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 tente d'exemplifier et a une approche pratique sur l'utilisation des outils mentionnés.

1 Introduction

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 langages de programmation pour minimiser le temps d'adaptation. Une solution permettant l'utilisation du code Matlab, python, c++, etc. est alors mise sur pied.

2 Mise en contexte

Les solutions présentées dans ce rapport répondent toutes au besoin d'utiliser le code directement sur un montage robotique. Cependant, elles présentent toutes différents avantages et inconvénients. Comme processus de validation, les solutions ont été testées en répondant à la même problématique : l'utilisation d'un bras robotique sans l'utilisation d'un noeud de calcul.

3 Définition des objectifs

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

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

Les solutions présentées dans ce rapport répondent toutes à ces critères.

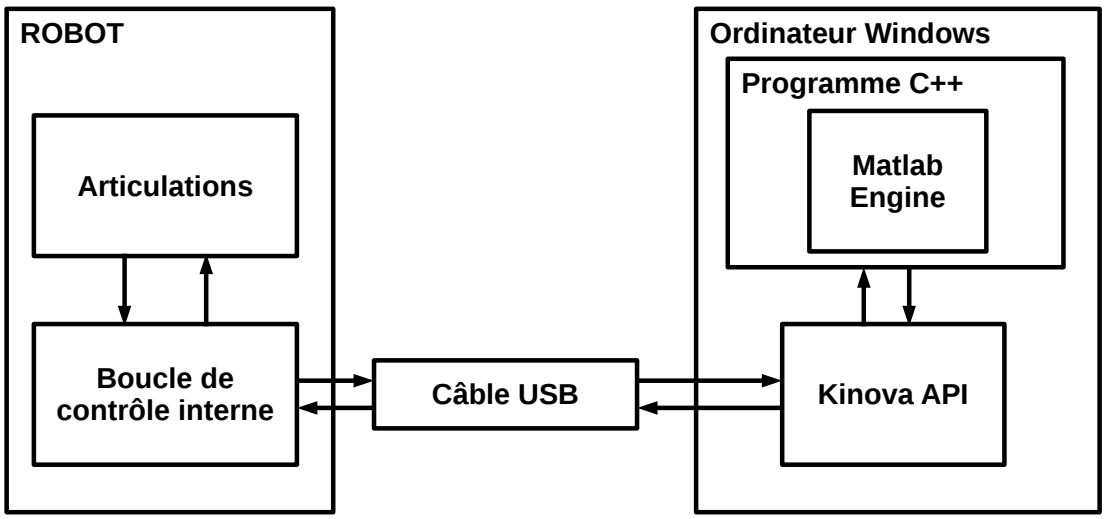

FIGURE 1 – Structure de l'environnement de développement.

4 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 représentée 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, telles que l'exécution de fonctions Matlab. Par la suite, il suffit d'utiliser directement l'API de kinova pour envoyer des commandes.

4.1 Exemple d'implémentation

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

4.1.1 Visual Studio

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

4.1.2 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 cliquer 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. Un ficher placé dans ce répertoire fera partie du projet et pourra être référé dans les scripts.

4.1.3 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.

4.1.4 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 sont disponibles sur le site web fourni. Un exemple de code utilisant la librairie, ainsi que tous les modules mentionnés dans cette section, est fourni en annexe A.

4.1.5 Lien avec Matlab

La description de cette partie se fera en deux 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'envoi de commandes au bras Jaco est expliquée.

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 la 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 inclu avec les version récentes de Matlab. Il est accessible en entrant la commande :

— $edit([matlabroot\backslash extern\backslash examples\backslash eng\_mat\backslash engdemo.cpp'])$;

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 subséquentes.

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.

4.2 Conclusion

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 présentée. 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.

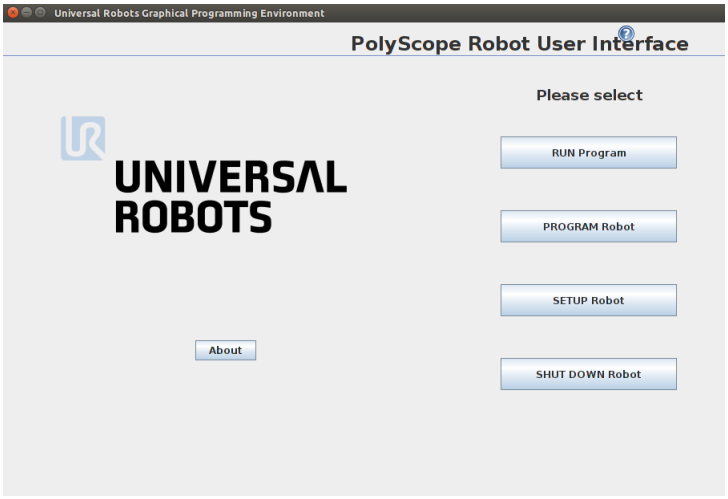

FIGURE 2 – Interface initiale d'URsim

5 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.

5.1 URsim et son utilisation

URsim permet l'interface avec un robot virtuel ou un robot réel connecté sur le réseau. Cette caractéristique est importante puisqu'il serait relativement facile de faire un simulateur directement dans matlab, cependant ce dernier ne pourrait pas directement contrôler le bras UR5. En utilisant URsim, le robot simulé agit directement comme un vrai robot, simulant les limitation de forces, de courant et les arrêts d'urgences. 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. Suite au lancement de la machine virtuelle et d'URsim, l'interface initiale d'URsim vous est présentée (Figure 2).

— 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

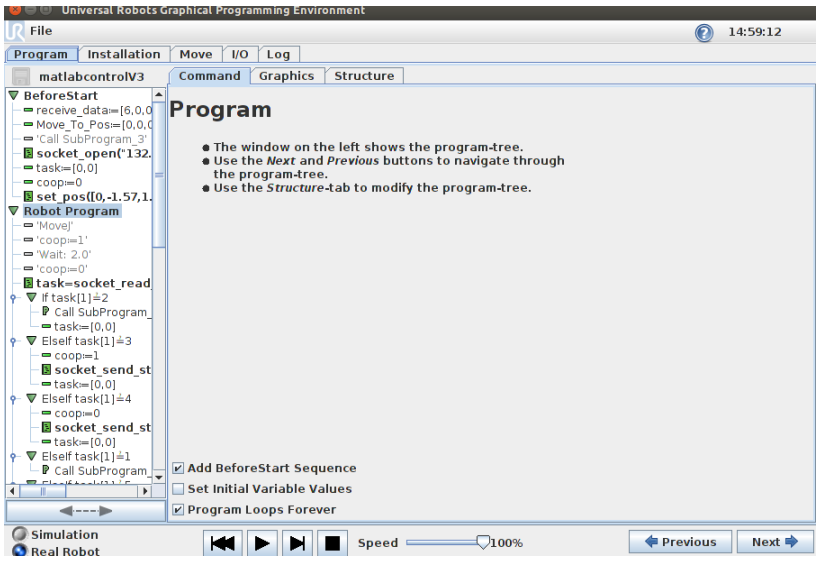

FIGURE 3 – Interface de programmation d'URsim

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.

5.2 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. En premier lieu, les manipulations requises dans le logiciel URsim lancé dans la machiner virtuelle seront décrites. Ensuite, les étapes à suivres dans l'environnement de développement Matlab sous windows seront expliquées.

5.2.1 Manipulations dans URsim

Un URscript a été rédigé afin de permettre à URSIM de répondre à certaines commandes envoyées sous format texte via Matlab. Les fichiers URscripts en question sont disponibles 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 dossier *programs*. Suivant le dépot du dossier *programs* dans le répertoire d'URsim, il est possible d'ouvrir le programme portant le nom *matlabcontrolV3.urp* après avoir sélectionné *Load Program* tel que vu dans la Figure 2.

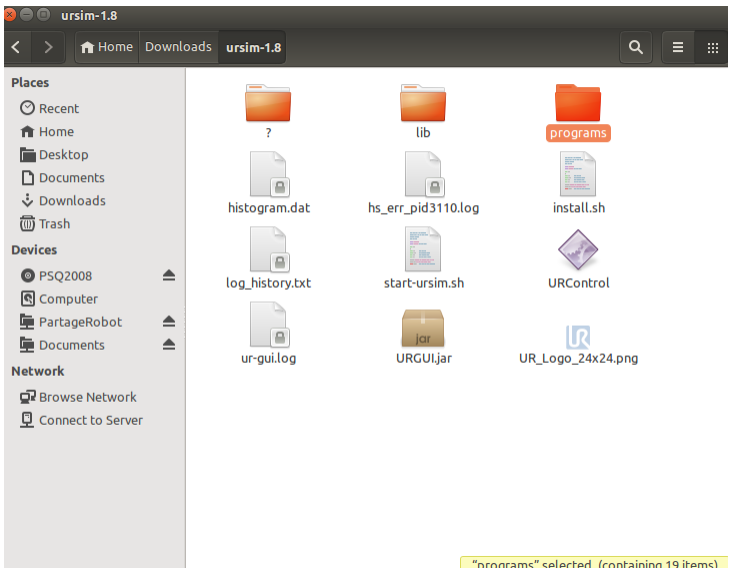

FIGURE 4 – Emplacement du dossier *programs*

```
Ethernet adapter VMware Network Adapter VMnet8:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::88:6b3c:9bf2:4f61%18
   IPv4 Address. . . . . . . . . . . : 192.168.93.1
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :
```

FIGURE 5 – Exemple de l'adresse ip du contrôleur de VMware qu'il faut prendre en note (IPv4 address)

Ensuite, il faut executer les étapes suivante :

— Modifier l'adresse IP dans la fonction *socket_open* du URscript pour qu'elle concorde avec l'adresse IP du controleur réseau de VMware. Cette infromation est disponible en ouvrant un invité de commande dans la session Windows et de tapper ipconfig. La Figure 5 représente la sortie de l'invité de commande qu'il faut remarquer lors de l'envoie de la commande ipconfig.

— Sélectionner l'option *Simulation* dans le bas de la fenêtre d'URsim.

— Appuyer sur le bouton *Play* présent de le bas, au centre, de la fenêtre d'URsim.

Ces étapes lancent l'URscript sur un robot virtuel et permet l'écoute et la réponse à des commandes envoyées par Matlab.

5.2.2 Interface Matlab-URsim

Une fois l'environnement virtuel d'URsim mis en place, l'environement matlab requis pour communiquer avec URsim doit être implémenté. Un environnement de travail matlab situé dans le répertoire github mentionné plus dans la dernière section contient plusieurs fichiers nécessaires à la communiquation avec URsim.

Un survol rapide du URscript *matlabcontrolV3.urp* permet d'observer qu'URsim attend l'arrivée de différents types de packets et détermine la commande à accomplir avec la valeur de la première donnée comprise dans une liste de longueur prédéterminée. Les fichiers Matlab permettant de faire

le lien avec URsim font seulement traduire les arguments dans la forme qui rend possible la lecture par VRep.

Voici la liste des fonctions disponible dans le répertoire github fourni :

— Robot_Pose_j = readrobotpose_j(Socket_conn). Lecture des position articulaires du robot.
— Robot_Pose = readrobotpose(Socket_conn). Lecture de la position à l'effecteur du robot.
— Robot_Pose_new = moverobot(Socket_conn,Goal_Pose,Orientation). Envoie de commande en position articulaires.
— Robot_speed = readrobotspeed(Socket_conn). Lecture des vitesses cartésiennes du robot.
— Robot_speed_j = readrobotspeed_j(Socket_conn). Lecture des vitesses articulaires du robot.
— Robot_speed_j_new = speedrobot(Socket_conn,theta_dot). Envoie de commande en vitesse articulaires.
— inv_kin = getinversekin(Socket_conn,next_pose). Calcul de la cinématique inverse du UR5.

Ces fonctions permettent l'appel des fonctions d'URsim listées dans l'annexe B. Une fois les packets lus par URsim, le URscript détermine quelle fonction appeler. Suivant la méthodologie utilisée, il est facile d'ajouter des fonctions au URscript et d'adapter des fonctions Matlab à la nouvelle commande à envoyer.

5.3 Logiciel de visualistion VRep

Bien que le logiciel URsim permet de représenter le robot dans un environnement 3D de base, il ne permet pas la représentation d'objets composant l'environnement de travail du robot. Pour se faire, le logiciel VRep est utilisé. Ce dernier permet d'afficher une multitude d'objets localisés dans une librairie déjà prédéfinie. Cette section couvre brièvement les étapes requise pour l'instalation et l'utilisatiuon de VRep avec matlab. Il est à noter que VRep peut également servir de simulateur pour le UR5, cependant, il n'est pas aussi fidèle qu'URsim pour représenter les limitation en couple, courrant ou les arrêts d'urgence.

5.3.1 Instalation de VRep

L'instalation de VRep peut se faire rapidement en téléchargeant la version d'éducation sur le site web officiel suivant.

5.3.2 Interface Matlab-VRep

VRep est doté d'une API très versatile pouvant être utilisée dans plusieurs langages de programation tels que C/C++, Python, Matlab, Java, Octave et Lua. Pour avoir accès aux fonction de l'API dans un projet Matlab, il faut aller chercher les fichiers suivants :

— remApi.m
— remoteApiProto.m
— remoteApi.dll

Ces fichiers sont situés dans les dossiers d'instalation de VRep. Sur le système d'exploitation Windows, cet endroit est normalement :
C :\Program Files (x86)\V-REP3\V-REP_PRO_EDU\programming\remoteApiBindings\matlab\matlab pour le fichier ".m" et :
C :\Program Files (x86)\V-REP3\V-REP_PRO_EDU\programming\remoteApiBindings\lib\lib\64Bit pour le fichier ".dll".

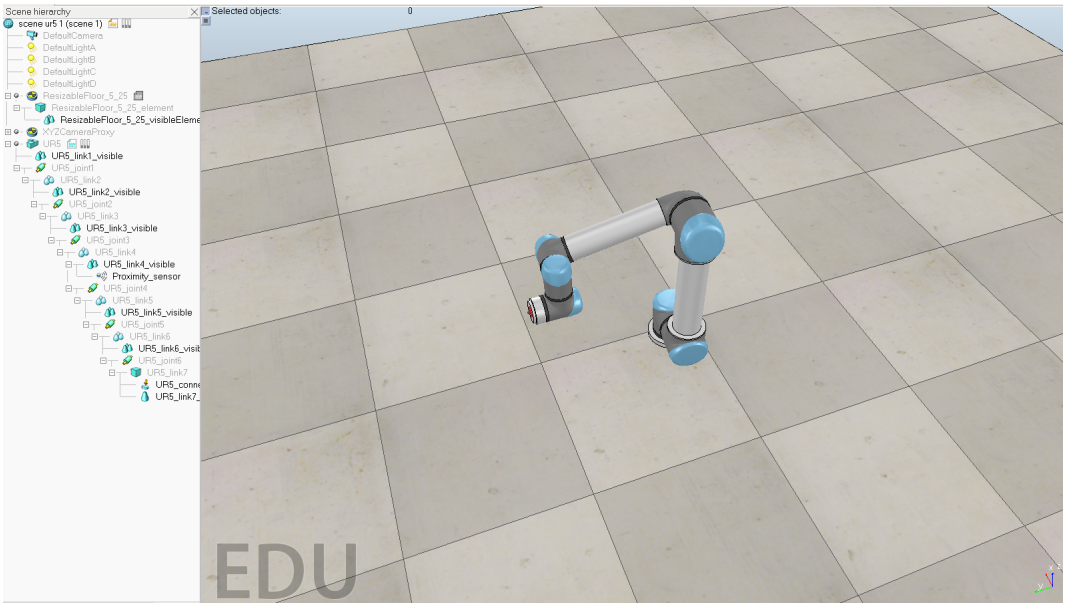

FIGURE 6 – Interface du simulateur VRep avec la scène chargée.

Suite à l'incorporation de ces fichiers dans l'environnement de travail, il est possible d'initialiser l'objet interagissant avec l'API dans matlab avec cette commande :

vrep=remApi('remoteApi') ;

Ensuite, il est possible d'envoyer des commandes à une instance de VRep. Dans l'environnement du simulateur, l'interaction du script matlab avec VRep se résume à l'envoie des positions articulaires lues dans URsim.

5.4 Exemple d'utilisation

Les étapes suivantes réfèrent aux scripts matlab situés dans le dossier *URsim-vrep-matlab* du dépot github mentionné plus haut. L'utilisation de la solution présentée dans cette section peut être résumée par les étapes suivantes :

— Connection d'un joystick dans l'ordinateur (requis pour l'exemple).
— Ouverture de la machine virtuelle et démarage d'URsim
— Chargement du URscript dans URsim.
— Ouverture de VRep sur la session windows.
— Chargement de la scène *scene_ur5_1.ttt* dans VRep.
— Démarage du script matlab nommé *Exemple_controle_ur5_vrep.m*.
— Appuyer sur le bouton "play" de URsim

Suite à l'exécution de ces étapes, le robot est contrôlable avec le joystick et devrait être visualisé dans VRep. La Figure 6 représente l'interface VRep lorsque la scène est chargée. Pour plus de détails quant à l'utilisation des fonctions dans matlab, il est possible de lire le code exemple qui a été commenté pour faciliter la compréhension. De plus, l'annexe C est constituée d'un document listant la magoritée des commandes disponibles.

5.5 Conclusion

La solution de contrôle du bras UR5 avec matlab à été présentée. L'instalation et l'utilisation de URsim dans une machine virtuelle à été abordée. Cependant la nécessité d'utiliser une machine virtuelle ainsi qu'un logiciel de visualisation tel que VRep est quelque peu compliquée.

6 Contrôle du bras Jaco dans le simulateur Gazebo par l'entremise de ROS

Cette section traite d'un environnement de travail hautement versatile permettant la représentation de plusieur architectures de robots. Pour les besoins du rapport, seulement un exemple avec le bras Jaco sera discuté. De plus, la complexité de ROS et du simulateur Gazebo oblige de restreindre la présente section à la description sommaire des étapes nécessaires à la mise en marche de l'environnement et à l'utlisation de l'exemple fourni. Les descriptions fournies ne cherchent pas à être fondamentalement exactes, mais plutôt à expliquer de manière intuitive et succincte le fonctionnement de l'infrastructure à des lecteurs non-initiés. Il est important de noter que cet environnement est disponible seulement sur les plateformes Linux.

6.1 Description de l'architecture générale de ROS

ROS, *robot operating system*, peut être décrit comme un ensemble de programmes (*noeuds*) pouvant fonctionner de manière indépendante, entre lesquels des cannaux de communications standardisés sont établis. Ces programmes communiquent entre eux en publiant l'information dans une liste des différents sujets (*topics*) publiés par les noeuds présentement en fonctionnement. Pour obtenir l'information publiée, un autre noeud peut s'abonner au sujet et ainsi lire l'information.

6.2 Description du simulateur Gazebo

Gazebo est un simulateur 3D pouvant fonctionner sans nécessiter l'instalation de ROS. Cependant, il existe une version intégrée de Gazebo dans ROS Kinetic, la version de ROS utilisée pour Ubuntu 16.04. Gazebo peut être vu comme un noeud ROS qui publie des informations par rapport aux différents composants de la simulation. En utilisant le standard de fichier URDF, c'est dans ce programme que le robot est défini et que les contrôleurs des articulations sont créés. Suivant la création du robot, Gazebo publie ainsi les informations de l'état du robot et s'abonne à des topics permettant la reception de commandes de moteurs. La Figure 7 représente l'interface du simulateur Gazebo lorsque le robot Jaco est configuré.

La configuration du robot Jaco à été adaptée d'une suite de programmes (package) open-souce publiée par [2].

6.3 Installation de ROS et de Gazebo

Comme mentionné plus haut ROS est disponible seulement sur les systèmes d'exploitation Linux. La séquence d'instructions suivante assume que le système d'exploitation Ubuntu 16.04 est fraichement installé.

- Aller sur le site web suivant.
- Dans un invité de commande (terminal), entrer, dans l'ordre, les commandes aux étapes 1.2 et 1.3 du tutoriel.
- Dans un terminal, entrer la commande : *sudo apt-get install ros-kinetic-desktop-full*. Ceci installe ROS et Gazebo.
- Dans un terminal, entrer, dans l'ordre, les commandes à l'étape 1.5 du tutoriel.
- Dans un terminal, entrer la commande : echo "source /opt/ros/kinetic/setup.bash" >> ∼/.bashrc
- Entrer ensuite la commande : *source ∼/.bashrc*

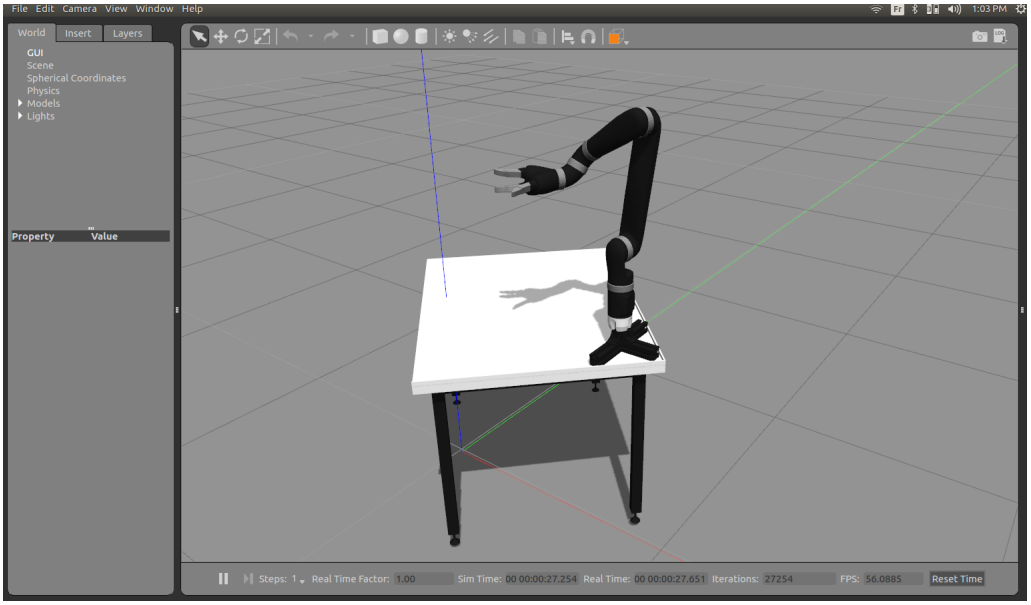

FIGURE 7 – Interface du simulateur Gazebo avec le robot Jaco configuré

— Dans un terminal, entrer la commande : *sudo apt-get install ros-kinetic-catkin*. Ce programme permet la construction d'un environnement de travail avec ROS.
— Aller sur le site web suivant.
— Entrer les commandes mentionnée dans le tutoriel sur cette page.
— Dans un terminal, entrer la commande : echo "source ∼/catkin_ws/devel/setup.bash" >> ∼/.bashrc
— Entrer ensuite la commande : *source ∼/.bashrc*

Après l'exécution de ces étapes, un environnement de travail pour ROS et Gazebo a été créé dans le fichier *catkin_ws* situé dans le répertoire *home* de la plateforme Ubuntu. Les fichiers sur lesquels le projet repose seront placés dans le sous-fichier *src* de l'environnement de travail.

6.4 Installation de la suite de programmes Jaco

Après avoir installé ROS et Gazebo, il est maintenant possible de venir grèfer les modules permettant le contrôle d'un bras jaco en simulation. Pour se faire, il est nécessaire de télécharger les dépots git-hub suivants :

— `https://github.com/wonwon0/jaco-arm-pkgs`
— `https://github.com/wonwon0/jaco_gazebo_tools`
— `https://github.com/wonwon0/kinova-ros`
— `https://github.com/wonwon0/joint-control-pkgs`
— `https://github.com/wonwon0/convenience-pkgs`

Après les avoirs positionnés dans le fichier *src* de l'environnement de travail, ouvrir un invité de commande dans le répertoire de base de l'environnement de travail (*catkin_ws*) et entrer successivement les commandes suivantes :

— catkin_make
— catkin_make install
— source ∼/.bashrc

Suivant l'éxécution de ces commandes, il est maintenant possible de démarrer l'environnement de simulation en entrant la commande :

— roslaunch roslaunch jaco_on_table jaco_on_table_gazebo_controlled.launch

dans n'importe quel invité de commande ouvert après l'installation. Suite à l'exécution de la commande, le simulateur Gazebo s'ouvre avec un robot joco positionné sur une table. Les *topics* pour communiquer avec le robot sont publiés sur un noeud ROS rendant le contrôle des moteurs possible. L'envirionnement de simulation devrait être similaire à ce qui est montré dans la Figure 7.

6.5 Contrôle du robot jaco en simulation et d'un réel bras Jaco

Le répertoire git-hub se nommant *jaco_gazebo_tools*, présent dans le dossier *src* de l'environnement de travail, contient différents scripts python permettant de contrôler le robot en simulation et dans une application réelle. Ces scripts requierent l'installation du module python *pygame* permettant l'utilisation d'un joystick. Pour ce faire, il faut seulement entrer la commande :

— pip install –user pyagme

Après l'ouverture d'un invité de commande dans le répertoire *jaco_gazebo_tools*, la simulation peut être lancée avec la commande suivante :

— sh launch_jaco_simulation.sh

Cette commande lance une simulation du bras jaco dans Gazebo. Il est important de savoir que ce script ne fonctionne que si un joystick est connecté à l'ordinateur. Pour controler un bras jaco réel, la commande est la suivante :

— sh launch_jaco_real_arm.sh

Cette commande lance aussi un simulateur Gazebo, mais ce dernier ne fait que répliquer la position du bras connecté sur l'ordinateur. Il est possible que le modèle du bras Jaco utilisé ne soit pas exactement le même que celui pour lequel les scripts pythons ont été développés. Des modifications dans le code seront nécessaires dans certains scripts. Les fichiers à modifier sont entre autre :

— joint_states_listener_real_jaco.py

— jaco_joints_client.py

Ces scripts ont été créés pour le modèle de bras Jaco*j2n6s300*. Pour plus d'information, il est possible de visiter le dépot github de Kinova :

— `https://github.com/Kinovarobotics/kinova-ros`

6.6 Conclusion

L'installation du simulateur Gazebo et de l'environnement de développement ROS a été couvert. Bien que l'utilisation en détail de cette suite de logiciel n'a pas été couverte, une approche pratique permet d'utiliser un simulateur d'un bras Jaco et de conrôler un véritable bras Jaco avec l'aide d'un joystick. Cette solution est la plus flexible et modulaire parmis les trois approches présentées dans ce rapport. Cependant elle a été, de loin, la plus longue à mettre sur pied.

Références

[1] First Reference...

[2] https ://github.com/JenniferBuehler

A

Exemple de code pour contrôle du bras Jaco sur Windows

robot_IO_ctrl_phleb38.cpp

```
1
2
3   #include <Engine.h>
4   #include <JacoUdp.h>
5
6   #include <Windows.h>
7   #include <conio.h>
8   #include <iostream>
9   #include <fstream>
10  #include <string>
11  #include <sstream>
12  #include <ctime>
13  #include "windows.h"
14  #include <chrono>
15  #include <math.h>
16
17  #include <time.h>
18  #include <sys\timeb.h>
19  #include <winbase.h>
20
21  #include "WindowsExample_AngularControl\Lib_Examples\CommunicationLayerWindows.h"
22  #include "WindowsExample_AngularControl\Lib_Examples\CommandLayer.h"
23  #include "WindowsExample_AngularControl\Lib_Examples\KinovaTypes.h"
24  #include <SDL.h>
25  #undef main
26  #pragma comment(lib,"libmat.lib")
27  #pragma comment(lib,"libeng.lib")
28  #pragma comment(lib,"libmex.lib")
29  #pragma comment(lib,"libmx.lib")
30  using namespace std;
31  using namespace System;
32  using namespace System::Threading;
33  using namespace System::Collections::Generic;
34
35  //A handle to the API.
36  HINSTANCE commandLayer_handle;
37
38  //Function pointers to the functions we need
39  int(*MyInitAPI)();
40  int(*MyCloseAPI)();
41  int(*MySendBasicTrajectory)(TrajectoryPoint command);
42  int(*MyGetDevices)(KinovaDevice devices[MAX_KINOVA_DEVICE], int &result);
43  int(*MySetActiveDevice)(KinovaDevice device);
44  int(*MyMoveHome)();
45  int(*MyInitFingers)();
46  int(*MyGetAngularCommand)(AngularPosition &);
47  int(*MyEraseAllTrajectories)();
48  int(*MyGetAngularPosition)(AngularPosition &);
49  int quitter = 1;
50  Engine *m_pEngine = engOpen("null");
51
52  double mc_get_accurate_time()
53  {
54      return((double)GetTickCount() / 1000.0);
55  }
56
57  double getTime()
58  {
59      struct timeb lTp;
60
61      ftime(&lTp);
62
63      double lTime = (double)lTp.time + 0.001f*lTp.millitm;
64
65      return lTime;
66  }
67
68  void afficherGraphique()
69  {
70      mxArray* dstate = mxCreateDoubleMatrix(1, 1, mxREAL);
71      double* pstate = mxGetPr(dstate);
72      *pstate = 0;
73      engPutVariable(m_pEngine, "state", dstate);
74      engEvalString(m_pEngine, "graphique");
75      *pstate = 1;
76      engPutVariable(m_pEngine, "state", dstate);
77      while (!quitter)
78      {
79          engEvalString(m_pEngine, "graphique");
80      }
81  }
82
83  int main(array<System::String ^> ^args)
84  {
85      // Ouverture de l'engine matlab
86      Engine *m_pEngine;
87      m_pEngine = engOpen("null");
88
89
90      //Chargement de l'API de kinova
91      commandLayer_handle = LoadLibrary(L"CommandLayerWindows.dll");
92
93      int programResult = 0;
94      Sint16 x_move = 0, y_move = 0, z_move = 0, z_move_1 = 0, z_move_2 = 0;
95      JacoUdp jacoInst;
96      int i = 0;
97      int j = 0;
98      int k = 0;
99      int flag = 0;
100     double time1, time2, ang_1, ang_2, ang_3;
101     int iteration = 10000;
102     void *ans = NULL;
```

```
103
104     int dummy = 0;
105     AngularPosition dataPosition;
106
107     // Definition des variables communiquant avec matlab (memes noms que dans matlab)
108     mxArray* dmove = mxCreateDoubleMatrix(1, 3, mxREAL);
109     mxArray* dtheta = mxCreateDoubleMatrix(1, 3, mxREAL);
110     mxArray* dtheta_next = mxCreateDoubleMatrix(1, 3, mxREAL);
111     mxArray* dtheta_delt = mxCreateDoubleMatrix(1, 3, mxREAL);
112     mxArray* dstate = mxCreateDoubleMatrix(1, 1, mxREAL);
113
114
115     // initialisation des valeurs des variables
116     double *pmove = new double[3];
117     pmove[0] = 0;
118     pmove[1] = 0;
119     pmove[2] = 0;
120     double *ptheta = new double[3];
121     ptheta[0] = 0;
122     ptheta[1] = 0;
123     ptheta[2] = 0;
124     double *ptheta_next = new double[3];
125     ptheta_next[0] = 0;
126     ptheta_next[1] = 0;
127     ptheta_next[2] = 0;
128
129     double *ptheta_delt = new double[3];
130     ptheta_delt[0] = 0;
131     ptheta_delt[1] = 0;
132     ptheta_delt[2] = 0;
133     double *pstate = mxGetPr(dstate);
134
135     memcpy(mxGetPr(dmove), pmove, 3 * sizeof(double));
136     memcpy(mxGetPr(dtheta), ptheta, 3 * sizeof(double));
137     memcpy(mxGetPr(dtheta_next), ptheta_next, 3 * sizeof(double));
138     memcpy(mxGetPr(dtheta_delt), ptheta_delt, 3 * sizeof(double));
139
140     *pstate = (double)0; // propre au script matlab
141
142     // envoi des valeur initiales dans le script maple
143
144     engPutVariable(m_pEngine, "move", dmove);
145     engPutVariable(m_pEngine, "theta", dtheta);
146
147     //execution de commandes matlab. Il faut ajouter le path dans lequel le script matlab se situe.
148     engEvalString(m_pEngine, "addpath('U:\\matlab\\rrr version phil')");
149     engEvalString(m_pEngine, "addpath('U:\\matlab\\rrr version phil\\geom3d\\geom3d')");
150     engEvalString(m_pEngine, "addpath('U:\\matlab\\rrr version phil\\geom3d\\meshes3d')");
151
152     //execution du script matlab.
153     engEvalString(m_pEngine, "slide_multi_3ddl_impl");
154
155     //changement d'etat dans le script
156     *pstate = (double)1;
157     engPutVariable(m_pEngine, "state", dstate);
158     engEvalString(m_pEngine, "slide_multi_3ddl_impl");
159
160     *pstate = (double)2;
161     engPutVariable(m_pEngine, "state", dstate);
162
163     // assignation des valeurs des position articulaires du bras jaco aux variable communiquant avec matlab
164     ptheta[0] = abs(((double)((dataPosition.Actuators.Actuator1 - 90.0) - 360.0) - 360.0)*3.14159265 / 180.0);
165     ptheta[1] = (double)((dataPosition.Actuators.Actuator2 - 0)*3.14159265 / 180.0);
166     ptheta[2] = abs(((double)((dataPosition.Actuators.Actuator3 - 210.0) - 360.0) - 360.0)*3.14159265 / 180.0);
167     ptheta_next[0] = abs(((double)((dataPosition.Actuators.Actuator1 - 90.0) - 360.0) - 360.0)*3.14159265 / 180.0);
168     ptheta_next[1] = (double)((dataPosition.Actuators.Actuator2 - 0)*3.14159265 / 180.0);
169     ptheta_next[2] = abs(((double)((dataPosition.Actuators.Actuator3 - 210.0) - 360.0) - 360.0)*3.14159265 / 180.0);
170
171     // envoi de ces valeurs
172     engPutVariable(m_pEngine, "theta", dtheta);
173
174     // Initialisation des pointeurs de fonctions de l'API de kinova. Ceci permet d'utiliser les fonctions dans ce script
175     MyInitAPI = (int(*)()) GetProcAddress(commandLayer_handle, "InitAPI");
176     MyCloseAPI = (int(*)()) GetProcAddress(commandLayer_handle, "CloseAPI");
177     MyGetDevices = (int(*)(KinovaDevice[MAX_KINOVA_DEVICE], int&)) GetProcAddress(commandLayer_handle, "GetDevices");
178     MySetActiveDevice = (int(*)(KinovaDevice)) GetProcAddress(commandLayer_handle, "SetActiveDevice");
179     MySendBasicTrajectory = (int(*)(TrajectoryPoint)) GetProcAddress(commandLayer_handle, "SendBasicTrajectory");
180     MyGetAngularCommand = (int(*)(AngularPosition &)) GetProcAddress(commandLayer_handle, "GetAngularCommand");
181     MyMoveHome = (int(*)()) GetProcAddress(commandLayer_handle, "MoveHome");
182     MyInitFingers = (int(*)()) GetProcAddress(commandLayer_handle, "InitFingers");
183     MyEraseAllTrajectories = (int(*)()) GetProcAddress(commandLayer_handle, "EraseAllTrajectories");
184     MyGetAngularPosition = (int(*)(AngularPosition &)) GetProcAddress(commandLayer_handle, "GetAngularPosition");
185
186     //Verify that all functions has been loaded correctly
187     if ((MyInitAPI == NULL) || (MyCloseAPI == NULL) || (MySendBasicTrajectory == NULL) ||
188         (MyGetDevices == NULL) || (MySetActiveDevice == NULL) || (MyGetAngularCommand == NULL) ||
189         (MyMoveHome == NULL) || (MyInitFingers == NULL))
190
191     {
192         cout << "* * *  E R R O R   D U R I N G   I N I T I A L I Z A T I O N  * * *" << endl;
193         programResult = 0;
194     }
195     else
196     {
197         cout << "I N I T I A L I Z A T I O N   C O M P L E T E D" << endl << endl;
198
199         int result = (*MyInitAPI)();
200
201         AngularPosition currentCommand;
202
203         cout << "Initialization's result :" << result << endl;
204
205         KinovaDevice list[MAX_KINOVA_DEVICE];
206         int devicesCount = MyGetDevices(list, result);
207
208         cout << "Found a robot on the USB bus (" << list[0].SerialNumber << ")" << endl;
```

```
209         MySetActiveDevice(list[0]);
210
211
212         TrajectoryPoint pointToSend;
213         int t = 0;
214         pointToSend.InitStruct();
215         MyMoveHome();
216         pointToSend.Position.Type = ANGULAR_VELOCITY;
217
218         t = 0;
219
220         MySendBasicTrajectory(pointToSend);
221
222         // definition des vitesses articulaires pour positionner le robot dans se position de départ
223
224         pointToSend.Position.Actuators.Actuator1 = 40;
225         pointToSend.Position.Actuators.Actuator2 = 0;
226         pointToSend.Position.Actuators.Actuator3 = 40;
227
228         // envoi des commandes au bras jaco
229         while (t < 600)
230         {
231             MySendBasicTrajectory(pointToSend);
232             t++;
233             Sleep(4);
234         }
235
236
237
238         SDL_Init(SDL_INIT_JOYSTICK);// initialise juste le joystick
239         SDL_Joystick *joystick; // on instancie
240         joystick = SDL_JoystickOpen(0); // on l'assigne au numero 0
241
242         time1 = getTime();
243         printf("time1 %f  \n", (time1));
244
245         if (quitter)
246         {
247             printf("Demarer le programme: boutton vert\n");
248             printf("arreter le programme : bouton rouge(apres avoir appuyer le bouton vert)\n");
249         }
250         while (quitter)
251         {
252             // lecture des boutons du joystick
253             SDL_JoystickUpdate();
254             if (SDL_JoystickGetButton(joystick, 0))
255             {
256                 printf("go");
257                 quitter = 0;
258                 //jacoInst.doReceiveData();
259             }
260         }
261         typedef std::chrono::high_resolution_clock Time;
262         auto start = Time::now();
263         auto start3 = Time::now();
264         auto temps_now = Time::now();
265         typedef std::chrono::microseconds micro_sec;
266         typedef std::chrono::duration<float> fsec;
267         fsec delta_t;
268         fsec delta_t3;
269         micro_sec dur = chrono::duration_cast<micro_sec>(delta_t);
270         micro_sec dur3 = chrono::duration_cast<micro_sec>(delta_t3);
271
272         pointToSend.Position.Type = ANGULAR_VELOCITY;
273
274
275         while (!quitter)
276             //for (int l = 0; l < 2000; l++ )
277         {
278             //lecture des positions articulaires du bras jaco
279             (*MyGetAngularPosition)(dataPosition);
280             temps_now = chrono::high_resolution_clock::now();
281             delta_t = temps_now - start;
282             dur = chrono::duration_cast<micro_sec>(delta_t);
283             // lecture des boutons du joystick
284             x_move = SDL_JoystickGetAxis(joystick, 0);
285             y_move = SDL_JoystickGetAxis(joystick, 1);
286             z_move_1 = SDL_JoystickGetAxis(joystick, 4);
287             z_move_2 = SDL_JoystickGetAxis(joystick, 5);
288             z_move = z_move_1;
289             if (abs((int)x_move) < 400) x_move = 0;
290             if (abs((int)y_move) < 400) y_move = 0;
291             if (abs((int)z_move) < 400) z_move = 0;
292
293
294             pmove[0] = (double)x_move / 32768;
295             pmove[1] = (double)y_move / 32768;
296             pmove[2] = (double)z_move / 32768;
297
298             // assignation des valeurs des position articulaires du bras jaco aux variable communiquant avec matlab
299             ptheta[0] = (double)((dataPosition.Actuators.Actuator1 + 270.0)*3.14159265 / 180.0);
300             ptheta[1] = (double)((dataPosition.Actuators.Actuator2 - 180.0)*3.14159265 / 180.0);
301             ptheta[2] = abs(((double)((dataPosition.Actuators.Actuator3 - 200.0) - 360.0) - 360.0)*3.14159265 / 180.0);
302             //ptheta[2] = (double)((dataPosition.Actuators.Actuator3 - 210.0)*3.14159265 / 180.0);
303             memcpy(mxGetPr(dmove), pmove, 3 * sizeof(double));
304             memcpy(mxGetPr(dtheta), ptheta, 3 * sizeof(double));
305
306             //envoi des variables vers matlab
307             engPutVariable(m_pEngine, "move", dmove);
308             engPutVariable(m_pEngine, "theta", dtheta);
309
310             // re-execution du script
311             engEvalString(m_pEngine, "slide_multi_3ddl_impl");
312
313             // lecture des valeurs mises a jour par le script
314             dtheta_delt = engGetVariable(m_pEngine, "delta_theta");
```

```
315             ans = mxGetData(dtheta_delt);
316             ptheta_delt[0] = ((double*)ans)[0];
317             ptheta_delt[1] = ((double*)ans)[1];
318             ptheta_delt[2] = ((double*)ans)[2];
319
320
321             // envoi des commandes au bras jaco
322             pointToSend.Position.Actuators.Actuator1 = ptheta_delt[0] / 0.01;
323             pointToSend.Position.Actuators.Actuator2 = ptheta_delt[1] / 0.01;
324             pointToSend.Position.Actuators.Actuator3 = -ptheta_delt[2] / 0.01;
325             MySendBasicTrajectory(pointToSend);
326
327
328             // lecture des boutons du joystick
329             SDL_JoystickUpdate();//nécessaire pour refaire la lecture des input du joystick (si non c'est tout le temps la meme valeur qui est lue
330
331             if (SDL_JoystickGetButton(joystick, 1))//on pousse le bouton rouge et le robot va vers home et on sort de la loop
332             {
333                 cout << endl << "WARNING: Your robot is now set to angular control. If you use the joystick, it will be a joint by joint movement." << endl;
334                 cout << endl << "C L O S I N G   A P I" << endl;
335                 quitter = 1;
336                 engEvalString(m_pEngine, "close all");
337                 engClose(m_pEngine);
338                 Sleep(1000);
339             }
340             if (SDL_JoystickGetButton(joystick, 2))//on pousse le bouton rouge et le robot va vers home et on sort de la loop
341             {
342                 engEvalString(m_pEngine, "plot3(pose_act(1,1), pose_act(1,2), pose_act(1,3), 'X', 'Color', 'b');");
343                 engEvalString(m_pEngine, "plot3(pose_act(3,1), pose_act(3,2), pose_act(3,3), 'X', 'Color', 'r');");
344             }
345
346             start3 = Time::now();
347             delta_t3 = temps_now - start3;
348             dur3 = chrono::duration_cast<micro_sec>(delta_t3);
349
350             while (abs(dur3.count()) < 9900)
351             {
352                 start3 = Time::now();
353                 delta_t3 = temps_now - start3;
354                 dur3 = chrono::duration_cast<micro_sec>(delta_t3);
355             }
356             //cout << dur3.count() << endl;
357         }
358         MyMoveHome();
359         result = (*MyCloseAPI)();
360         time2 = getTime();
361         Console::WriteLine("Example done...");
362         Console::WriteLine("Temps total:  " + (time2 - time1));
363
364         Thread::Sleep(2000);
365
366
367     }
368     return 0;
369
370 }
371
```

B

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/sˆ2)

`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/sˆ2)

`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

C

Fonctions de l'API VRep

Remote API Functions (Matlab)

simxAddStatusbarMessage (regular API equivalent: sim.addStatusbarMessage)

Description	Adds a message to the status bar.
Matlab synopsis	[number returnCode]=simxAddStatusbarMessage(number clientID,string message,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **message**: the message to display **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxAppendStringSignal

Description	DEPRECATED. Refer to simxWriteStringStream instead. Appends a string to a string signal. If that signal is not yet present, it is added. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats. See also simxSetStringSignal.
Matlab synopsis	[number returnCode]=simxAppendStringSignal(number clientID,string signalName,string signalValueToAppend,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValueToAppend**: value to append to the signal. That value may contain any value, including embedded zeros. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave

simxAuxiliaryConsoleClose (regular API equivalent: sim.auxiliaryConsoleClose)

Description	Closes an auxiliary console window. See also simxAuxiliaryConsoleOpen.
Matlab synopsis	[number returnCode]=simxAuxiliaryConsoleClose(number clientID,number consoleHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **consoleHandle**: the handle of the console window, previously returned by the simxAuxiliaryConsoleOpen command **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxAuxiliaryConsoleOpen (regular API equivalent: sim.auxiliaryConsoleOpen)

Description	Opens an auxiliary console window for text display. This console window is different from the application main console window. Console window handles are shared across all simulator scenes. See also simxAuxiliaryConsolePrint, simxAuxiliaryConsoleShow and simxAuxiliaryConsoleClose.
Matlab synopsis	[number returnCode,number consoleHandle]=simxAuxiliaryConsoleOpen(number clientID,string title,number maxLines,number mode,array position,array size,array textColor,array backgroundColor,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **title**: the title of the console window **maxLines**: the number of text lines that can be displayed and buffered **mode**: bit-coded value. Bit0 set indicates that the console window will automatically close at simulation end, bit1 set indicates that lines will be wrapped, bit2 set indicates that the user can close the console window, bit3 set indicates that the console will automatically be hidden during simulation pause, bit4 set indicates that the console will not automatically hide when the user

	switches to another scene. **position**: the initial position of the console window (x and y value). Can be [] for default values **size**: the initial size of the console window (x and y value). Can be [] for default values **textColor**: the color of the text (rgb values, 0-1). Can be [] for default values **backgroundColor**: the background color of the console window (rgb values, 0-1). Can be [] for default values **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **consoleHandle**: the handle of the created console
Other languages	C/C++, Python, Java, Octave, Lua

simxAuxiliaryConsolePrint (regular API equivalent: sim.auxiliaryConsolePrint)

Description	Prints to an auxiliary console window. See also simxAuxiliaryConsoleOpen.
Matlab synopsis	[number returnCode]=simxAuxiliaryConsolePrint(number clientID,number consoleHandle,string txt,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **consoleHandle**: the handle of the console window, previously returned by the simxAuxiliaryConsoleOpen command **txt**: the text to append, or [] to clear the console window **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxAuxiliaryConsoleShow (regular API equivalent: sim.auxiliaryConsoleShow)

Description	Shows or hides an auxiliary console window. See also simxAuxiliaryConsoleOpen and simxAuxiliaryConsoleClose.
Matlab synopsis	[number returnCode]=simxAuxiliaryConsoleShow(number clientID,number consoleHandle,boolean showState,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **consoleHandle**: the handle of the console window, previously returned by the simxAuxiliaryConsoleOpen command **showState**: indicates whether the console should be hidden (false) or shown (true) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxBreakForceSensor (regular API equivalent: sim.breakForceSensor)

Description	Allows breaking a force sensor during simulation. A broken force sensor will lose its positional and orientational constraints. See also simxReadForceSensor.
Matlab synopsis	[number returnCode]=simxBreakForceSensor(number clientID,number forceSensorHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **forceSensorHandle**: handle of the force sensor **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxCallScriptFunction (regular API equivalent: sim.callScriptFunction)

Description	Remotely calls a V-REP script function. When calling simulation scripts, then simulation must be running (and threaded scripts must still be running, i.e. not ended yet). Refer to this section for additional details.
Matlab synopsis	[number returnCode,array outInts,array outFloats,string outStrings,array outBuffer]=simxCallScriptFunction(number clientID,string scriptDescription,number scriptHandleOrType,string functionName,array inInts,array inFloats,string inStrings,array inBuffer,number operationMode)

Matlab parameters	**clientID**: the client ID. refer to simxStart. **scriptDescription**: the name of the scene object where the script is attached to, or an empty string if the script has no associated scene object. **scriptHandleOrType**: the handle of the script, otherwise the type of the script: *sim_scripttype_mainscript* (0): the main script will be called. *sim_scripttype_childscript* (1): a child script will be called. *sim_scripttype_customizationscript* (6): a customization script will be called. **functionName**: the name of the Lua function to call in the specified script. **inInts**: the input integer values that are handed over to the script function. Can be []. **inFloats**: the input floating-point values that are handed over to the script function. Can be []. **inStrings**: the input strings that are handed over to the script function. Each string should be terminated with a zero char, e.g. ['Hello' 0 'world!' 0]. Can be an empty string. **inBuffer**: the input buffer that is handed over to the script function. Can be []. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **outInts**: the returned integer values. **outFloats**: the returned floating-point values. **outStrings**: the returned strings. Each string is terminated with a zero char. **outBuffer**: the returned buffer.
Other languages	C/C++, Python, Java, Octave, Lua

simxClearFloatSignal (regular API equivalent: sim.clearFloatSignal)

Description	Clears a float signal (removes it). See also simxSetFloatSignal, simxClearIntegerSignal and simxClearStringSignal.
Matlab synopsis	[number returnCode]=simxClearFloatSignal(number clientID,string signalName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal or an empty string to clear all float signals **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxClearIntegerSignal (regular API equivalent: sim.clearIntegerSignal)

Description	Clears an integer signal (removes it). See also simxSetIntegerSignal, simxClearFloatSignal and simxClearStringSignal.
Matlab synopsis	[number returnCode]=simxClearIntegerSignal(number clientID,string signalName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal or an empty string to clear all integer signals **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxClearStringSignal (regular API equivalent: sim.clearStringSignal)

Description	Clears a string signal (removes it). See also simxSetStringSignal, simxClearIntegerSignal and simxClearFloatSignal.
Matlab synopsis	[number returnCode]=simxClearStringSignal(number clientID,string signalName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal or an empty string to clear all string signals **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxCloseScene (regular API equivalent: simCloseScene)

Description	Closes current scene, and switches to another open scene. If there is no other open scene, a new scene is then created. Should only be called when simulation is not running and is only executed

	by continuous remote API server services. See also simxLoadScene.
Matlab synopsis	[number returnCode]=simxCloseScene(number clientID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxCopyPasteObjects (regular API equivalent: sim.copyPasteObjects)

Description	Copies and pastes objects, together with all their associated calculation objects and child scripts. To copy and paste whole models, you can simply copy and paste the model base object.
Matlab synopsis	[number returnCode,array newObjectHandles]=simxCopyPasteObjects(number clientID,array objectHandles,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandles**: an array containing the handles of the objects to copy **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **newObjectHandles**: an array of handles of newly created objects. Individual objects of a new model are not returned, but only the model base.
Other languages	C/C++, Python, Java, Octave, Lua

simxCreateBuffer (regular API equivalent: simCreateBuffer)

Description	Creates a buffer. The buffer needs to be released with simxReleaseBuffer except otherwise explicitly specified. This is a remote API helper function.
Matlab synopsis	[libpointer buffer]=simxCreateBuffer(number bufferSize)
Matlab parameters	**bufferSize**: size of the buffer in bytes
Matlab return values	**buffer**: a pointer to the created buffer
Other languages	C/C++, Python

simxCreateDummy (regular API equivalent: sim.createDummy)

Description	Creates a dummy in the scene.
Matlab synopsis	[number returnCode,number dummyHandle]=simxCreateDummy(number clientID,number size,array colors,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **size**: the size of the dummy. **colors**: 4*3 bytes (0-255) for ambient_diffuse RGB, 3 reserved values (set to zero), specular RGB and emissive RGB. Can be [] for default colors. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return value	**returnCode**: a remote API function return code **dummyHandle**: handle of the created dummy.
Other languages	C/C++, Python, Java, Octave, Lua

simxDisplayDialog (regular API equivalent: sim.displayDialog)

Description	Displays a generic dialog box during simulation (and only during simulation!). Use in conjunction with simxGetDialogResult, simxGetDialogInput and simxEndDialog. Use custom user interfaces instead if a higher customization level is required.
Matlab synopsis	[number returnCode,number dialogHandle,number uiHandle]=simxDisplayDialog(number clientID,string titleText,string mainText,number dialogType,string initialText,array titleColors,array dialogColors,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **titleText**: Title bar text **mainText**: Information text **dialogType**: a generic dialog style **initialText**: Initial text in the edit box if the dialog is of type sim_dlgstyle_input. **titleColors**: Title bar color (6 number values for RGB for background and foreground), can be [] for default colors **dialogColors**: Dialog color (6 number values for RGB for background and foreground), can be [] for default colors **operationMode**: a remote API function operation mode. Recommended operation mode for this

	function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **dialogHandle**: handle of the generic dialog (different from an OpenGl-based custom UI handle!! (see hereafter)). This handle should be used with the following functions: simxGetDialogResult, simxGetDialogInput and simxEndDialog. **uiHandle**: the handle of the corresponding OpenGl-based custom UI.
Other languages	C/C++, Python, Java, Octave, Lua

simxEndDialog (regular API equivalent: sim.endDialog)

Description	Closes and releases resource from a previous call to simxDisplayDialog. Even if the dialog is not visible anymore, you should release resources by using this function (however at the end of a simulation, all dialog resources are automatically released).
Matlab synopsis	[number returnCode]=simxEndDialog(number clientID,number dialogHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **dialogHandle**: handle of generic dialog (return value of simxDisplayDialog) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxEraseFile

Description	Erases a file on the server side. This function is used by several other functions internally (e.g. simxLoadModel). See also simxTransferFile. This is a remote API helper function.
Matlab synopsis	[number returnCode]=simxEraseFile(number clientID,string fileName_serverSide,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **fileName_serverSide**: the file to erase on the server side. For now, do not specify a path (the file will be erased in the remote API plugin directory) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxFinish

Description	Ends the communication thread. This should be the very last remote API function called on the client side. simxFinish should only be called after a successfull call to simxStart. This is a remote API helper function.
Matlab synopsis	simxFinish(number clientID)
Matlab parameters	**clientID**: the client ID. refer to simxStart. Can be -1 to end all running communication threads.
Matlab return values	none
Other languages	C/C++, Python, Java, Octave, Lua

simxGetAndClearStringSignal

Description	DEPRECATED. Refer to simxReadStringStream instead. Gets the value of a string signal, then clears it. Useful to retrieve continuous data from the server. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats. See also simxGetStringSignal.
Matlab synopsis	[number returnCode,string signalValue]=simxGetAndClearStringSignal(number clientID,string signalName,number operationMode)

Matlab parameters	**clientID**: the client ID. refer to simxStart.
	signalName: name of the signal
	operationMode: a remote API function operation mode. Since this function will clear a read signal, and we cannot afford to wait for a reply (well, we could, but that would mean a blocking operation), the function operates in a special mode and should be used as in following example:
	
	`% Initialization phase:`
	`[err,signal]=vrep.simxGetAndClearStringSignal(clientID,'sig',`

	`vrep.simx_opmode_streaming);`
	`% while we are connected:`
	`while (vrep.simxGetConnectionId(clientID)~=-1)`
	`[err,signal]=vrep.simxGetAndClearStringSignal(clientID,'sig',`
	`vrep.simx_opmode_buffer);`
	`if (err==vrep.simx_return_ok)`
	`% A signal was retrieved.`
	`% Enable streaming again (was automatically disabled with the positive event):`
	`[err,signal]=vrep.simxGetAndClearStringSignal(clientID,'sig',`
	`vrep.simx_opmode_streaming);`
	`end`
	
	`..`
	
	`end`

Matlab return values	**returnCode**: a remote API function return code **signalValue**: the signal data (that may contain any value, including embedded zeros).
Other languages	C/C++, Python, Java, Octave

simxGetArrayParameter (regular API equivalent: sim.getArrayParameter)

Description	Retrieves 3 values from an array. See the array parameter identifiers. See also simxSetArrayParameter, simxGetBooleanParameter, simxGetIntegerParameter, simxGetFloatingParameter and simxGetStringParameter.
Matlab synopsis	[number returnCode,array paramValues]=simxGetArrayParameter(number clientID,number paramIdentifier,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: an array parameter identifier **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking (if not called on a regular basis)
Matlab return values	**returnCode**: a remote API function return code **paramValues**: 3 values representing the array parameter
Other languages	C/C++, Python, Java, Octave, Lua

simxGetBooleanParameter (regular API equivalent: sim.getBoolParameter)

Description	Retrieves a boolean value. See the Boolean parameter identifiers. See also simxSetBooleanParameter, simxGetIntegerParameter, simxGetFloatingParameter, simxGetArrayParameter and simxGetStringParameter.
Matlab synopsis	[number returnCode,boolean paramValue]=simxGetBooleanParameter(number clientID,number paramIdentifier,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: a Boolean parameter identifier **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking (if not called on a regular basis)
Matlab return values	**returnCode**: a remote API function return code **paramValue**: the parameter value
Other languages	C/C++, Python, Java, Octave, Lua

simxGetCollectionHandle (regular API equivalent: sim.getCollectionHandle)

Description	Retrieves a collection handle based on its name. If the client application is launched from a child script, then you could also let the child script figure out what handle correspond to what collection, and send the handles as additional arguments to the client application during its launch. See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number handle]=simxGetCollectionHandle(number clientID,string collectionName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **collectionName**: name of the collection. If possibe, don't rely on the automatic name adjustment mechanism, and always specify the full collection name, including the #: if the collection is 'myCollection', specify 'myCollection#', if the collection is 'myCollection#0', specify 'myCollection#0', etc. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **handle**: the handle
Other languages	C/C++, Python, Java, Octave, Lua

simxGetCollisionHandle (regular API equivalent: sim.getCollisionHandle)

Description	Retrieves a collision object handle based on its name. If the client application is launched from a child script, then you could also let the child script figure out what handle correspond to what collision object, and send the handles as additional arguments to the client application during its launch. See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number handle]=simxGetCollisionHandle(number clientID,string collisionObjectName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **collisionObjectName**: name of the collision object. If possibe, don't rely on the automatic name adjustment mechanism, and always specify the full collision object name, including the #: if the collision object is 'myCollision', specify 'myCollision#', if the collision object is 'myCollision#0', specify 'myCollision#0', etc. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **handle**: the handle
Other languages	C/C++, Python, Java, Octave, Lua

simxGetConnectionId

Description	Returns the ID of the current connection. Use this function to track the connection state to the server. See also simxStart. This is a remote API helper function.
Matlab synopsis	[number connectionId]=simxGetConnectionId(number clientID)
Matlab parameters	**clientID**: the client ID. refer to simxStart.
Matlab return values	**connectionId**: a connection ID, or -1 if the client is not connected to the server. Different connection IDs indicate temporary disconections in-between.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetDialogInput (regular API equivalent: sim.getDialogInput)

Description	Queries the text the user entered into a generic dialog box of style sim_dlgstyle_input. To be used after simxDisplayDialog was called and after simxGetDialogResult returned sim_dlgret_ok.
Matlab synopsis	[number returnCode,string inputText]=simxGetDialogInput(number clientID,number dialogHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **dialogHandle**: handle of generic dialog (return value of simxDisplayDialog) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **inputText**: the string the user entered.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetDialogResult (regular API equivalent: sim.getDialogResult)

Description	Queries the result of a dialog box. To be used after simxDisplayDialog was called.
Matlab synopsis	[number returnCode,number result]=simxGetDialogResult(number clientID,number dialogHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **dialogHandle**: handle of generic dialog (return value of simxDisplayDialog) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code **result**: the result value. Note. If the return value is sim_dlgret_still_open, the dialog was not closed and no button was pressed. Otherwise, you should free resources with simxEndDialog (the dialog might not be visible anymore, but is still present)
Other languages	C/C++, Python, Java, Octave, Lua

simxGetDistanceHandle (regular API equivalent: sim.getDistanceHandle)

Description	Retrieves a distance object handle based on its name. If the client application is launched from a child script, then you could also let the child script figure out what handle correspond to what distance object, and send the handles as additional arguments to the client application during its launch. See also simxGetObjectGroupData.

Matlab synopsis	[number returnCode,number handle]=simxGetDistanceHandle(number clientID,string distanceObjectName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **distanceObjectName**: name of the distance object. If possibe, don't rely on the automatic name adjustment mechanism, and always specify the full distance object name, including the #: if the distance object is 'myDistance', specify 'myDistance#', if the distance object is 'myDistance#0', specify 'myDistance#0', etc. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **handle**: the handle
Other languages	C/C++, Python, Java, Octave, Lua

simxGetFloatingParameter (regular API equivalent: sim.getFloatParameter)

Description	Retrieves a floating point value. See the floating-point parameter identifiers. See also simxSetFloatingParameter, simxGetBooleanParameter, simxGetIntegerParameter, simxGetArrayParameter and simxGetStringParameter.
Matlab synopsis	[number returnCode,number paramValue]=simxGetFloatingParameter(number clientID,number paramIdentifier,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: a floating parameter identifier **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking (if not called on a regular basis)
Matlab return values	**returnCode**: a remote API function return code **paramValue**: the parameter value
Other languages	C/C++, Python, Java, Octave, Lua

simxGetFloatSignal (regular API equivalent: sim.getFloatSignal)

Description	Gets the value of a float signal. Signals are cleared at simulation start. See also simxSetFloatSignal, simxClearFloatSignal, simxGetIntegerSignal and simxGetStringSignal.
Matlab synopsis	[number returnCode,number signalValue]=simxGetFloatSignal(number clientID,string signalName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **signalValue**: the value of the signal
Other languages	C/C++, Python, Java, Octave, Lua

simxGetInMessageInfo

Description	Retrieves information about the last received message from the server. This is a remote API helper function. See also simxGetOutMessageInfo. If the client didn't receive any command reply from the server for a while, the data retrieved with this function won't be up-to-date. In order to avoid this, you should start at least one streaming command, which will guarantee regular message income.
Matlab synopsis	[number result,number info]=simxGetInMessageInfo(number clientID,number infoType)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **infoType**: an inbox message info type
Matlab return values	**result**: -1 in case of an error **info**: the requested information
Other languages	C/C++, Python, Java, Octave, Lua

simxGetIntegerParameter (regular API equivalent: sim.getInt32Parameter)

Description	Retrieves an integer value. See the integer parameter identifiers. See also simxSetIntegerParameter, simxGetBooleanParameter, simxGetFloatingParameter, simxGetArrayParameter and simxGetStringParameter.
Matlab synopsis	[number returnCode,number paramValue]=simxGetIntegerParameter(number clientID,number paramIdentifier,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: an integer parameter identifier **operationMode**: a remote API function operation mode. Recommended operation mode for this

	function is simx_opmode_blocking (if not called on a regular basis)
Matlab return values	**returnCode**: a remote API function return code **paramValue**: the parameter value
Other languages	C/C++, Python, Java, Octave, Lua

simxGetIntegerSignal (regular API equivalent: sim.getIntegerSignal)

Description	Gets the value of an integer signal. Signals are cleared at simulation start. See also simxSetIntegerSignal, simxClearIntegerSignal, simxGetFloatSignal and simxGetStringSignal.
Matlab synopsis	[number returnCode,number signalValue]=simxGetIntegerSignal(number clientID,string signalName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **signalValue**: the value of the signal
Other languages	C/C++, Python, Java, Octave, Lua

simxGetJointForce (regular API equivalent: sim.getJointForce)

Description	Retrieves the force or torque applied to a joint along/about its active axis. This function retrieves meaningful information only if the joint is prismatic or revolute, and is dynamically enabled. With the Bullet engine, this function returns the force or torque applied to the joint motor (torques from joint limits are not taken into account). With the ODE or Vortex engine, this function returns the total force or torque applied to a joint along/about its z-axis. See also simxSetJointForce, simxReadForceSensor and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number force]=simxGetJointForce(number clientID,number jointHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **force**: the force or the torque applied to the joint along/about its z-axis
Other languages	C/C++, Python, Java, Octave, Lua

simxGetJointMatrix (regular API equivalent: sim.getJointMatrix)

Description	Retrieves the intrinsic transformation matrix of a joint (the transformation caused by the joint movement). See also simxSetSphericalJointMatrix.
Matlab synopsis	[number returnCode,array matrix]=simxGetJointMatrix(number clientID,number jointHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **matrix**: 12 number values. See the regular API equivalent function for details
Other languages	C/C++, Python, Java, Octave, Lua

simxGetJointPosition (regular API equivalent: sim.getJointPosition)

Description	Retrieves the intrinsic position of a joint. This function cannot be used with spherical joints (use simxGetJointMatrix instead). See also simxSetJointPosition and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number position]=simxGetJointPosition(number clientID,number jointHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **position**: intrinsic position of the joint. This is a one-dimensional value: if the joint is revolute, the rotation angle is returned, if the joint is prismatic, the translation amount is returned, etc.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetLastCmdTime

Description	Retrieves the simulation time of the last fetched command (i.e. when the last fetched command was processed on the server side). The function can be used to verify how *fresh* a command reply is, or whether a command reply was recently updated. For example:
	
	`[err,res,img]=vrep.simxGetVisionSensorImage(clientID,handle,0,vrep.simx_opmode_buffer);`
	`if (err==vrep.simx_return_ok)`
	`  imageAcquisitionTime=vrep.simxGetLastCmdTime(clientID);`
	`end`
	
	If some streaming commands are running, simxGetLastCmdTime will always retrieve the current simulation time, otherwise, only the simulation time of the last command that retrieved data from V-REP. This is a remote API helper function.

Matlab synopsis	[number cmdTime]=simxGetLastCmdTime(number clientID)
Matlab parameters	**clientID**: the client ID. refer to simxStart.
Matlab return values	**cmdTime**: the simulation time in milliseconds when the command reply was generated, or 0 if simulation was not running.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetLastErrors (regular API equivalent: sim.getLastError)

Description	Retrieves the last 50 errors that occured on the server side, and clears the error buffer there. Only errors that occured because of this client will be reported.
Matlab synopsis	[number returnCode,cell errorStrings]=simxGetLastErrors(number clientID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls) when not debugging. For debugging purposes, use simx_opmode_blocking.
Matlab return values	**returnCode**: a remote API function return code **errorStrings**: all error strings
Other languages	C/C++, Python, Java, Octave, Lua

simxGetModelProperty (regular API equivalent: sim.getModelProperty)

Description	Retrieves the properties of a model. See also simxSetModelProperty.
Matlab synopsis	[number returnCode,number prop]=simxGetModelProperty(number clientID,number objectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls), or simx_opmode_blocking (depending on the intended usage)
Matlab return values	**returnCode**: a remote API function return code **prop**: the model property value
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectChild (regular API equivalent: sim.getObjectChild)

Description	Retrieves the handle of an object's child object. See also simxGetObjectParent.
Matlab synopsis	[number returnCode,number childObjectHandle]=simxGetObjectChild(number clientID,number parentObjectHandle,number childIndex,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **parentObjectHandle**: handle of the object **childIndex**: zero-based index of the child's position. To retrieve all children of an object, call the function by increasing the index until the child handle is -1 **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **childObjectHandle**: the handle of the child object. If the value is -1, there is no child at the given index
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectFloatParameter (regular API equivalent: sim.getObjectFloatParameter)

Description	Retrieves a floating-point parameter of a object. See also simxSetObjectFloatParameter and simxGetObjectIntParameter.
Matlab synopsis	[number returnCode,number parameterValue]=simxGetObjectFloatParameter(number clientID,number objectHandle,number parameterID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **parameterID**: identifier of the parameter to retrieve. See the list of all possible object parameter identifiers **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls), or simx_opmode_blocking (depending on the intended usage)
Matlab return values	**returnCode**: a remote API function return code **parameterValue**: the value of the parameter
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectGroupData

Description	Simultaneously retrieves data of various objects in a V-REP scene.
Matlab synopsis	[number returnCode,array handles,array intData,array floatData,array stringData]=simxGetObjectGroupData(number clientID,number objectType,number dataType,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectType**: a scene object type, sim_appobj_object_type for all scene objects, or a collection handle. **dataType**: the type of data that is desired: 0: retrieves the object names (in stringData.) 1: retrieves the object types (in intData) 2: retrieves the parent object handles (in intData) 3: retrieves the absolute object positions (in floatData. There are 3 values for each object (x,y,z)) 4: retrieves the local object positions (in floatData. There are 3 values for each object (x,y,z)) 5: retrieves the absolute object orientations as Euler angles (in floatData. There are 3 values for each object (alpha,beta,gamma)) 6: retrieves the local object orientations as Euler angles (in floatData. There are 3 values for each object (alpha,beta,gamma)) 7: retrieves the absolute object orientations as quaternions (in floatData. There are 4 values for each object (qx,qy,qz,qw)) 8: retrieves the local object orientations as quaternions (in floatData. There are 4 values for each object (qx,qy,qz,qw)) 9: retrieves the absolute object positions and orientations (as Euler angles) (in floatData. There are 6 values for each object (x,y,z,alpha,beta,gamma)) 10: retrieves the local object positions and orientations (as Euler angles) (in floatData. There are 6 values for each object (x,y,z,alpha,beta,gamma)) 11: retrieves the absolute object positions and orientations (as quaternions) (in floatData. There are 7 values for each object (x,y,z,qx,qy,qz,qw)) 12: retrieves the local object positions and orientations (as quaternions) (in floatData. There are 7 values for each object (x,y,z,qx,qy,qz,qw)) 13: retrieves proximity sensor data (in intData (2 values): detection state, detected object handle. In floatData (6 values): detected point (x,y,z) and detected surface normal (nx,ny,nz)) 14: retrieves force sensor data (in intData (1 values): force sensor state. In floatData (6 values): force (fx,fy,fz) and torque (tx,ty,tz)) 15: retrieves joint state data (in floatData (2 values): position, force/torque) 16: retrieves joint properties data (in intData (2 values): joint type, joint mode (bit16=hybid operation). In floatData (2 values): joint limit low, joint range (-1.0 if joint is cyclic)) 17: retrieves the object linear velocity (in floatData. There are 3 values for each object (vx,vy,vz)) 18: retrieves the object angular velocity as Euler angles per seconds (in floatData. There are 3 values for each object (dAlpha,dBeta,dGamma)) 19: retrieves the object linear and angular velocity (in floatData. There are 6 values for each object (vx,vy,vz,dAlpha,dBeta,dGamma)) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking or simx_opmode_streaming.
Matlab return values	**returnCode**: a remote API function return code **handles**: the object handles. **intData**: the integer values. **floatData**: the float values. **stringData**: the string values.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectHandle (regular API equivalent: sim.getObjectHandle)

Description	Retrieves an object handle based on its name. If the client application is launched from a child script, then you could also let the child script figure out what handle correspond to what objects, and send the handles as additional arguments to the client application during its launch. See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number handle]=simxGetObjectHandle(number clientID,string objectName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectName**: name of the object. If possibe, don't rely on the automatic name adjustment mechanism, and always specify the full object name, including the #: if the object is 'myJoint', specify 'myJoint#', if the object is 'myJoint#0', specify 'myJoint#0', etc. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **handle**: the handle
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectIntParameter (regular API equivalent: sim.getObjectInt32Parameter)

Description	Retrieves an integer parameter of a object. See also simxSetObjectIntParameter and simxGetObjectFloatParameter.
Matlab synopsis	[number returnCode,number parameterValue]=simxGetObjectIntParameter(number clientID,number objectHandle,number parameterID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **parameterID**: identifier of the parameter to retrieve. See the list of all possible object parameter identifiers **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls), or simx_opmode_blocking (depending on the intended usage)
Matlab return values	**returnCode**: a remote API function return code **parameterValue**: the value of the parameter
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectOrientation (regular API equivalent: sim.getObjectOrientation)

Description	Retrieves the orientation (Euler angles) of an object. See also simxSetObjectOrientation, simxGetObjectQuaternion, simxGetObjectPosition and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,array eulerAngles]=simxGetObjectOrientation(number clientID,number objectHandle,number relativeToObjectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame we want the orientation. Specify -1 to retrieve the absolute orientation, sim_handle_parent to retrieve the orientation relative to the object's parent, or an object handle relative to whose reference frame you want the orientation **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **eulerAngles**: 3 values representing the Euler angles (alpha, beta and gamma)
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectParent (regular API equivalent: sim.getObjectParent)

Description	Retrieves the handle of an object's parent object. See also simxGetObjectChild and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number parentObjectHandle]=simxGetObjectParent(number clientID,number objectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **parentObjectHandle**: the handle of the parent object. If the value is -1, the object has no parent
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectPosition (regular API equivalent: sim.getObjectPosition)

Description	Retrieves the position of an object. See also simxSetObjectPosition, simxGetObjectOrientation, simxGetObjectQuaternion and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,array position]=simxGetObjectPosition(number clientID,number objectHandle,number relativeToObjectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame we want the position. Specify -1 to retrieve the absolute position, sim_handle_parent to retrieve the position relative to the object's parent, or an object handle relative to whose reference frame you want the position **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **position**: 3 values representing the position
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectQuaternion (regular API equivalent: sim.getObjectQuaternion)

Description	Retrieves the quaternion of an object. See also simxSetObjectQuaternion.
Matlab synopsis	[number returnCode,array quat]=simxGetObjectQuaternion(number clientID,number objectHandle,number relativeToObjectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame we want the quaternion. Specify -1 to retrieve the absolute quaternion, sim_handle_parent to retrieve the quaternion relative to the object's parent, or an object handle relative to whose reference frame you want the quaternion **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **quat**: 4 values representing the quaternion (x, y, z, w)
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjects (regular API equivalent: sim.getObjects)

Description	Retrieves object handles of a given type, or of all types (i.e. all object handles). See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,array objectHandles]=simxGetObjects(number clientID,number objectType,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectType**: object type (sim_object_shape_type, sim_object_joint_type, etc., or sim_handle_all for any type of object **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **objectHandles**: an array containing object handles.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectSelection (regular API equivalent: sim.getObjectSelection)

Description	Retrieves all selected object's handles. See also simxSetObjectSelection.
Matlab synopsis	[number returnCode,array objectHandles]=simxGetObjectSelection(number clientID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls), or simx_opmode_blocking depending on the intent.
Matlab return values	**returnCode**: a remote API function return code **objectHandles**: an array containing the handles of all selected objects
Other languages	C/C++, Python, Java, Octave, Lua

simxGetObjectVelocity (regular API equivalent: sim.getObjectVelocity)

Description	Retrieves the linear and angular velocity of an object. See also simxGetObjectPosition, simxGetObjectOrientation and simxGetObjectGroupData.

Matlab synopsis	[number returnCode,array linearVelocity,array angularVelocity]=simxGetObjectVelocity(number clientID,number objectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **linearVelocity**: 3 values representing the linear velocity (vx, vy, vz). **angularVelocity**: 3 values representing the angular velocity (dAlpha, dBeta, dGamma).
Other languages	C/C++, Python, Java, Octave, Lua

simxGetOutMessageInfo

Description	Retrieves information about the next message to send to the server. This is a remote API helper function. See also simxGetInMessageInfo.
Matlab synopsis	[number result,number info]=simxGetOutMessageInfo(number clientID,number infoType)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **infoType**: an outbox message info type
Matlab return values	**result**: -1 in case of an error **info**: the requested information
Other languages	C/C++, Python, Java, Octave, Lua

simxGetPingTime

Description	Retrieves the time needed for a command to be sent to the server, executed, and sent back. That time depends on various factors like the client settings, the network load, whether a simulation is running, whether the simulation is real-time, the simulation time step, etc. The function is blocking. This is a remote API helper function.
Matlab synopsis	[number returnCode,number pingTime]=simxGetPingTime(number clientID)
Matlab parameters	**clientID**: the client ID. refer to simxStart.
Matlab return values	**returnCode**: a remote API function return code **pingTime**: the ping time in milliseconds.
Other languages	C/C++, Python, Java, Octave, Lua

simxGetStringParameter (regular API equivalent: sim.getStringParameter)

Description	Retrieves a string value. See the string parameter identifiers. See also simxGetBooleanParameter, simxGetIntegerParameter, simxGetArrayParameter and simxGetFloatingParameter.
Matlab synopsis	[number returnCode,string paramValue]=simxGetStringParameter(number clientID,number paramIdentifier,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: a string parameter identifier **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking (if not called on a regular basis)
Matlab return values	**returnCode**: a remote API function return code **paramValue**: the parameter value (a string)
Other languages	C/C++, Python, Java, Octave, Lua

simxGetStringSignal (regular API equivalent: sim.getStringSignal)

Description	Gets the value of a string signal. Signals are cleared at simulation start. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats. See also simxSetStringSignal, simxReadStringStream, simxClearStringSignal, simxGetIntegerSignal and simxGetFloatSignal.
Matlab synopsis	[number returnCode,string signalValue]=simxGetStringSignal(number clientID,string signalName,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **signalValue**: the signal data (that may contain any value, including embedded zeros).
Other languages	C/C++, Python, Java, Octave, Lua

simxGetUIButtonProperty (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxGetUIEventButton (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxGetUIHandle (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxGetUISlider (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxGetVisionSensorDepthBuffer (regular API equivalent: sim.getVisionSensorDepthBuffer)

Description	Retrieves the depth buffer of a vision sensor as a pointer. The returned data doesn't make sense if sim.handleVisionSensor wasn't called previously (sim.handleVisionSensor is called by default in the main script if the vision sensor is not tagged as explicit handling). Use the simxGetLastCmdTime function to verify the *freshness* of the retrieved data. See also simxGetVisionSensorDepthBuffer2 and simxGetVisionSensorImage.
Matlab synopsis	[number returnCode,array resolution,libpointer buffer]=simxGetVisionSensorDepthBuffer(number clientID,number sensorHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)

Matlab return values	**returnCode**: a remote API function return code
	resolution: 2 number values representing the resolution of the image
	buffer: a libpointer object to the data. To access individual depth buffer pixels, use following code:
	`buffer.setDataType('singlePtr',1,resolution(1)*resolution(2));`
	`buffer.value(pixelIndex);`
	Values are in the range of 0-1 (0=closest to sensor, 1=farthest from sensor). The buffer remains valid until next call to a simx-function.

Other languages	C/C++, Python, Java, Octave, Lua

simxGetVisionSensorDepthBuffer2 (regular API equivalent: sim.getVisionSensorDepthBuffer)

Description	Retrieves the depth buffer of a vision sensor as an image array. The returned data doesn't make sense if sim.handleVisionSensor wasn't called previously (sim.handleVisionSensor is called by default in the main script if the vision sensor is not tagged as explicit handling). Use the simxGetLastCmdTime function to verify the *freshness* of the retrieved data. This function is much slower than simxGetVisionSensorDepthBuffer. See also simxGetVisionSensorImage.
Matlab synopsis	[number returnCode,array resolution,matrix buffer]=simxGetVisionSensorDepthBuffer2(number clientID,number sensorHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **resolution**: 2 number values representing the resolution of the image **buffer**: the depth buffer data. Values are in the range of 0-1 (0=closest to sensor, 1=farthest from sensor).
Other languages	C/C++, Python, Java, Octave, Lua

simxGetVisionSensorImage (regular API equivalent: sim.getVisionSensorImage)

Description	Retrieves the image of a vision sensor as a pointer. The returned data doesn't make sense if sim.handleVisionSensor wasn't called previously (sim.handleVisionSensor is called by default in the main script if the vision sensor is not tagged as explicit handling). Use the simxGetLastCmdTime function to verify the *freshness* of the retrieved data. See also simxGetVisionSensorImage2, simxSetVisionSensorImage, simxGetVisionSensorDepthBuffer and

	simxReadVisionSensor.
Matlab synopsis	[number returnCode,array resolution,libpointer image]=simxGetVisionSensorImage(number clientID,number sensorHandle,number options,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **options**: image options, bit-coded: bit0 set: each image pixel is a byte (greyscale image), otherwise each image pixel is a rgb byte-triplet **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)

Matlab return values	**returnCode**: a remote API function return code
	resolution: 2 number values representing the resolution of the image
	image: a libpointer object to the data. To access individual pixels, use following code:
	`image.setDataType('uint8Ptr',1,resolution(1)*resolution(2)*bytesPerPixel);`
	`image.value(pixelIndex);`
	Values are in the range of 0-255. The buffer remains valid until next call to a simx-function.

Other languages	C/C++, Python, Java, Octave, Lua

simxGetVisionSensorImage2 (regular API equivalent: sim.getVisionSensorImage)

Description	Retrieves the image of a vision sensor as an image array. The returned data doesn't make sense if sim.handleVisionSensor wasn't called previously (sim.handleVisionSensor is called by default in the main script if the vision sensor is not tagged as explicit handling). Use the simxGetLastCmdTime function to verify the *freshness* of the retrieved data. See also simxGetVisionSensorImage, simxSetVisionSensorImage, simxGetVisionSensorDepthBuffer and simxReadVisionSensor.
Matlab synopsis	[number returnCode,array resolution,matrix image]=simxGetVisionSensorImage2(number clientID,number sensorHandle,number options,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **options**: image options, bit-coded: bit0 set: each image pixel is a byte (greyscale image), otherwise each image pixel is a rgb byte-triplet **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **resolution**: 2 number values representing the resolution of the image **image**: the image data. Values are in the range of 0-255.
Other languages	C/C++, Python, Java, Octave, Lua

simxJointGetForce (REPRECATED)

Description	DEPRECATED. See simxGetJointForce instead.
Matlab synopsis	[number returnCode,number force]=simxJointGetForce(number clientID,number jointHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **force**: the force or the torque applied to the joint along/about its z-axis
Other languages	C/C++, Python, Java, Octave, Lua

simxLoadModel (regular API equivalent: sim.loadModel)

Description	Loads a previously saved model. See also simxLoadScene and simxTransferFile.
Matlab synopsis	[number returnCode,number baseHandle]=simxLoadModel(number clientID,string modelPathAndName,number options,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **modelPathAndName**: the model filename, including the path and extension ('ttm'). The file is relative to the client or server system depending on the **options** value (see next argument) **options**: options, bit-coded: bit0 set: the specified file is located on the client side (in that case the function will be blocking since the model first has to be transferred to the server). Otherwise it is located on the server side **operationMode**: a remote API function operation mode. Recommended operation mode for this

	function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code **baseHandle**: the loaded model base.
Other languages	C/C++, Python, Java, Octave, Lua

simxLoadScene (regular API equivalent: sim.loadScene)

Description	Loads a previously saved scene. Should only be called when simulation is not running and is only executed by continuous remote API server services. See also simxCloseScene, simxLoadModel, and simxTransferFile.
Matlab synopsis	[number returnCode]=simxLoadScene(number clientID,string scenePathAndName,number options,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **scenePathAndName**: the scene filename, including the path and extension ('ttt'). The file is relative to the client or server system depending on the **options** value (see next argument) **options**: options, bit-coded: bit0 set: the specified file is located on the client side (in that case the function will be blocking since the scene first has to be transferred to the server). Otherwise it is located on the server side **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxLoadUI (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxPackFloats

Description	Packs an array of floats into a string. This is a remote API helper function. See also simxUnpackFloats and simxPackInts.
Matlab synopsis	[string packedData]=simxPackFloats(array floatValues)
Matlab parameters	**floatValues**: an array of numbers we wish to pack as floats
Matlab return values	**packedData**: a string that contains the packed values. Each values takes exactly 4 bytes in the string.
Other languages	Java, Octave, Python, Lua

simxPackInts

Description	Packs an array of integers into a string. This is a remote API helper function. See also simxUnpackInts and simxPackFloats.
Matlab synopsis	[string packedData]=simxPackInts(array intValues)
Matlab parameters	**intValues**: an array of numbers we wish to pack as integers
Matlab return values	**packedData**: a string that contains the packed values. Each values takes exactly 4 bytes in the string.
Other languages	Java, Octave, Python, Lua

simxPauseCommunication

Description	Allows to temporarily halt the communication thread from sending data. This can be useful if you need to send several values to V-REP that should be received and evaluated at the same time. This is a remote API helper function.
Matlab synopsis	[number returnCode]=simxPauseCommunication(number clientIDboolean pause)

Matlab parameters	**clientID**: the client ID. refer to simxStart.
	pause: whether the communication thread should pause or run normally.
	
	Usage example:
	
	`vrep.simxPauseCommunication(clientID,1);`
	`vrep.simxSetJointPosition(clientID,joint1Handle,joint1Value,vrep.simx_opmode_oneshot);`
	`vrep.simxSetJointPosition(clientID,joint2Handle,joint2Value,vrep.simx_opmode_oneshot);`
	`vrep.simxSetJointPosition(clientID,joint3Handle,joint3Value,vrep.simx_opmode_oneshot);`
	`vrep.simxPauseCommunication(clientID,0);`

	`% Above's 3 joints will be received and set on the V-REP side at the same time`
Matlab return values	**returnCode**: 0 in case of operation success.
Other languages	C/C++, Python, Java, Octave, Lua

simxPauseSimulation (regular API equivalent: sim.pauseSimulation)

Description	Requests a pause of a simulation. See also simxStartSimulation and simxStopSimulation.
Matlab synopsis	[number returnCode]=simxPauseSimulation(number clientID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **operationMode**: a remote API function operation mode. Recommended operation modes for this function is simx_opmode_oneshot.
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxQuery

Description	DEPRECATED. Refer to simxCallScriptFunction instead.
	
	Sends a query string to V-REP, and waits for a reply string. Query and reply strings can be accessed via string signals. This function allows for instance to have a child script, another remote API client or a ROS node handle special requests coming from this remote API client, then send a reply back. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats.
	
	Usage example where a child script handles a request:
	
	`% Following is the remote API client side:`
	`[res,replyData]=vrep.simxQuery(clientID,'request','send me a 42','reply',5000)`
	`if (res==vrep.simx_return_ok)`
	`  fprintf('The reply is: %s\n',replyData);`
	`end`
	
	`-- This is the child script side. The child script is non-threaded and`
	`-- following part executed at each simulation pass:`
	`req=sim.getStringSignal('request')`
	`if (req) then`
	`  sim.clearStringSignal('request')`
	`  if (req=='send me a 42') then`
	`  sim.setStringSignal('reply','42\0') -- will be automatically cleared by the client`
	`  end`
	`end`

Matlab synopsis	[number returnCode string retSignalValue]=simxQuery(number clientID,string signalName,string signalValue,string retSignalName,number timeOutInMs)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal that contains the request string **signalValue**: the request string. **retSignalName**: name of the signal that contains the reply string **timeOutInMs**: the maximum time in milliseconds that the function will wait for a reply.
Matlab return value	**returnCode**: a remote API function return code **retSignalValue**: the reply string
Other languages	C/C++, Python, Java, Octave, Lua

simxReadCollision (regular API equivalent: sim.readCollision)

Description	Reads the collision state of a registered collision object. This function doesn't perform collision detection, it merely reads the result from a previous call to sim.handleCollision (sim.handleCollision is called in the default main script). See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,boolean collisionState]=simxReadCollision(number clientID,number collisionObjectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **collisionObjectHandle**: handle of the collision object **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)

Matlab return values	**returnCode**: a remote API function return code **collisionState**: the collision state (false: not colliding)
Other languages	C/C++, Python, Java, Octave, Lua

simxReadDistance (regular API equivalent: sim.readDistance)

Description	Reads the distance that a registered distance object measured. This function doesn't perform minimum distance calculation, it merely reads the result from a previous call to sim.handleDistance (sim.handleDistance is called in the default main script). See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number minimumDistance]=simxReadDistance(number clientID,number distanceObjectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **distanceObjectHandle**: handle of the distance object **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **minimumDistance**: the minimum distance. If the distance object wasn't handled yet, the distance value will be larger than 1e36.
Other languages	C/C++, Python, Java, Octave, Lua

simxReadForceSensor (regular API equivalent: sim.readForceSensor)

Description	Reads the force and torque applied to a force sensor (filtered values are read), and its current state ('unbroken' or 'broken'). See also simxBreakForceSensor, simxGetJointForce and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,number state,array forceVector,array torqueVector]=simxReadForceSensor(number clientID,number forceSensorHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **forceSensorHandle**: handle of the force sensor **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **state**: the state of the force sensor: bit 0 set: force and torque data is available, otherwise it is not (yet) available (e.g. when not enough values are present for the filter) bit 1 set: force sensor is broken, otherwise it is still intact ('unbroken') **forceVector**: 3 values representing the force vector **torqueVector**: 3 values representing the torque vector
Other languages	C/C++, Python, Java, Octave, Lua

simxReadProximitySensor (regular API equivalent: sim.readProximitySensor)

Description	Reads the state of a proximity sensor. This function doesn't perform detection, it merely reads the result from a previous call to sim.handleProximitySensor (sim.handleProximitySensor is called in the default main script). See also simxGetObjectGroupData.
Matlab synopsis	[number returnCode,boolean detectionState,array detectedPoint,number detectedObjectHandle,array detectedSurfaceNormalVector]=simxReadProximitySensor(number clientID,number sensorHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the proximity sensor **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **detectionState**: the detection state (false=no detection) **detectedPoint**: the detected point coordinates (relative to the sensor reference frame) **detectedObjectHandle**: the handle of the detected object **detectedSurfaceNormalVector**: the normal vector (normalized) of the detected surface. Relative to the sensor reference frame
Other languages	C/C++, Python, Java, Octave, Lua

simxReadStringStream

Description	Gets the value of a string signal, then clears it. Useful to retrieve continuous data from the server. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats. See also simxWriteStringStream.
Matlab synopsis	[number returnCode,string signalValue]=simxReadStringStream(number clientID,string signalName,number operationMode)

Matlab parameters	**clientID**: the client ID. refer to simxStart.
	signalName: name of the signal
	operationMode: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls). simx_opmode_blocking is forbidden. Use a construction like following in order to continuously exchange data with V-REP:
	
	Remote API client side:
	
	`% Initialization phase:`
	`[err,signal]=vrep.simxReadStringStream(clientID,'toClient',`
	`vrep.simx_opmode_streaming);`
	`% while we are connected:`
	`while (vrep.simxGetConnectionId(clientID)~=-1)`
	`[err,signal]=vrep.simxReadStringStream(clientID,'toClient',`
	`vrep.simx_opmode_buffer);`
	`if (err==vrep.simx_return_ok)`
	`% Data produced by the child script was retrieved! Send it back to the child script:`
	`vrep.simxWriteStringStream(clientID,'fromClient',signal,`
	`vrep.simx_opmode_oneshot);`
	`end`
	`end`
	
	Server side (V-REP), from a non-threaded child script:
	
	`function sysCall_init()`
	`-- initialization phase:`
	`i=0`
	`lastReceived=-1`
	`end`
	
	`function sysCall_actuation()`
	`-- First send a stream of integers that count up:`
	`dat=sim.getStringSignal('toClient')`
	`if not dat then`
	`dat=''`
	`end`
	`dat=dat..sim.packInt32Table({i})`
	`i=i+1`
	`sim.setStringSignal('toClient',dat)`
	
	`-- Here receive the integer stream in return and check if each number is correct:`
	`dat=sim.getStringSignal('fromClient')`
	`if dat then`
	`sim.clearStringSignal('fromClient')`
	`dat=sim.unpackInt32Table(dat)`
	`for j=1,#dat,1 do`
	`if (dat[j]~=lastReceived+1) then`
	`print('Error')`
	`else`
	`io.write('.')`
	`lastReceived=dat[j]`
	`end`
	`end`
	`end`
	`end`

Matlab return values	**returnCode**: a remote API function return code **signalValue**: the signal data (that may contain any value, including embedded zeros)
Other languages	C/C++, Python, Java, Octave, Lua

simxReadVisionSensor (regular API equivalent: sim.readVisionSensor)

Description	Reads the state of a vision sensor. This function doesn't perform detection, it merely reads the result from a previous call to sim.handleVisionSensor (sim.handleVisionSensor is called in the default main script). See also simxGetVisionSensorImage and simxGetObjectGroupData.
Matlab synopsis	[number returnCode,boolean detectionState,array auxData,array auxPacketInfo]=simxReadVisionSensor(number clientID,number sensorHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_streaming (the first call) and simx_opmode_buffer (the following calls)
Matlab return values	**returnCode**: a remote API function return code **detectionState**: the detection state (i.e. the trigger state) **auxData**: all auxiliary values returned from the applied filters. By default V-REP returns one packet of 15 auxiliary values:the minimum of {intensity, red, green, blue, depth value}, the maximum of {intensity, red, green, blue, depth value}, and the average of {intensity, red, green, blue, depth

	value}. If additional filter components return values, then they will be appended as packets after the first packet. **auxPacketInfo**: an array containing an entry for each returned packet. Each entry represents the number of values in each packets. Use this info to extract individual packets from **auxData**.
Other languages	C/C++, Python, Java, Octave, Lua

simxReleaseBuffer (regular API equivalent: simReleaseBuffer)

Description	Releases a buffer previously created with simxCreateBuffer or a buffer returned by a remote API function. This is a remote API helper function.
Matlab synopsis	simxReleaseBuffer(libpointer buffer)
Matlab parameters	**buffer**: buffer to be released
Matlab return values	none
Other languages	C/C++, Python

simxRemoveModel (regular API equivalent: sim.removeModel)

Description	Removes a model from the scene. See also simxRemoveObject.
Matlab synopsis	[number returnCode]=simxRemoveModel(number clientID,number objectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the model to remove (object should be flagged as *model base*). **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot (or simx_opmode_blocking)
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxRemoveObject (regular API equivalent: sim.removeObject)

Description	Removes a scene object. See also simxRemoveModel.
Matlab synopsis	[number returnCode]=simxRemoveObject(number clientID,number objectHandle,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object to remove **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot (or simx_opmode_blocking)
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxRemoveUI (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxSetArrayParameter (regular API equivalent: sim.setArrayParameter)

Description	Sets 3 values of an array parameter. See also simxGetArrayParameter, simxSetBooleanParameter, simxSetIntegerParameter and simxSetFloatingParameter.
Matlab synopsis	[number returnCode]=simxSetArrayParameter(number clientID,number paramIdentifier,array paramValues,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: an array parameter identifier **paramValues**: the array containing the 3 values to set **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetBooleanParameter (regular API equivalent: sim.setBoolParameter)

Description	Sets a boolean parameter. See also simxGetBooleanParameter, simxSetIntegerParameter,

	simxSetArrayParameter and simxSetFloatingParameter.
Matlab synopsis	[number returnCode]=simxSetBooleanParameter(number clientID,number paramIdentifier,boolean paramValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: a Boolean parameter identifier **paramValue**: the parameter value **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetFloatingParameter (regular API equivalent: sim.setFloatParameter)

Description	Sets a floating point parameter. See also simxGetFloatingParameter, simxSetBooleanParameter, simxSetArrayParameter and simxSetIntegerParameter.
Matlab synopsis	[number returnCode]=simxSetFloatingParameter(number clientID,number paramIdentifier,number paramValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: a floating parameter identifier **paramValue**: the parameter value **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetFloatSignal (regular API equivalent: sim.setFloatSignal)

Description	Sets the value of a float signal. If that signal is not yet present, it is added. See also simxGetFloatSignal, simxClearFloatSignal, simxSetIntegerSignal and simxSetStringSignal.
Matlab synopsis	[number returnCode]=simxSetFloatSignal(number clientID,string signalName,number signalValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValue**: value of the signal **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetIntegerParameter (regular API equivalent: sim.setInt32Parameter)

Description	Sets an integer parameter. See also simxGetIntegerParameter, simxSetBooleanParameter, simxSetArrayParameter and simxSetFloatingParameter.
Matlab synopsis	[number returnCode]=simxSetIntegerParameter(number clientID,number paramIdentifier,number paramValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **paramIdentifier**: an integer parameter identifier **paramValue**: the parameter value **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetIntegerSignal (regular API equivalent: sim.setIntegerSignal)

Description	Sets the value of an integer signal. If that signal is not yet present, it is added. See also simxGetIntegerSignal, simxClearIntegerSignal, simxSetFloatSignal and simxSetStringSignal.
Matlab synopsis	[number returnCode]=simxSetIntegerSignal(number clientID,string signalName,number signalValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValue**: value of the signal **operationMode**: a remote API function operation mode. Recommended operation mode for this

	function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetJointForce (regular API equivalent: sim.setJointForce)

Description	Sets the maximum force or torque that a joint can exert. This function has no effect when the joint is not dynamically enabled, or when it is a spherical joint. See also simxGetJointForce.
Matlab synopsis	[number returnCode]=simxSetJointForce(number clientID,number jointHandle,number force,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **force**: the maximum force or torque that the joint can exert **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetJointPosition (regular API equivalent: sim.setJointPosition)

Description	Sets the intrinsic position of a joint. May have no effect depending on the joint mode. This function cannot be used with spherical joints (use simxSetSphericalJointMatrix instead). If you want to set several joints that should be applied at the exact same time on the V-REP side, then use simxPauseCommunication. See also simxGetJointPosition and simxSetJointTargetPosition.
Matlab synopsis	[number returnCode]=simxSetJointPosition(number clientID,number jointHandle,number position,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **position**: position of the joint (angular or linear value depending on the joint type) **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_oneshot or simx_opmode_streaming
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetJointTargetPosition (regular API equivalent: sim.setJointTargetPosition)

Description	Sets the target position of a joint if the joint is in torque/force mode (also make sure that the joint's motor and position control are enabled). See also simxSetJointPosition.
Matlab synopsis	[number returnCode]=simxSetJointTargetPosition(number clientID,number jointHandle,number targetPosition,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **targetPosition**: target position of the joint (angular or linear value depending on the joint type) **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_oneshot or simx_opmode_streaming
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetJointTargetVelocity (regular API equivalent: sim.setJointTargetVelocity)

Description	Sets the intrinsic target velocity of a non-spherical joint. This command makes only sense when the joint mode is in torque/force mode: the dynamics functionality and the joint motor have to be enabled (position control should however be disabled)
Matlab synopsis	[number returnCode]=simxSetJointTargetVelocity(number clientID,number jointHandle,number targetVelocity,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **targetVelocity**: target velocity of the joint (linear or angular velocity depending on the joint-type) **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_oneshot or simx_opmode_streaming
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetModelProperty (regular API equivalent: sim.setModelProperty)

Description	Sets the properties of a model. See also simxGetModelProperty.
Matlab synopsis	[number returnCode]=simxSetModelProperty(number clientID,number objectHandle,number prop,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **prop**: a model property value **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectFloatParameter (regular API equivalent: sim.setObjectFloatParameter)

Description	Sets a floating-point parameter of a object. See also simxGetObjectFloatParameter and simxSetObjectIntParameter.
Matlab synopsis	[number returnCode]=simxSetObjectFloatParameter(number clientID,number objectHandle,number parameterID,number parameterValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **parameterID**: identifier of the parameter to set. See the list of all possible object parameter identifiers **parameterValue**: the desired value of the parameter **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectIntParameter (regular API equivalent: sim.setObjectInt32Parameter)

Description	Sets an integer parameter of a object. See also simxGetObjectIntParameter and simxSetObjectFloatParameter.
Matlab synopsis	[number returnCode]=simxSetObjectIntParameter(number clientID,number objectHandle,number parameterID,number parameterValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **parameterID**: identifier of the parameter to set. See the list of all possible object parameter identifiers **parameterValue**: the desired value of the parameter **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectOrientation (regular API equivalent: sim.setObjectOrientation)

Description	Sets the orientation (Euler angles) of an object. Dynamically simulated objects will implicitely be reset before the command is applied (i.e. similar to calling sim.resetDynamicObject just before). See also simxGetObjectOrientation, simxSetObjectQuaternion and simxSetObjectPosition.
Matlab synopsis	[number returnCode]=simxSetObjectOrientation(number clientID,number objectHandle,number relativeToObjectHandle,array eulerAngles,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame the orientation is specified. Specify -1 to set the absolute orientation, sim_handle_parent to set the orientation relative to the object's parent, or an object handle relative to whose reference frame the orientation is specified. **eulerAngles**: Euler angles (alpha, beta and gamma) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectParent (regular API equivalent: sim.setObjectParent)

Description	Sets an object's parent object. See also simxGetObjectParent.
Matlab synopsis	[number returnCode]=simxSetObjectParent(number clientID,number objectHandle,number parentObject,boolean keepInPlace,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object that will become child of the parent object. Can be combined with sim_handleflag_assembly, if the two objects can be assembled via a predefined assembly transformation (refer to the **assembling** option in the object common properties). In that case, **parentObject** can't be -1, and **keepInPlace** should be set to false. **parentObject**: handle of the object that will become parent, or -1 if the object should become parentless **keepInPlace**: indicates whether the object's absolute position and orientation should stay same **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot or simx_opmode_blocking depending on the intent
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectPosition (regular API equivalent: sim.setObjectPosition)

Description	Sets the position of an object. Dynamically simulated objects will implicitely be reset before the command is applied (i.e. similar to calling sim.resetDynamicObject just before). See also simxGetObjectPosition, simxSetObjectQuaternion and simxSetObjectOrientation.
Matlab synopsis	[number returnCode]=simxSetObjectPosition(number clientID,number objectHandle,number relativeToObjectHandle,array position,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame the position is specified. Specify -1 to set the absolute position, sim_handle_parent to set the position relative to the object's parent, or an object handle relative to whose reference frame the position is specified. **position**: the position values (x, y and z) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectQuaternion (regular API equivalent: sim.setObjectQuaternion)

Description	Sets the orientation of an object as quaternion. Dynamically simulated objects will implicitely be reset before the command is applied (i.e. similar to calling sim.resetDynamicObject just before). See also simxGetObjectQuaternion.
Matlab synopsis	[number returnCode]=simxSetObjectQuaternion(number clientID,number objectHandle,number relativeToObjectHandle,array quat,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame the quaternion is specified. Specify -1 to set the absolute quaternion, sim_handle_parent to set the quaternion relative to the object's parent, or an object handle relative to whose reference frame the quaternion is specified. **quat**: the quaternion values (x, y, z, w) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetObjectSelection

Description	Sets the selection state for objects. See also simxGetObjectSelection.
Matlab synopsis	[number returnCode]=simxSetObjectSelection(number clientID,array objectHandles,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **objectHandles**: an array of object handles **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot or simx_opmode_blocking depending on the intent.

Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetSphericalJointMatrix (regular API equivalent: sim.setSphericalJointMatrix)

Description	Sets the intrinsic orientation matrix of a spherical joint object. This function cannot be used with non-spherical joints (use simxSetJointPosition instead). See also simxGetJointMatrix..
Matlab synopsis	[number returnCode]=simxSetSphericalJointMatrix(number clientID,number jointHandle,array matrix,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **jointHandle**: handle of the joint **matrix**: 12 number values. See the regular API equivalent function for details **operationMode**: a remote API function operation mode. Recommended operation modes for this function are simx_opmode_oneshot or simx_opmode_streaming
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetStringSignal (regular API equivalent: sim.setStringSignal)

Description	Sets the value of a string signal. If that signal is not yet present, it is added. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats. See also simxWriteStringStream, simxGetStringSignal, simxClearStringSignal, simxSetIntegerSignal and simxSetFloatSignal.
Matlab synopsis	[number returnCode]=simxSetStringSignal(number clientID,string signalName,string signalValue,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValue**: value of the signal (which may contain any value, including embedded zeros) **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetUIButtonLabel (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxSetUIButtonProperty (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxSetUISlider (DEPRECATED)

Description	DEPRECATED. Use the Qt-based custom user interfaces, via simxCallScriptFunction instead.

simxSetVisionSensorImage (regular API equivalent: sim.setVisionSensorImage)

Description	Sets the image of a vision sensor (and applies any image processing filter if specified in the vision sensor dialog). The image is provided as a libpointer. Make sure the vision sensor is flagged as **use external image**. The *regular* use of this function is to first read the data from a vision sensor with simxGetVisionSensorImage, do some custom filtering, then write the modified image to a passive vision sensor. The alternate use of this function is to display textures, video images, etc. by using a vision sensor object (without however making use of the vision sensor functionality), since a vision sensor can be *looked through* like camera objects. See also simxSetVisionSensorImage2.
Matlab synopsis	[number returnCode]=simxSetVisionSensorImage(number clientID,number sensorHandle,libpointer image,number bufferSize,number options,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **image**: the image data **bufferSize**: size of the image data **options**: image options, bit-coded:

	bit0 set: each image pixel is a byte (greyscale image), otherwise each image pixel is a rgb byte-triplet **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSetVisionSensorImage2 (regular API equivalent: sim.setVisionSensorImage)

Description	Sets the image of a vision sensor (and applies any image processing filter if specified in the vision sensor dialog). The image is provided as an image array. Make sure the vision sensor is flagged as **use external image**. The *regular* use of this function is to first read the data from a vision sensor with simxGetVisionSensorImage2, do some custom filtering, then write the modified image to a passive vision sensor. The alternate use of this function is to display textures, video images, etc. by using a vision sensor object (without however making use of the vision sensor functionality), since a vision sensor can be *looked through* like camera objects. See also simxSetVisionSensorImage which is much faster.
Matlab synopsis	[number returnCode]=simxSetVisionSensorImage2(number clientID,number sensorHandle,matrix image,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **sensorHandle**: handle of the vision sensor **image**: the image data **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxStart

Description	Starts a communication thread with the server (i.e. V-REP). A same client may start several communication threads (but only one communication thread for a given IP and port). This should be the very first remote API function called on the client side. Make sure to start an appropriate remote API server service on the server side, that will wait for a connection. See also simxFinish. This is a remote API helper function.
Matlab synopsis	[number clientID]=simxStart(string connectionAddress,number connectionPort,boolean waitUntilConnected,boolean doNotReconnectOnceDisconnected,number timeOutInMs,number commThreadCycleInMs)
Matlab parameters	**connectionAddress**: the ip address where the server is located (i.e. V-REP) **connectionPort**: the port number where to connect. Specify a negative port number in order to use shared memory, instead of socket communication. **waitUntilConnected**: if true, then the function blocks until connected (or timed out). **doNotReconnectOnceDisconnected**: if true, then the communication thread will not attempt a second connection if a connection was lost. **timeOutInMs**: if positive: the connection time-out in milliseconds for the first connection attempt. In that case, the time-out for blocking function calls is 5000 milliseconds. if negative: its positive value is the time-out for blocking function calls. In that case, the connection time-out for the first connection attempt is 5000 milliseconds. **commThreadCycleInMs**: indicates how often data packets are sent back and forth. Reducing this number improves responsiveness, and a default value of 5 is recommended.
Matlab return values	**clientID**: the client ID, or -1 if the connection to the server was not possible (i.e. a timeout was reached). A call to simxStart should always be followed at the end with a call to simxFinish if simxStart didn't return -1
Other languages	C/C++, Python, Java, Octave, Lua

simxStartSimulation (regular API equivalent: sim.startSimulation)

Description	Requests a start of a simulation (or a resume of a paused simulation). This function is only executed by continuous remote API server services. See also simxPauseSimulation and simxStopSimulation.
Matlab synopsis	[number returnCode]=simxStartSimulation(number clientID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot.
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxStopSimulation (regular API equivalent: sim.stopSimulation)

Description	Requests a stop of the running simulation. See also simxStartSimulation and simxPauseSimulation.
Matlab synopsis	[number returnCode]=simxStopSimulation(number clientID,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **operationMode**: a remote API function operation mode. Recommended operation modes for this function is simx_opmode_oneshot.
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSynchronous

Description	Enables or disables the synchronous operation mode for the remote API server service that the client is connected to. The function is blocking. While in synchronous operation mode, the client application is in charge of triggering the next simulation step. Only pre-enabled remote API server services will successfully execute this function. See also simxSynchronousTrigger and this section. This is a remote API helper function.
Matlab synopsis	[number returnCode]=simxSynchronous(number clientID,boolean enable)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **enable**: the enable state of the synchronous operation
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxSynchronousTrigger

Description	Sends a synchronization trigger signal to the server. The function is blocking. The server needs to be previously enabled for synchronous operation via the simxSynchronous function. The trigger signal will inform V-REP to execute the next simulation step (i.e. to call simHandleMainScript). While in synchronous operation mode, the client application is in charge of triggering the next simulation step, otherwise simulation will stall. See also this section. This is a remote API helper function.
Matlab synopsis	[number returnCode]=simxSynchronousTrigger(number clientID)
Matlab parameters	**clientID**: the client ID. refer to simxStart.
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxTransferFile

Description	Allows transferring a file from the client to the server. This function is used by several other functions internally (e.g. simxLoadModel). See also simxEraseFile. This is a remote API helper function.
Matlab synopsis	[number returnCode]=simxTransferFile(number clientID,string filePathAndName,string fileName_serverSide,number timeOut,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **filePathAndName**: the local file name and path (i.e. on the client side) **fileName_serverSide**: a file name under which the transferred file will be saved on the server side. For now, do not specify a path (the file will be saved in the remote API plugin directory) **timeOut**: a timeout value in milliseconds **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_blocking
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua

simxUnpackFloats

Description	Unpacks a string into an array of floats. This is a remote API helper function. See also simxPackFloats and simxUnpackInts.
Matlab synopsis	[array floatValues]=simxUnpackFloats(string packedData)

Matlab parameters	**packedData**: a string that contains the packed values. Each values takes exactly 4 bytes in the string.
Matlab return values	**floatValues**: an array of numbers that were unpacked as floats
Other languages	Java, Octave, Python, Lua

simxUnpackInts

Description	Unpacks a string into an array of integers. This is a remote API helper function. See also simxPackInts and simxUnpackFloats.
Matlab synopsis	[array intValues]=simxUnpackInts(string packedData)
Matlab parameters	**packedData**: a string that contains the packed values. Each values takes exactly 4 bytes in the string.
Matlab return values	**intValues**: an array of numbers that were unpacked as integers
Other languages	Java, Octave, Python, Lua

simxWriteStringStream

Description	Appends a string to a string signal. If that signal is not yet present, it is added. To pack/unpack integers/floats into/from a string, refer to simxPackInts, simxPackFloats, simxUnpackInts and simxUnpackFloats. See also simxReadStringStream.
Matlab synopsis	[number returnCode]=simxWriteStringStream(number clientID,string signalName,string signalValueToAppend,number operationMode)
Matlab parameters	**clientID**: the client ID. refer to simxStart. **signalName**: name of the signal **signalValueToAppend**: value to append to the signal. That value may contain any value, including embedded zeros. **operationMode**: a remote API function operation mode. Recommended operation mode for this function is simx_opmode_oneshot
Matlab return values	**returnCode**: a remote API function return code
Other languages	C/C++, Python, Java, Octave, Lua