère l'interface entre les actionneurs et les requêtes communiquées au robot via la connection USB. Dans le cas du Jaco, cette API permet l'utilisation de différentes fonctions rédigées en c++. Ceci cause problème lorsqu'un développeur tente d'utiliser un programme écrit en Matlab ou en python pour commander les articulations du robots. La solution à cet inconvénient est d'initier une instance Matlab via un un script c++. Cette instance Matlab pourra ensuite exécuter des commandes que le script c++ lui envoi, tel que l'exécution de fonctions Matlab. Par la suite, il suffit d'utiliser directement l'API de kinova pour envoyer des commandes.

3.1 Exemple d'implémentation

Différents composants doivent être mis en place pour réaliser l'implémentation. Le présent exemple permet de configure un projet visual studio pour contrôler un bras Jaco en communiquant avec Matlab.

3.1.1 Visual Studio

Il est possible de télécharger VisualStudio sur le site web des développeurs.

3.1.2 Création d'un projet

Un projet VisualStudio peut être créé en cliquant sur *file/new/project* dans la fenêtre de visual studio. Ensuite, sélectionner *Visual c++* dans le menu gauche de la nouvelle fenêtre et cliquez sur *Empty Project*. Dans le bas de la fenêtre, il est possible de nommer le projet et sélectionner un répertoire. Le répertoire choisi sera référé comme étant le *workspace* ou *project direcotry* dans la pluspart des tutoriels en anglais. Tout ficher placé dans ce répertoire fera partie du projet et pourra être référé dans les scripts.

3.1.3 Définition du fichier principal du projet

Pour être en mesure d'écrire le script, un fichier source doit être créé. Pour créer ce ficher, il faut cliquer droit sur l'item *Source Files* dans l'arbre du projet situé à la droite de la fenêtre Visual

Studio. Ensuite, il faut sélectionner *add/new item* et cliquer sur *c++ File*. Il est ensuite possible de nommer le fichier dans le bas de la fenêtre.

3.1.4 Utilisation d'un joystick

Pour être en mesure d'envoyer des commandes avec un joystick, la librairie SDL2 doit être téléchargée. Bien sûr, si le projet ne requiert pas de contrôleur joystick, cette étape n'est pas nécessaire. Après avoir décompressé le dossier téléchargé, lire le fichier VisualC.html pour les instructions sur la démarche à suivre pour utiliser la libraire (en particulier la partie parlant de l'utilisation avec Visual Studio). Les détails de l'utilisation pratique de cette librairie seront couverts dans une section subséquentes.

3.1.5 Lien avec Matlab

La description de cette partie se fera en trois temps. Dans un premier temps, une explication des différentes composantes devant être mises en places pour monter l'environnement de développement sera présentée. Ensuite, l'installation des pilotes pour l'envoie de commandes au bras Jaco est expliquée. Finalement, une description de différentes parties du script permettant le contrôle du bras Jaco sera fait.

Environnement de travail Les différentes librairies, exécutables et les entêtes (*headers*) nécessaires à l'utilisation de Matlab via un script c++ sont disponible dans le répertoire retourné lorsque l'on envoie les commandes suivantes dans la console Matlab :

1. fullfile(matlabroot,'bin','win64').
2. fullfile(matlabroot,'extern','include'),
3. fullfile(matlabroot,'extern','lib'),

Pour les utiliser dans un projet Visual Studio, il est nécessaire de suivre les étapes suivantes :

— Il est tout d'abord très important de s'assurer que l'option x64 est sélectionnée dans le menu déroulant dans le haut et au centre de la fenêtre principale de Visual Studio.
— Aller dans *Project\'nom-de-votre-projet' Properties...* dans le haut de la fenêtre principale de Visual Studio.
— Naviguer dans l'onglet Debugging de la fenêtre venant de s'ouvrir.
— Écrire *PATH=* suivi du résultat obtenu lors de l'envoie de la commande 1 dans le console Matlab, en omettant les guillemets,(devrait ressembler à *C :\Program Files\MATLAB\R2017b\bin\win64*) dans le section *Environnement*. Écrire Directement à la suite de la dernière entrée : *;%PATH%*
— Naviguer dans l'onglet VC++ Directories de la même fenêtre.
— **Pour les prochaines étapes** : Si un item était déjà présent dans cette ligne, il faut simplement séparer les entrées avec un " ;".
— Insérer le résultat obtenu lors de l'envoie de la commande 2 dans le console Matlab, en omettant les guillemets, dans le section *Include Directories*.
— Insérer le résultat obtenu lors de l'envoie de la commande 3 dans le console Matlab, en omettant les guillemets, dans le section *Library Directories*.
— Naviguer dans l'onglet *Linker* et le sous-onglet *Input* de la même fenêtre.
— Entrer au début d'*Additional Dependencies* l'entrée suivante : *libmx.lib ;libeng.lib ;libmex.lib ;libmat.lib ;*.

Pour tester l'environnement de travail, un script c++ est inclue avec les version récentes de Matlab. Il est accessible en entrant la commande :

— *edit([matlabroot\extern\examples\eng_mat\engdemo.cpp'])*;

Il faut seulement copier-coller l'entièreté du script dans le fichier source (.cpp) du projet Visual Studio. Le clic sur le bouton *Local Windows Debugger* lancera la compilation et devrait faire apparaitre un invité de commande. La première compilation est normalement beaucoup plus longue que les compilations subséquente.

Installation de Kinova SDK Pour être en mesure d'envoyer des commandes au robot jaco, il est nécessaire d'installer les pilotes fournis sur le site web de Kinova. Suivant l'installation de Kinova SDK, des directives doivent être suivies pour que le robot soit bien reconnu par le système d'exploitation. Ces directives sont disponible dans le répertoire d'installation de *KinovaSDK\Guides\Kinova SDK-User Guide* (souvent situé dans *Program Files x86*). Pour tester l'envoi de commandes au robot et valider l'installation, une interface graphique est disponible dans *KinovaSDK\GUI*. Des exemples de scripts c++ sont aussi disponibles dans le répertoire *KinovaSDK\Examples*.

À la suite du test, il est maintenant possible de combiner les informations des deux tutoriels pour faire fonctionner des scripts matlab, transférer des données de l'environnement de travail matlab vers le *stack* du programme c++ et vice versa.

Dans cette section, un survol de l'implémentation de l'environnement de développement permettant le contrôle du bras Jaco sous Windows avec l'option d'utiliser des scripts Matlab a été présenté. Bien que seulement le bras Jaco a été mentionné, il est toutefois possible d'utiliser les API *c++* d'autres systèmes robotiques tels que les bras de Universal Robot. Cependant, une autre option pour contrôler le bras UR5 sous windows est disponible. Cette dernière est présentée dans la section suivante.

4 Solution de contrôle du bras UR5 développée sous Windows

Le contrôle du bras UR5 sous windows présente quelques défis puisque la majorité des outils de développement pour cet équipement sont développés sur la plateforme Linux. Sachant que cette plateforme est souvent méconnue, la solution développée pour interagir avec le bras tentera de minimiser le travail requis avec Linux et fera le lien entre Matlab, URsim et un logiciel de visualisation ce nommant VRep. Ce dernier permet l'insertion et la visualisation de différents objets dans l'environnement du robot puisque URsim ne peut que représenter le robot lui-même. Tout d'abord, cette section expliquera l'installation de URsim et les différentes fonctionnalités utiles à la mise en place de l'infrastructure. Ensuite, le moyen de communication entre Matlab et URsim sera décrit. Finalement, l'interaction avec le logiciel de visualisation VRep sera expliqué.

4.1 URsim et son utilisation

URsim permet l'interface avec un robot virtuel ou un robot réel connecté sur le réseau. Ce logiciel permet la programmation de routines de base ainsi que la définition de fonctions plus complexe permettant, entre autre, la communication UDP avec d'autre logiciels. URsim, en soit, n'est disponible que sur Linux. Cependant, sur le site web de l'entreprise Universal Robots, une machine virtuelle sur laquelle URsim est déjà installée est disponible. Ceci permet de conserver l'environnement de développement Windows en exécutant URsim dans une machine virtuelle Linux. Le logiciel avec lequel cette machine virtuelle a été utilisée pour l'environnement de développement a été VmWare. Après avoir suivi les instructions d'installation fournies avec les fichiers disponibles

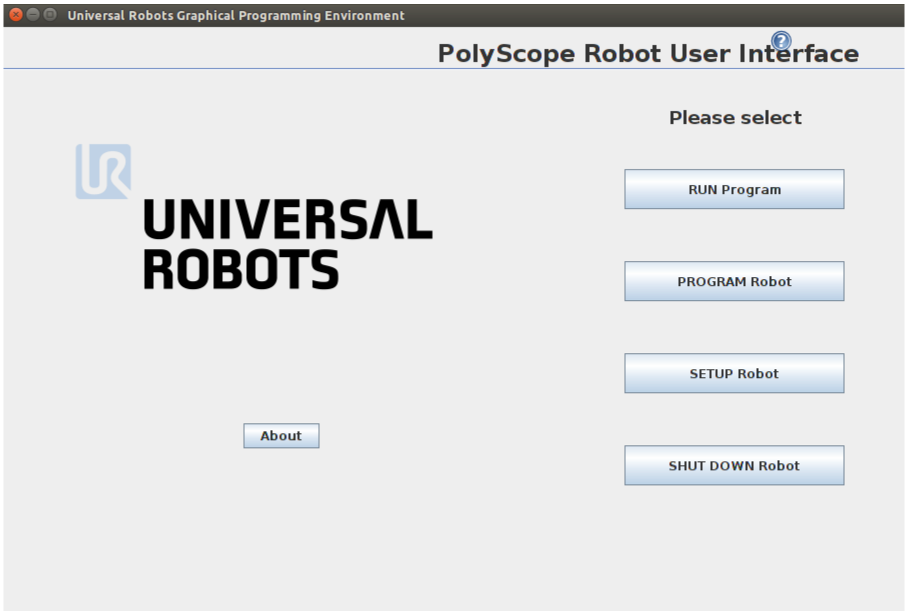

FIGURE 2 – Interface initiale d'URsim

sur le site web de Universal Robots, il est possible de lancer URsim. L'interface initiale d'URsim est présente dans la Figure suivante :

— Le bouton *RUN Program* permet de faire fonctionner des routines contenant différentes fonctions. Ces routines sont communément appelées *URscript*. Ces scripts peuvent variés en complexité allant d'une simple séquence de positions à atteindre jusqu'à un programme complexe gérant de multiples sockets de communications UDP et répondant à des appels de fonctions listées dans la librairie d'URsim.
— Le bouton *PROGRAM Robot* permet la rédaction ou la modification d'URscripts
— Le bouton *SETUP robot* permet la modification de paramètres tels que l'adresse IP d'un robot réel qu'un utilisateur voudrait contrôler via l'instance d'URsim installée sur la machine virtuelle.

Quand un utilisateur charge un URscript, il appui alors sur *PROGRAM ROBOT*, puis sur *Load Program* (ou sur *Empty program* si il n'y a pas de programmes déjà disponibles). Une fois cette opération complétée la fenêtre présente à la Figure 3. Ceci constitue l'interface de programmation d'URscripts. Un manuel de programmation URscript est inclue dans l'appendice A, cependant, pour le fonctionnement de l'environnement de développement tel que présenté, aucune programmation URscript ne sera nécessaire.

4.2 Lien URsim-Matlab

Comme mentionné précédemment, les URscripts peuvent gérer des sockets de communication UDP et utiliser des fonctions tirées de la librairie d'URsim. C'est de cette manière qu'un URscript a été rédigé de manière a répondre à certaines commandes envoyée sous format texte par Matlab. Les fichiers URscripts en question sont disponible dans le dossier *programs* du dépot github suivant. Pour rendre ces URscripts disponibles pour URsim, il faut seulement remplacer le dossier *programs* présent dans le répertoire d'instalation d'URsim par celui téléchargé depuis le répertoire github mentionné plus haut. La figure 4 représente grosso modo l'endroit dans lequel il faut remplacer le

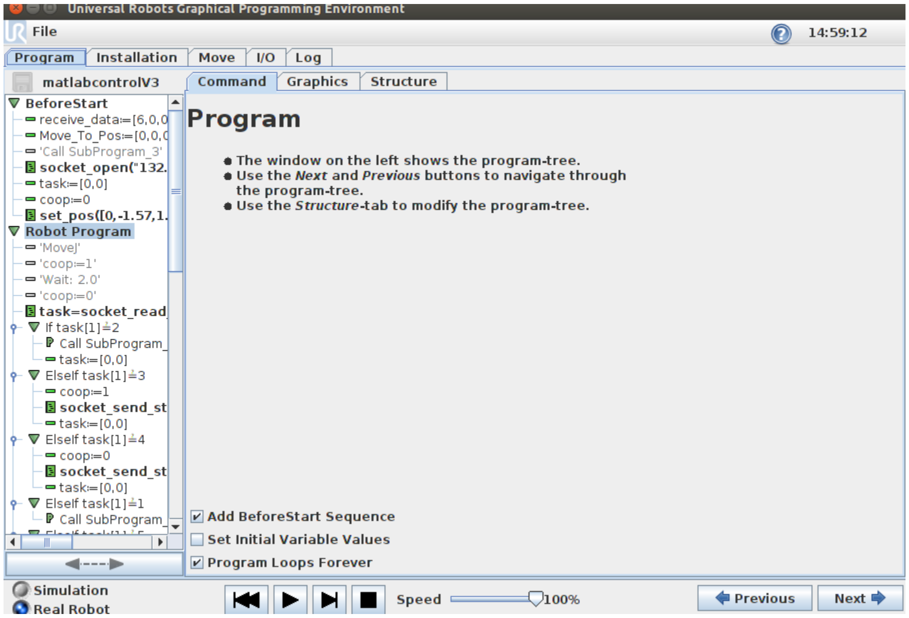

FIGURE 3 – Interface de programmation d'URsim

dossier *programs*.

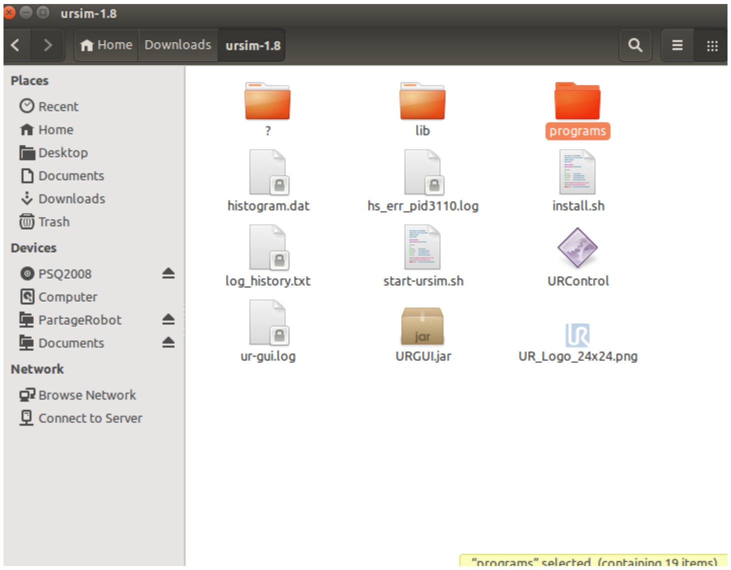

FIGURE 4 – Interface de programmation d'URsim

Références

[1] First Reference...

A
Manuel de programmation URscript

UNIVERSAL ROBOTS

The URScript Programming Language

Version 3.1
January 28, 2015

Contents

1 The URScript Programming Language

1.1 Introduction

The Universal Robot can be controlled a three different levels: The *Graphical User-Interface Level*, the *Script Level* and the *C-API Level*. URScript is the robot programming languange used to control the robot at the *Script Level*. Like any other programming language URScript has variables, types, flow of control statements, function etc. In addition URScript has a number of built-in variables and functions which monitors and controls the I/O and the movements of the robot.

1.2 Connecting to URControl

URControl is the low-level robot controller running on the Mini-ITX PC in the controller cabinet. When the PC boots up URControl starts up as a daemon (like a service) and PolyScope User Interface connects as a client using a local TCP/IP connection.

Programming a robot at the *Script Level* is done by writing a client application (running at another PC) and connecting to URControl using a TCP/IP socket.

- **hostname**: ur-xx (or the ip-adresse found in the about dialog-box in PolyScope if the robot is not in dns.)
- **port**: 30002

When connected URScript programs or commands are sent i clear text on the socket. Each line is terminated by "\n".

1.3 Numbers, Variables and Types

The syntax of arithmetic expressions in URScript is very standard:

```
1+2-3
4*5/6
(1+2)*3/(4-5)
```

In boolean expressions the boolean operators are spelled out:

```
True or False and (1 == 2)
1 > 2 or 3 != 4 xor 5 < -6
not 42 >= 87 and 87 <= 42
```

Variable assignment is done using the equal sign "=":

```
foo = 42
bar = False or True and not False
baz = 87-13/3.1415
hello = \q{Hello, World!}
```

```
l = [1,2,4]
target = p[0.4,0.4,0.0,0.0,3.14159,0.0]
```

The fundamental type of a variable is deduced from the first assignment of the variable. In the example above `foo` is an `int` and `bar` is a `bool`. `target` is a pose, a combination of a position and orientation.

The fundamental types are:

- `none`
- `bool`
- `number` - either `int` or `float`
- `pose`
- `string`

A pose is given as `p[x,y,z,ax,ay,az]`, where `x,y,z` is the position of the TCP, and `ax,ay,az` is the orientation of the TCP, given in axis-angle notation.

1.4 Flow of Control

The flow of control of a program is changed by `if-statements`:

```
if a > 3:
  a = a + 1
elif b < 7:
  b = b * a
else:
  a = a + b
end
```

and `while-loops`:

```
l = [1,2,3,4,5]
i = 0
while i < 5:
  l[i] = l[i]*2
end
```

To stop a loop prematurely the `break` statement can be used. Similarly the `continue` statement can be used to pass control to the next iteration of the nearest enclosing loop.

1.4.1 Special keywords

- `halt` Terminates the program

- `return` Returns from a function

1.5 Function

A function is declared as follows:

```
def add(a, b):
  return a+b
end
```

The function can then be called like this:

```
result = add(1, 4)
```

It is also possible to give function arguments default values:

```
def add(a=0,b=0):
  return a+b
end
```

URScript also supports named parameters.

1.6 Remote Procedure Call (RPC)

Remote Procedure Calls (RPCs) are similar to normal Function calls, except that the function is defined and executed remotely. On the remote site, the RPC function being called must exist with the same number of parameters and corresponding types (together the function's signature). If the function is not defined remotely, it will stop the program execution. The controller uses the XMLRPC standard to send the parameters to the remote site and retrieve the result(s). During a RPC call the controller waits for the remote function to complete. The XMLRPC standard is among others supported by C++ (xmlrpc-c library), Python and Java.

On the UR script side, a program to initialize a camera, take a snapshot and retrieve a new target pose would look something like:

```
camera = rpc_factory("xmlrpc", "http://127.0.0.1/RPC2")
if (! camera.initialize("RGB")):
  popup("Camera was not initialized")
camera.takeSnapshot()
target = camera.getTarget()
...
```

First the `rpc_factory` (see `Interfaces section`) creates a XMLRPC connection to the specified "remote" server. The `camera` variable is the handle for the remote function calls. The user needs to initialize the camera and therefore calls `camera.initialize("RGB")`. The function returns a boolean value to indicate if the request was successful. In order to find a target position (somehow) the camera first needs to take a picture, hence the `camera.takeSnapshot()` call. After the snapshot was taken the image analysis in the remote site figures out the location of the target. Then the program asks for the exact target location with the function call `target = camera.getTarget()`. On return the `target` variable will be assigned the result. The camera.`initialize("RGB")`, `takeSnapshot() and getTarget()` functions are the responsibility of the RPC server.

The `Technical support website` contains more examples of XMLRPC servers.

1.7 Scoping rules

A urscript program is declared as a function without parameters:

```
def myProg():

end
```

Every variable declared inside a program exits at a global scope, except when they are declared inside a function. In that case the variable are local to that function. Two qualifiers are available to modify this behaviour. The `local` qualifier tells the runtime to treat a variable inside a function, as being truly local, even if a global variable with the same name exists. The `global` qualifier forces a variable declared inside a function, to be globally accessible.

In the following example, `a` is a global variable, so the variable inside the function is the same variable declared in the program:

```
def myProg():

  a = 0

  def myFun():
    a = 1
    return a
  end

  r = myFun()
end
```

In this next example, `a` is declared `local` inside the function, so the two variables are different, even though they have the same name:

```
def myProg():

  a = 0

  def myFun():
    local a = 1
    return a
  end

  r = myFun()
```

```
end
```

Beware that the global variable is no longer accessible from within the function, as the local variable masks the global variable of the same name.

1.8 Threads

Threads are supported by a number of special commands.

To declare a new thread a syntax similar to the declaration of functions are used:

```
thread myThread():
  # Do some stuff
  return
end
```

A couple of things should be noted. First of all, a thread cannot take any parameters, and so the parentheses in the declaration must be empty. Second, although a return statement is allowed in the thread, the value returned is discarded, and cannot be accessed from outside the thread. A thread can contain other threads, the same way a function can contain other functions. Threads can in other words be nested, allowing for a thread hierarchy to be formed.

To run a thread use the following syntax:

```
thread myThread():
  # Do some stuff
  return
end

thrd = run myThread()
```

The value returned by the `run` command is a handle to the running thread. This handle can be used to interact with a running thread. The run command spawns off the new thread, and then goes off to execute the instruction following the `run` instruction.

To wait for a running thread to finish, use the `join` command:

```
thread myThread():
  # Do some stuff
  return
end

thrd = run myThread()
```

```
join thrd
```

This halts the calling threads execution, until the thread is finished

executing. If the thread is already finished, the statement has no effect.

To kill a running thread, use the `kill` command:

```
thread myThread():
  # Do some stuff
  return
end

thrd = run myThread()

kill thrd
```

After the call to kill, the thread is stopped, and the thread handle is no longer valid. If the thread has children, these are killed as well.

To protect against race conditions and other thread related issues, support for critical sections are provided. A critical section ensures that the code it encloses is allow to finish, before another thread is allowed to run. It is therefore important that the critical section is kept as short as possible. The syntax is as follows:

```
thread myThread():
  enter_critical
  # Do some stuff
  exit_critical
  return
end
```

1.8.1 Threads and scope

The scoping rules for threads are exactly the same, as those used for functions. See 1.7 for a discussion of these rules.

1.8.2 Thread scheduling

Because the primary purpose of the urscript scripting language is to control the robot, the scheduling policy is largely based upon the realtime demands of this task.

The robot must be controlled a frequency of 125 Hz, or in other words, it must be told what to do every 0.008 second (each 0.008 second period is called a frame). To achieve this, each thread is given a "physical" (or robot) time slice of 0.008 seconds to use, and all threads in a runnable state is then scheduled in a round robin[1] fashion. Each time a thread is scheduled, it can use a piece of its time slice (by executing instructions that control the robot), or it can execute instructions that do not control the robot, and therefore do not use any "physical" time. If a thread uses up its entire time slice, it is placed in a non-runnable state, and is not allowed to run until the next frame starts. If a thread does not use its time slice within a frame, it is expected to switch to a non-runnable state before the end of the frame[2]. The reason for this state switching can be a join instruction or simply because the thread terminates.

It should be noted that even though the `sleep` instruction does not control the robot, it still uses "physical" time. The same is true for the `sync` instruction.

1.9 Program Label Messages

A special feature is added to the script code, to make it simple to keep track of which lines are executed by the runtime machine. An example *Program Label Message* in the script code looks as follows;

```
sleep(0.5)
$ 3 \q{AfterSleep}
set_standard_digital_out(7, True)
```

After the the Runtime Machine executes the sleep command, it will send a message of type `PROGRAM_LABEL` to the latest connected primary client. The message will hold the number *3* and the text *AfterSleep*. This way the connected client can keep track of which lines of codes are being executed by the Runtime Machine.

2 Module motion

This module contains functions and variables built into the URScript programming language.

URScript programs are executed in real-time in the URControl RuntimeMachine (RTMachine). The RuntimeMachine communicates with the robot with a frequency of 125hz.

Robot trajectories are generated online by calling the move functions `movej`, `movel` and the speed functions `speedj`, `speedl`.

Joint positions (q) and joint speeds (qd) are represented directly as lists of 6 Floats, one for each robot joint. Tool poses (x) are represented as poses also consisting of 6 Floats. In a pose, the first 3 coordinates is a position vector and the last 3 an axis-angle (*http://en.wikipedia.org/wiki/Axis_angle*).

[1]Before the start of each frame the threads are sorted, such that the thread with the largest remaining time slice is to be scheduled first.

[2]If this expectation is not met, the program is stopped.

2.1 Functions

conveyor_pulse_decode(*type, A, B*)

Tells the robot controller to treat digital inputs number A and B as pulses for a conveyor encoder. Only digital input 0, 1, 2 or 3 can be used.

```
>>> conveyor_pulse_decode(1,0,1)
```

This example shows how to set up quadrature pulse decoding with input A = digital_in(0) and input B = digital_in(1)

```
>>> conveyor_pulse_decode(2,3)
```

This example shows how to set up rising and falling edge pulse decoding with input A = digital_in(3). Note that you do not have to set parameter B (as it is not used anyway).

Parameters

`type`: An integer determining how to treat the inputs on A and B

0 is no encoder, pulse decoding is disabled.

1 is quadrature encoder, input A and B must be square waves with 90 degree offset. Direction of the conveyor can be determined.

2 is rising and falling edge on single input (A).

3 is rising edge on single input (A).

4 is falling edge on single input (A).

The controller can decode inputs at up to 40kHz

`A`: Encoder input A, values of 0-3 are the digital inputs 0-3.

`B`: Encoder input B, values of 0-3 are the digital inputs 0-3.

end_force_mode()

Resets the robot mode from force mode to normal operation.

This is also done when a program stops.

end_freedrive_mode()

Set robot back in normal position control mode after freedrive mode.

end_teach_mode()

Set robot back in normal position control mode after freedrive mode.

force_mode(*task_frame*, *selection_vector*, *wrench*, *type*, *limits*)

Set robot to be controlled in force mode

Parameters

`task_frame:`	A pose vector that defines the force frame relative to the base frame.
`selection_vector:`	A 6d vector that may only contain 0 or 1. 1 means that the robot will be compliant in the corresponding axis of the task frame, 0 means the robot is not compliant along/about that axis.
`wrench:`	The forces/torques the robot is to apply to its environment. These values have different meanings whether they correspond to a compliant axis or not. Compliant axis: The robot will adjust its position along/about the axis in order to achieve the specified force/torque. Non-compliant axis: The robot follows the trajectory of the program but will account for an external force/torque of the specified value.
`type:`	An integer specifying how the robot interprets the force frame. 1: The force frame is transformed in a way such that its y-axis is aligned with a vector pointing from the robot tcp towards the origin of the force frame. 2: The force frame is not transformed. 3: The force frame is transformed in a way such that its x-axis is the projection of the robot tcp velocity vector onto the x-y plane of the force frame. All other values of type are invalid.
`limits:`	A 6d vector with float values that are interpreted differently for compliant/non-compliant axes: Compliant axes: The limit values for compliant axes are the maximum allowed tcp speed along/about the axis. Non-compliant axes: The limit values for non-compliant axes are the maximum allowed deviation along/about an axis between the actual tcp position and the one set by the program.

freedrive_mode()

Set robot in freedrive mode. In this mode the robot can be moved around by hand in the same way as by pressing the "freedrive" button. The robot wuill not be able to follow a trajectory (eg. a movej) in this mode.

get_conveyor_tick_count()

Tells the tick count of the encoder, note that the controller interpolates tick counts to get more accurate movements with low resolution encoders

Return Value

The conveyor encoder tick count

movec(*pose_via*, *pose_to*, *a*=`1.2`, *v*=`0.25`, *r*=`0`)

Move Circular: Move to position (circular in tool-space)

TCP moves on the circular arc segment from current pose, through pose_via to pose_to. Accelerates to and moves with constant tool speed v.

Parameters

`pose_via`: path point (note: only position is used). (pose_via can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)

`pose_to`: target pose (pose_to can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)

`a`: tool acceleration (m/s^2)

`v`: tool speed (m/s)

`r`: blend radius (of target pose) (m)

movej(*q*, *a*=`1.4`, *v*=`1.05`, *t*=`0`, *r*=`0`)

Move to position (linear in joint-space) When using this command, the robot must be at standstill or come from a movej og movel with a blend. The speed and acceleration parameters controls the trapezoid speed profile of the move. The t parameters can be used in stead to set the time for this move. Time setting has priority over speed and acceleration settings. The blend radius can be set with the r parameters, to avoid the robot stopping at the point. However, if he blend region of this mover overlaps with previous or following regions, this move will be skipped, and an 'Overlapping Blends' warning message will be generated.

Parameters

`q`: joint positions (q can also be specified as a pose, then inverse kinematics is used to calculate the corresponding joint positions)

`a`: joint acceleration of leading axis (rad/s^2)

`v`: joint speed of leading axis (rad/s)

`t`: time (S)

`r`: blend radius (m)

movel(*pose*, *a*=`1.2`, *v*=`0.25`, *t*=`0`, *r*=`0`)

Move to position (linear in tool-space)

See movej.

Parameters

`pose`: target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)

`a`: tool acceleration (m/s^2)

`v`: tool speed (m/s)

`t`: time (S)

`r`: blend radius (m)

movep(*pose*, *a*=`1.2`, *v*=`0.25`, *r*=`0`)

Move Process

Blend circular (in tool-space) and move linear (in tool-space) to position. Accelerates to and moves with constant tool speed v.

Parameters

- `pose:` target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)
- `a:` tool acceleration (m/s^2)
- `v:` tool speed (m/s)
- `r:` blend radius (m)

servoc(*pose*, *a*=`1.2`, *v*=`0.25`, *r*=`0`)

Servo Circular

Servo to position (circular in tool-space). Accelerates to and moves with constant tool speed v.

Parameters

- `pose:` target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)
- `a:` tool acceleration (m/s^2)
- `v:` tool speed (m/s)
- `r:` blend radius (of target pose) (m)

servoj(*q*, *a*, *v*, *t*=`0.008`, *lookahead_time*=`0.1`, *gain*=`300`)

Servo to position (linear in joint-space)

Servo function used for online control of the robot. The lookahead time and the gain can be used to smoothen or sharpen the trajectory.

Note: A high gain or a short lookahead time may cause instability. Prefered use is to call this function with a new setpoint (q) in each time step (thus the default t=0.008)

Parameters

`q:`	joint positions (rad)
`a:`	NOT used in current version
`v:`	NOT used in current version
`t:`	time (S)
`lookahead_time:`	time (S), range (0.03,0.2) smoothens the trajectory with this lookahead time
`gain:`	proportional gain for following target position, range (100,2000)

set_conveyor_tick_count(*tick_count, absolute_encoder_resolution*=0)

Tells the robot controller the tick count of the encoder. This function is useful for absolute encoders, use conveyor_pulse_decode() for setting up an incremental encoder. For circular conveyors, the value must be between 0 and the number of ticks per revolution.

Parameters

`tick_count:`	Tick count of the conveyor (Integer)
`absolute_encoder_resolution:`	Resolution of the encoder, needed to handle wrapping nicely. (Integer) 0 is a 32 bit signed encoder, range (-2147483648 ; 2147483647) (default) 1 is a 8 bit unsigned encoder, range (0 ; 255) 2 is a 16 bit unsigned encoder, range (0 ; 65535) 3 is a 24 bit unsigned encoder, range (0 ; 16777215) 4 is a 32 bit unsigned encoder, range (0 ; 4294967295)

set_pos(*q*)

Set joint positions of simulated robot

Parameters

`q:` joint positions

speedj(*qd, a, t_min*)

Joint speed

Accelerate to and move with constant joint speed

Parameters

`qd:` joint speeds (rad/s)

`a:` joint acceleration (rad/s^2) (of leading axis)

`t_min:` minimal time before function returns

speedl(*xd, a, t_min*)

Tool speed

Accelerate to and move with constant tool speed

http://axiom.anu.edu.au/~roy/spatial/index.html

Parameters

`xd:` tool speed (m/s) (spatial vector)

`a:` tool acceleration (/s^2)

`t_min:` minimal time before function returns

stop_conveyor_tracking()

Makes robot movement (movej etc.) follow the original trajectory instead of the conveyor specified by track_conveyor_linear() or track_conveyor_circular().

stopj(*a*)

Stop (linear in joint space)

Decellerate joint speeds to zero

Parameters

`a:` joint acceleration (rad/s^2) (of leading axis)

stopl(*a*)

Stop (linear in tool space)

Decellerate tool speed to zero

Parameters

`a:` tool accleration (m/s^2)

teach_mode()

Set robot in freedrive mode. In this mode the robot can be moved around by hand in the same way as by pressing the "freedrive" button. The robot wuill not be able to follow a trajectory (eg. a movej) in this mode.

track_conveyor_circular(*center*, *ticks_per_revolution*, *rotate_tool*)

Makes robot movement (movej() etc.) track a circular conveyor.

```
>>> track_conveyor_circular(p[0.5,0.5,0,0,0,0],500.0, false)
```

The example code makes the robot track a circular conveyor with center in p(0.5,0.5,0,0,0,0) of the robot base coordinate system, where 500 ticks on the encoder corresponds to one revolution of the circular conveyor around the center.

Parameters

`center:`	Pose vector that determines the center the conveyor in the base coordinate system of the robot.
`ticks_per_revolution:`	How many tichs the encoder sees when the conveyor moves one revolution.
`rotate_tool:`	Should the tool rotate with the coneyor or stay in the orientation specified by the trajectory (movel() etc.).

track_conveyor_linear(*direction*, *ticks_per_meter*)

Makes robot movement (movej() etc.) track a linear conveyor.

```
>>> track_conveyor_linear(p[1,0,0,0,0,0],1000.0)
```

The example code makes the robot track a conveyor in the x-axis of the robot base coordinate system, where 1000 ticks on the encoder corresponds to 1m along the x-axis.

Parameters

`direction:`	Pose vector that determines the direction of the conveyor in the base coordinate system of the robot
`ticks_per_meter:`	How many tichs the encoder sees when the conveyor moves one meter

2.2 Variables

Name	Description
__package__	**Value:** 'Motion'
a_joint_default	**Value:** 1.4
a_tool_default	**Value:** 1.2
v_joint_default	**Value:** 1.05
v_tool_default	**Value:** 0.25

3 Module internals

3.1 Functions

force()

Returns the force exceted at the TCP

Return the current externally excerted force at the TCP. The force is the norm of Fx, Fy, and Fz calculated using get_tcp_force().

Return Value

The force in Newtons (float)

get_actual_joint_positions()

Returns the actual angular positions of all joints

The angular actual positions are expressed in radians and returned as a vector of length 6. Note that the output might differ from the output of get_target_joint_positions(), especially durring acceleration and heavy loads.

Return Value

The current actual joint angular position vector in rad : (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_actual_joint_speeds()

Returns the actual angular velocities of all joints

The angular actual velocities are expressed in radians pr. second and returned as a vector of length 6. Note that the output might differ from the output of get_target_joint_speeds(), especially durring acceleration and heavy loads.

Return Value

The current actual joint angular velocity vector in rad/s: (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_actual_tcp_pose()

Returns the current measured tool pose

Returns the 6d pose representing the tool position and orientation specified in the base frame. The calculation of this pose is based on the actual robot encoder readings.

Return Value

The current actual TCP vector : ((X, Y, Z, Rx, Ry, Rz))

get_actual_tcp_speed()

Returns the current measured TCP speed

The speed of the TCP retuned in a pose structure. The first three values are the cartesian speeds along x,y,z, and the last three define the current rotation axis, rx,ry,rz, and the length |rz,ry,rz| defines the angular velocity in radians/s.

Return Value

The current actual TCP velocity vector; ((X, Y, Z, Rx, Ry, Rz))

get_controller_temp()

Returns the temperature of the control box

The temperature of the robot control box in degrees Celcius.

Return Value

A temperature in degrees Celcius (float)

get_inverse_kin(*x*, *qnear*=`[-1.6, -1.7, -2.2, -0.8, 1.6, 0.0]`, *maxPositionError*=`0.0001`, *maxOrientationError*=`0.0001`)

Inverse kinematics

Inverse kinematic transformation (tool space -> joint space). Solution closest to current joint positions is returned, unless qnear defines one.

Parameters

`x`: tool pose (spatial vector)

`qnear`: joint positions to select solution. Optional.

`maxPositionError`: Define the max allowed position error. Optional.

`maxOrientationError`: Define the max allowed orientation error. Optional.

Return Value

joint positions

get_joint_temp(*j*)

Returns the temperature of joint j

The temperature of the joint house of joint j, counting from zero. j=0 is the base joint, and j=5 is the last joint before the tool flange.

Parameters

`j`: The joint number (int)

Return Value

A temperature in degrees Celcius (float)

get_joint_torques()

Returns the torques of all joints

The torque on the joints, corrected by the torque needed to move the robot itself (gravity, friction, etc.), returned as a vector of length 6.

Return Value

The joint torque vector in ; ((float))

get_target_joint_positions()

Returns the desired angular position of all joints

The angular target positions are expressed in radians and returned as a vector of length 6. Note that the output might differ from the output of get_actual_joint_positions(), especially durring acceleration and heavy loads.

Return Value

The current target joint angular position vector in rad: (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_target_joint_speeds()

Returns the desired angular velocities of all joints

The angular target velocities are expressed in radians pr. second and returned as a vector of length 6. Note that the output might differ from the output of get_actual_joint_speeds(), especially durring acceleration and heavy loads.

Return Value

The current target joint angular velocity vector in rad/s: (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_target_tcp_pose()

Returns the current target tool pose

Returns the 6d pose representing the tool position and orientation specified in the base frame. The calculation of this pose is based on the current target joint positions.

Return Value

The current target TCP vector; ((X, Y, Z, Rx, Ry, Rz))

get_target_tcp_speed()

Returns the current target TCP speed

The desired speed of the TCP retuned in a pose structure. The first three values are the cartesian speeds along x,y,z, and the last three define the current rotation axis, rx,ry,rz, and the length |rz,ry,rz| defines the angular velocity in radians/s.

Return Value

The TCP speed; (pose)

get_tcp_force()

Returns the wrench (Force/Torque vector) at the TCP

The external wrench is computed baed on the error between the joint torques required to stay on the trajectory and the expected joint torques. The function returns "p(Fx (N), Fy(N), Fz(N), TRx (Nm), TRy (Nm), TRz (Nm))". where Fx, Fy, and Fx are the forces in the axes of the robot base coordinate system measured in Newtons, and TRx, TRy, and TRz are the torques around these axes measyred in Newton times Meters.

Return Value

the wrench (pose)

popup(*s*, *title*=`'Popup'`, *warning*=`False`, *error*=`False`)

Display popup on GUI

Display message in popup window on GUI.

Parameters

`s:`	message string
`title:`	title string
`warning:`	warning message?
`error:`	error message?

powerdown()

Shutdown the robot, and power off the robot and controller.

set_gravity(*d*)

Set the direction of the acceleration experienced by the robot. When the robot mounting is fixed, this corresponds to an accleration of g away from the earth's centre.

```
>>> set_gravity([0, 9.82*sin(theta), 9.82*cos(theta)])
```

will set the acceleration for a robot that is rotated "theta" radians around the x-axis of the robot base coordinate system

Parameters

`d:` 3D vector, describing the direction of the gravity, relative to the base of the robot.

set_payload(*m*, *CoG*)

Set payload mass and center of gravity

Sets the mass and center of gravity (abbr. CoG) of the payload.

This function must be called, when the payload weight or weigh distribution changes significantly - I.e when the robot picks up or puts down a heavy workpiece.

The CoG argument is optional - If not provided, the Tool Center Point (TCP) will be used as the Center of Gravity (CoG). If the CoG argument is omitted, later calls to set_tcp(pose) will change CoG to the new TCP.

The CoG is specified as a Vector, (CoGx, CoGy, CoGz), displacement, from the toolmount.

Parameters

`m:` mass in kilograms

`CoG:` Center of Gravity: (CoGx, CoGy, CoGz) in meters. Optional.

set_tcp(*pose*)

Set the Tool Center Point

Sets the transformation from the output flange coordinate system to the TCP as a pose.

Parameters

`pose:` A pose describing the transformation.

sleep(*t*)

Sleep for an amount of time

Parameters

`t:` time (s)

sync()

Uses up the remaining "physical" time a thread has in the current frame.

textmsg(*s1, s2=''*)

Send text message to log

Send message with s1 and s2 concatenated to be shown on the GUI log-tab

Parameters

`s1`: message string, variables of other types (int, bool poses etc.) can also be sent

`s2`: message string, variables of other types (int, bool poses etc.) can also be sent

3.2 Variables

Name	Description
__package__	**Value:** `None`

4 Module urmath

4.1 Functions

acos(*f*)

Returns the arc cosine of f

Returns the principal value of the arc cosine of f, expressed in radians. A runtime error is raised if f lies outside the range (-1, 1).

Parameters

`f`: floating point value

Return Value

the arc cosine of f.

asin(*f*)

Returns the arc sine of f

Returns the principal value of the arc sine of f, expressed in radians. A runtime error is raised if f lies outside the range (-1, 1).

Parameters

`f`: floating point value

Return Value

the arc sine of f.

atan(*f*)

Returns the arc tangent of f

Returns the principal value of the arc tangent of f, expressed in radians.

Parameters

`f`: floating point value

Return Value

the arc tangent of f.

atan2(*x*, *y*)

Returns the arc tangent of x/y

Returns the principal value of the arc tangent of x/y, expressed in radians. To compute the value, the function uses the sign of both arguments to determine the quadrant.

Parameters

`x`: floating point value

`y`: floating point value

Return Value

the arc tangent of x/y.

binary_list_to_integer(*l*)

Returns the value represented by the content of list l

Returns the integer value represented by the bools contained in the list l when evaluated as a signed binary number.

Parameters

`l`: The list of bools to be converted to an integer. The bool at index 0 is evaluated as the least significant bit. False represents a zero and True represents a one. If the list is empty this function returns 0. If the list contains more than 32 bools, the function returns the signed integer value of the first 32 bools in the list.

Return Value

The integer value of the binary list content.

ceil(*f*)

Returns the smallest integer value that is not less than f

Rounds floating point number to the smallest integer no greater than f.

Parameters

`f`: floating point value

Return Value

rounded integer

cos(*f*)

Returns the cosine of f

Returns the cosine of an angle of f radians.

Parameters

`f`: floating point value

Return Value

the cosine of f.

d2r(*d*)

Returns degrees-to-radians of d

Returns the radian value of 'd' degrees. Actually: (d/180)*MATH_PI

Parameters

`d`: The angle in degrees

Return Value

The angle in radians

floor(*f*)

Returns largest integer not greater than f

Rounds floating point number to the largest integer no greater than f.

Parameters

`f`: floating point value

Return Value

rounded integer

get_list_length(*v*)

Returns the length of a list variable

The length of a list is the number of entries the list is composed of.

Parameters

`v`: A list variable

Return Value

An integer specifying the length of the given list

integer_to_binary_list(*x*)

Returns the binary representation of x

Returns a list of bools as the binary representation of the signed integer value x.

Parameters

`x`: The integer value to be converted to a binary list.

Return Value

A list of 32 bools, where False represents a zero and True represents a one. The bool at index 0 is the least significant bit.

interpolate_pose(*p_from*, *p_to*, *alpha*)

Linear interpolation of tool position and orientation.

When alhpa is 0, returns p_from. When alpha is 1, returns p_to. As alpha goes from 0 to 1, returns a pose going in a straigt line (and geodaetic orientation change) from p_from to p_to. If alpha is less than 0, returns a point before p_from on the line. If alpha is greater than 1, returns a pose after p_to on the line.

Parameters

`p_from`: tool pose (pose)

`p_to`: tool pose (pose)

`alpha`: Floating point number

Return Value

interpolated pose (pose)

length(*v*)

Returns the length of a list variable or a string

The length of a list or string is the number of entries or charterers it is composed of.

Parameters

`v`: A list or string variable

Return Value

An integer specifying the length of the given list or string

log(*b*, *f*)

Returns the logarithm of f to the base b

Returns the logarithm of f to the base b. If b or f are negative, or if b i 1 an runtime error is raised.

Parameters

`b`: floating point value

`f`: floating point value

Return Value

the logarithm of f to the base of b.

norm(*a*)

Returns the norm of the argument

The argument can be one of four diffrent types:

Pose: In this case the euclidian norm of the pose is returned.

Float: In this case fabs(a) is returned.

Int: In this case abs(a) is returned.

List: In this case the euclidian norm of the list is returned, the list elements must be numbers.

Parameters

`a`: Pose, float, int or List

Return Value

norm of a

point_dist(*p_from*, *p_to*)

Point distance

Parameters

`p_from`: tool pose (pose)

`p_to`: tool pose (pose)

Return Value

Distance between the two tool positions (without considering rotations)

pose_add(*p_1, p_2*)

Pose addition

Both arguments contain three position parameters (x, y, z) jointly called P, and three rotation parameters (R_x, R_y, R_z) jointly called R. This function calculates the result x_3 as the addition of the given poses as follows:

p_3.P = p_1.P + p_2.P

p_3.R = p_1.R * p_2.R

Parameters

`p_1`: tool pose 1(pose)

`p_2`: tool pose 2 (pose)

Return Value

Sum of position parts and product of rotation parts (pose)

pose_dist(*p_from, p_to*)

Pose distance

Parameters

`p_from`: tool pose (pose)

`p_to`: tool pose (pose)

Return Value

distance

pose_inv(*p_from*)

Get the invers of a pose

Parameters

`p_from`: tool pose (spatial vector)

Return Value

inverse tool pose transformation (spatial vector)

pose_sub(*p_to, p_from*)

Pose subtraction

Parameters

`p_to`: tool pose (spatial vector)

`p_from`: tool pose (spatial vector)

Return Value

tool pose transformation (spatial vector)

pose_trans(*p_from*, *p_from_to*)

Pose transformation

The first argument, p_from, is used to transform the second argument, p_from_to, and the result is then returned. This means that the result is the resulting pose, when starting at the coordinate system of p_from, and then in that coordinate system moving p_from_to.

This function can be seen in two different views. Either the function transforms, that is translates and rotates, p_from_to by the parameters of p_from. Or the function is used to get the resulting pose, when first making a move of p_from and then from there, a move of p_from_to.

If the poses were regarded as transformation matrices, it would look like:

T_world->to = T_world->from * T_from->to

T_x->to = T_x->from * T_from->to

Parameters

`p_from`: starting pose (spatial vector)

`p_from_to`: pose change relative to starting pose (spatial vector)

Return Value

resulting pose (spatial vector)

pow(*base*, *exponent*)

Returns base raised to the power of exponent

Returns the result of raising base to the power of exponent. If base is negative and exponent is not an integral value, or if base is zero and exponent is negative, a runtime error is raised.

Parameters

`base`: floating point value

`exponent`: floating point value

Return Value

base raised to the power of exponent

r2d(*r*)

Returns radians-to-degrees of r

Returns the degree value of 'r' radians.

Parameters

`r`: The angle in radians

Return Value

The angle in degrees

random()

Random Number

Return Value

peseudo-random number between 0 and 1 (float)

sin(*f*)

Returns the sine of f

Returns the sine of an angle of f radians.

Parameters

`f`: floating point value

Return Value

the sine of f.

sqrt(*f*)

Returns the square root of f

Returns the square root of f. If f is negative, an runtime error is raised.

Parameters

`f`: floating point value

Return Value

the square root of f.

tan(*f*)

Returns the tangent of f

Returns the tangent of an angle of f radians.

Parameters

`f`: floating point value

Return Value

the tangent of f.

4.2 Variables

Name	Description
__package__	**Value:** `None`

5 Module interfaces

5.1 Functions

get_analog_in(*n*)

Deprecated: Get analog input level

Parameters

`n`: The number (id) of the input, integer: (0:3)

Return Value

float, The signal level (0,1)

Deprecated: The `get_standard_analog_in` and `get_tool_analog_in` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For backwards compatibility n:2-3 go to the tool analog inputs.

get_analog_out(*n*)

Deprecated: Get analog output level

Parameters

`n`: The number (id) of the input, integer: (0:1)

Return Value

float, The signal level (0;1)

Deprecated: The `get_standard_analog_out` replaces this function. This function might be removed in the next major release.

get_configurable_digital_in(*n*)

Get configurable digital input signal level

See also `get_standard_digital_in` and `get_tool_digital_in`.

Parameters

`n`: The number (id) of the input, integer: (0:7)

Return Value

boolean, The signal level.

get_configurable_digital_out(*n*)

Get configurable digital output signal level

See also `get_standard_digital_out` and `get_tool_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:7)

Return Value

boolean, The signal level.

get_digital_in(*n*)

Deprecated: Get digital input signal level

Parameters

`n`: The number (id) of the input, integer: (0:9)

Return Value

boolean, The signal level.

Deprecated: The `get_standard_digital_in` and `get_tool_digital_in` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For backwards compatibility n:8-9 go to the tool digital inputs.

get_digital_out(*n*)

Deprecated: Get digital output signal level

Parameters

`n`: The number (id) of the output, integer: (0:9)

Return Value

boolean, The signal level.

Deprecated: The `get_standard_digital_out` and `get_tool_digital_out` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For backwards compatibility n:8-9 go to the tool digital outputs.

get_euromap_input(*port_number*)

Reads the current value of a specific Euromap67 input signal. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> var = get_euromap_input(3)
```

Parameters

`port_number`: An integer specifying one of the available Euromap67 input signals.

Return Value

A boolean, either True or False

get_euromap_output(*port_number*)

Reads the current value of a specific Euromap67 output signal. This means the value that is sent from the robot to the injection moulding machine. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> var = get_euromap_output(3)
```

Parameters

`port_number`: An integer specifying one of the available Euromap67 output signals.

Return Value

A boolean, either True or False

get_flag(*n*)

Flags behave like internal digital outputs. The keep information between program runs.

Parameters

`n`: The number (id) of the flag, intereger: (0:32)

Return Value

Boolean, The stored bit.

get_standard_analog_in(*n*)

Get standard analog input signal level

See also `get_tool_analog_in`.

Parameters

`n`: The number (id) of the input, integer: (0:1)

Return Value

boolean, The signal level.

get_standard_analog_out(*n*)

Get standard analog output level

Parameters

`n`: The number (id) of the input, integer: (0:1)

Return Value

float, The signal level (0;1)

get_standard_digital_in(*n*)

Get standard digital input signal level

See also `get_configurable_digital_in` and `get_tool_digital_in`.

Parameters

`n`: The number (id) of the input, integer: (0:7)

Return Value

boolean, The signal level.

get_standard_digital_out(*n*)

Get standard digital output signal level

See also `get_configurable_digital_out` and `get_tool_digital_out`.

Parameters

`n`: The number (id) of the input, integer: (0:7)

Return Value

boolean, The signal level.

get_tool_analog_in(*n*)

Get tool analog input level

See also `get_standard_analog_in`.

Parameters

`n`: The number (id) of the input, integer: (0:1)

Return Value

float, The signal level (0,1)

get_tool_digital_in(*n*)

Get tool digital input signal level

See also `get_configurable_digital_in` and `get_standard_digital_in`.

Parameters

`n`: The number (id) of the input, integer: (0:1)

Return Value

boolean, The signal level.

get_tool_digital_out(*n*)

Get tool digital output signal level

See also `get_standard_digital_out` and `get_configurable_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:1)

Return Value

boolean, The signal level.

modbus_add_signal(*IP*, *slave_number*, *signal_address*, *signal_type*, *signal_name*)

Adds a new modbus signal for the controller to supervise. Expects no response.

```
>>> modbus_add_signal("172.140.17.11", 255, 5, 1, "output1")
```

Parameters

`IP:`	A string specifying the IP address of the modbus unit to which the modbus signal is connected.
`slave_number:`	An integer normally not used and set to 255, but is a free choice between 0 and 255.
`signal_address:`	An integer specifying the address of the either the coil or the register that this new signal should reflect. Consult the configuration of the modbus unit for this information.
`signal_type:`	An integer specifying the type of signal to add. 0 = digital input, 1 = digital output, 2 = register input and 3 = register output.
`signal_name:`	A string uniquely identifying the signal. If a string is supplied which is equal to an already added signal, the new signal will replace the old one.

modbus_delete_signal(*signal_name*)

Deletes the signal identified by the supplied signal name.

```
>>> modbus_delete_signal("output1")
```

Parameters

`signal_name:`	A string equal to the name of the signal that should be deleted.

modbus_get_signal_status(*signal_name, is_secondary_program*)

Reads the current value of a specific signal.

```
>>> modbus_get_signal_status("output1",False)
```

Parameters

`signal_name:`	A string equal to the name of the signal for which the value should be gotten.
`is_secondary_program:`	A boolean for interal use only. Must be set to False.

Return Value

An integer or a boolean. For digital signals: True or False. For register signals: The register value expressed as an unsigned integer. For all signals: -1 for inactive signal, check then the signal name, addresses and connections.

modbus_send_custom_command(*IP, slave_number, function_code, data*)

Sends a command specified by the user to the modbus unit located on the specified IP address. Cannot be used to request data, since the response will not be received. The user is responsible for supplying data which is meaningful to the supplied function code. The builtin function takes care of constructing the modbus frame, so the user should not be concerned with the length of the command.

```
>>> modbus_send_custom_command("172.140.17.11",103,6,[17,32,2,88])
```

The above example sets the watchdog timeout on a Beckhoff BK9050 to 600 ms. That is done using the modbus function code 6 (preset single register) and then supplying the register address in the first two bytes of the data array ((17,32) = (0x1120)) and the desired register content in the last two bytes ((2,88) = (0x0258) = dec 600).

Parameters

`IP:`	A string specifying the IP address locating the modbus unit to which the custom command should be send.
`slave_number:`	An integer specifying the slave number to use for the custom command.
`function_code:`	An integer specifying the function code for the custom command.
`data:`	An array of integers in which each entry must be a valid byte (0-255) value.

modbus_set_output_register(*signal_name*, *register_value*, *is_secondary_program*)

Sets the output register signal identified by the given name to the given value.

```
>>> modbus_set_output_register("output1",300,False)
```

Parameters

`signal_name`:	A string identifying an output register signal that in advance has been added.
`register_value`:	An integer which must be a valid word (0-65535) value.
`is_secondary_program`:	A boolean for interal use only. Must be set to False.

modbus_set_output_signal(*signal_name*, *digital_value*, *is_secondary_program*)

Sets the output digital signal identified by the given name to the given value.

```
>>> modbus_set_output_signal("output2",True,False)
```

Parameters

`signal_name`:	A string identifying an output digital signal that in advance has been added.
`digital_value`:	A boolean to which value the signal will be set.
`is_secondary_program`:	A boolean for interal use only. Must be set to False.

modbus_set_runstate_dependent_choice(*signal_name, runstate_choice*)

Sets whether an output signal must preserve its state from a program, or it must be set either high or low when a program is not running.

```
>>> modbus_set_runstate_dependent_choice("output2",1)
```

Parameters

`signal_name:` A string identifying an output digital signal that in advance has been added.

`runstate_choice:` An integer: 0 = preserve program state, 1 = set low when a program is not running, 2 = set high when a program is not running.

modbus_set_signal_update_frequency(*signal_name, update_frequency*)

Sets the frequency with which the robot will send requests to the Modbus controller to either read or write the signal value.

```
>>> modbus_set_signal_update_frequency("output2",20)
```

Parameters

`signal_name:` A string identifying an output digital signal that in advance has been added.

`update_frequency:` An integer in the range 0-125 specifying the update frequency in Hz.

read_port_bit(*address*)

Reads one of the ports, which can also be accessed by Modbus clients

```
>>> boolval = read_port_bit(3)
```

Parameters

`address:` Address of the port (See portmap on Support site, page "UsingModbusServer")

Return Value

The value held by the port (True, False)

read_port_register(*address*)

Reads one of the ports, which can also be accessed by Modbus clients

```
>>> intval = read_port_register(3)
```

Parameters

`address`: Address of the port (See portmap on Support site, page "UsingModbusServer")

Return Value

The signed integer value held by the port (-32768 : 32767)

rpc_factory(*type, url*)

Creates a new Remote Procedure Call (RPC) handle. Please read the subsection ef{Remote Procedure Call (RPC)} for a more detailed description of RPCs.

```
>>> proxy = rpc_factory("xmlrpc", "http://127.0.0.1:8080/RPC2")
```

Parameters

`type`: The type of RPC backed to use. Currently only the "xmlrpc" protocol is available.

`url`: The URL to the RPC server. Currently two protocols are supported: pstream and http. The pstream URL looks like "<ip-address>:<port>", for instance "127.0.0.1:8080" to make a local connection on port 8080. A http URL generally looks like "http://<ip-address>:<port>/<path>", whereby the <path> depends on the setup of the http server. In the example given above a connection to a local Python webserver on port 8080 is made, which expects XMLRPC calls to come in on the path "RPC2". @note Giving the RPC instance a good name makes programs much more readable (i.e. "proxy" is not a very good name).

Return Value

A RPC handle with a connection to the specified server using the designated RPC backend. If the server is not available the function and program will fail. Any function that is made available on the server can be called using this instance. For example "bool isTargetAvailable(int number, ...)" would be "proxy.isTargetAvailable(var_1, ...)", whereby any number of arguments are supported (denoted by the ...).

set_analog_inputrange(*port*, *range*)

Deprecated: Set range of analog inputs

Port 0 and 1 is in the controller box, 2 and 3 is in the tool connector.

Parameters

`port`: analog input port number, 0,1 = controller, 2,3 = tool

`range`: *Controller* analog input range 0: *0-5V* (maps automatically onto range 2) and range 2: *0-10V*.

`range`: *Tool* analog input range 0: *0-5V* (maps automatically onto range 1), 1: *0-10V* and 2: *4-20mA*.

Deprecated: The `set_standard_analog_input_domain` and `set_tool_analog_input_domain` replace this function. Ports 2-3 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For *Controller* inputs ranges 1: *-5-5V* and 3: *-10-10V* are no longer supported and will show an exception in the GUI.

set_analog_out(*n*, *f*)

Deprecated: Set analog output level

Parameters

`n`: The number (id) of the input, integer: (0:1)

`f`: The signal level (0;1) (float)

Deprecated: The `set_standard_analog_out` replaces this function. This function might be removed in the next major release.

set_analog_outputdomain(*port*, *domain*)

Set domain of analog outputs

Parameters

`port`: analog output port number

`domain`: analog output domain: 0: *4-20mA*, 1: *0-10V*

set_configurable_digital_out(*n*, *b*)

Set configurable digital output signal level

See also `set_standard_digital_out` and `set_tool_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:7)

`b`: The signal level. (boolean)

set_digital_out(*n*, *b*)

Deprecated: Set digital output signal level

Parameters

`n`: The number (id) of the output, integer: (0:9)

`b`: The signal level. (boolean)

Deprecated: The `set_standard_digital_out` and `set_tool_digital_out` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

set_euromap_output(*port_number*, *signal_value*)

Sets the value of a specific Euromap67 output signal. This means the value that is sent from the robot to the injection moulding machine. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> set_euromap_output(3,True)
```

Parameters

`port_number`: An integer specifying one of the available Euromap67 output signals.

`signal_value`: A boolean, either True or False

set_euromap_runstate_dependent_choice(*port_number*, *runstate_choice*)

Sets whether an Euromap67 output signal must preserve its state from a program, or it must be set either high or low when a program is not running. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> set_euromap_runstate_dependent_choice(3,0)
```

Parameters

`port_number`: An integer specifying a Euromap67 output signal.

`runstate_choice`: An integer: 0 = preserve program state, 1 = set low when a program is not running, 2 = set high when a program is not running.

set_flag(*n*, *b*)

Flags behave like internal digital outputs. The keep information between program runs.

Parameters

`n`: The number (id) of the flag, integer: (0:32)

`b`: The stored bit. (boolean)

set_standard_analog_input_domain(*port*, *domain*)

Set domain of standard analog inputs in the controller box

For the tool inputs see `set_tool_analog_input_domain`.

Parameters

`port`: analog input port number: 0 or 1

`domain`: analog input domains: 0: *4-20mA*, 1: *0-10V*

set_standard_analog_out(*n*, *f*)

Set standard analog output level

Parameters

`n`: The number (id) of the input, integer: (0:1)

`f`: The relative signal level (0;1) (float)

set_standard_digital_out(*n*)

Set standard digital output signal level

See also `set_configurable_digital_out` and `set_tool_digital_out`.

Parameters

`n`: The number (id) of the input, integer: (0:7)

Return Value

boolean, The signal level.

set_tool_analog_input_domain(*port*, *domain*)

Set domain of analog inputs in the tool

For the controller box inputs see `set_standard_analog_input_domain`.

Parameters

`port`: analog input port number: 0 or 1

`domain`: analog input domains: 0: *4-20mA*, 1: *0-10V*

set_tool_digital_out(*n*, *b*)

Set tool digital output signal level

See also `set_configurable_digital_out` and `set_standard_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:1)

`b`: The signal level. (boolean)

set_tool_voltage(*voltage*)

Sets the voltage level for the power supply that delivers power to the connector plug in the tool flange of the robot. The votage can be 0, 12 or 24 volts.

Parameters

`voltage`: The voltage (as an integer) at the tool connector, integer: 0, 12 or 24.

socket_close(*socket_name*=`'socket_0'`)

Closes ethernet communication

Closes down the socket connection to the server.

```
>>> socket_comm_close()
```

Parameters

`socket_name`: Name of socket (string)

socket_get_var(*name*, *socket_name*=`'socket_0'`)

Reads an integer from the server

Sends the message "get <name> " through the socket, expects the response "<name> <int> " within 2 seconds. Returns 0 after timeout

```
>>> x_pos = socket_get_var("POS_X")
```

Parameters

`name`: Variable name (string)

`socket_name`: Name of socket (string)

Return Value

an integer from the server (int), 0 is the timeout value

socket_open(*address*, *port*, *socket_name*=`'socket_0'`)

Open ethernet communication

Attempts to open a socket connection, times out after 2 seconds.

Parameters

`address`: Server address (string)

`port`: Port number (int)

`socket_name`: Name of socket (string)

Return Value

False if failed, True if connection succesfully established

socket_read_ascii_float(*number*, *socket_name*=`'socket_0'`)

Reads a number of ascii formatted floats from the TCP/IP connected. A maximum of 30 values can be read in one command.

```
>>> list_of_four_floats = socket_read_ascii_float(4)
```

The format of the numbers should be in parantheses, and seperated by ",". An example list of four numbers could look like "(1.414 , 3.14159, 1.616, 0.0)".

The returned list contains the total numbers read, and then each number in succession. For example a read_ascii_float on the example above would return (4, 1.414, 3.14159, 1.616, 0.0).

A failed read or timeout after 2 seconds will return the list with 0 as first element and then "Not a number (nan)" in the following elements (ex. (0, nan., nan, nan) for a read of three numbers).

Parameters

`number`: The number of variables to read (int)

`socket_name`: Name of socket (string)

Return Value

A list of numbers read (list of floats, length=number+1)

socket_read_binary_integer(*number*, *socket_name*=`'socket_0'`)

Reads a number of 32 bit integers from the TCP/IP connected. Bytes are in network byte order. A maximum of 30 values can be read in one command.

```
>>> list_of_three_ints = socket_read_binary_integer(3)
```

Returns (for example) (3,100,2000,30000), if there is a timeout (2 seconds) or the reply is invalid, (0,-1,-1,-1) is returned, indicating that 0 integers have been read

Parameters

`number`: The number of variables to read (int)

`socket_name`: Name of socket (string)

Return Value

A list of numbers read (list of ints, length=number+1)

socket_read_byte_list(*number*, *socket_name*=`'socket_0'`)

Reads a number of bytes from the TCP/IP connected. Bytes are in network byte order. A maximum of 30 values can be read in one command.

```
>>> list_of_three_ints = socket_read_byte_list(3)
```

Returns (for example) (3,100,200,44), if there is a timeout (2 seconds) or the reply is invalid, (0,-1,-1,-1) is returned, indicating that 0 bytes have been read

Parameters

`number`: The number of variables to read (int)

`socket_name`: Name of socket (string)

Return Value

A list of numbers read (list of ints, length=number+1)

socket_read_string(*socket_name*='socket_0', *prefix*='', *suffix*='')

Reads a string from the TCP/IP connected. Bytes are in network byte order.

```
>>> string_from_server = socket_read_string()
```

Returns (for example) "reply string from the server", if there is a timeout (2 seconds) or the reply is invalid, an empty string is returned (""). You can test if the string is empty with an if-statement.

```
>>> if(string_from_server) :
>>>   popup("the string is not empty")
>>> end
```

The optional parameters, "prefix" and "suffix", can be used to express what is extracted from the socket. The "prefix" specifies the start of the substring (message) extracted from the socket. The data upto the end of the "prefix" will be ignored and removed from the socket. The "suffix" specifies the end of the substring (message) extracted from the socket. Any remaning data on the socket, after the "suffix", will be preserved. E.g. if the socket server sends a string "noise>hello<", the controller can receive the "hello" by calling this script function with the prefix=">" and suffix="<".

```
>>> hello = socket_read_string(prefix=">", suffix="<")
```

By using the "prefix" and "suffix" it is also possible send multiple string to the controller at ones, because the suffix defines then the message ends. E.g. sending ">hello<>world<"

```
>>> hello = socket_read_string(prefix=">", suffix="<")
>>> world = socket_read_string(prefix=">", suffix="<")
```

Parameters

`socket_name`: Name of socket (string)

`prefix`: Defines a prefix (string)

`suffix`: Defines a suffix (string)

Return Value

A string variable

socket_send_byte(*value*, *socket_name*=`'socket_0'`)

Sends a byte to the server

Sends the byte <value> through the socket. Expects no response. Can be used to send special ASCII characters; 10 is newline, 2 is start of text, 3 is end of text.

Parameters

`value`: The number to send (byte)

`socket_name`: Name of socket (string)

socket_send_int(*value*, *socket_name*=`'socket_0'`)

Sends an int (int32_t) to the server

Sends the int <value> through the socket. Send in network byte order. Expects no response.

Parameters

`value`: The number to send (int)

`socket_name`: Name of socket (string)

socket_send_line(*str*, *socket_name*=`'socket_0'`)

Sends a string with a newline character to the server - useful for communicatin with the UR dashboard server

Sends the string <str> through the socket in ASCII coding. Expects no response.

Parameters

`str`: The string to send (ascii)

`socket_name`: Name of socket (string)

socket_send_string(*str*, *socket_name*=`'socket_0'`)

Sends a string to the server

Sends the string <str> through the socket in ASCII coding. Expects no response.

Parameters

`str`: The string to send (ascii)

`socket_name`: Name of socket (string)

socket_set_var(*name, value, socket_name=*'socket_0')

Sends an integer to the server

Sends the message "set <name> <value> " through the socket. Expects no response.

```
>>> socket_set_var("POS_Y",2200)
```

Parameters

`name`: Variable name (string)

`value`: The number to send (int)

`socket_name`: Name of socket (string)

write_port_bit(*address, value*)

Writes one of the ports, which can also be accessed by Modbus clients

```
>>> write_port_bit(3,True)
```

Parameters

`address`: Address of the port (See portmap on Support site, page "UsingModbusServer")

`value`: Value to be set in the register (True, False)

write_port_register(*address, value*)

Writes one of the ports, which can also be accessed by Modbus clients

```
>>> write_port_register(3,100)
```

Parameters

`address`: Address of the port (See portmap on Support site, page "UsingModbusServer")

`value`: Value to be set in the port (0 : 65536) or (-32768 : 32767)

5.2 Variables

Name	Description
__package__	**Value:** `None`