Introduction to Cognitive Robotics

Module 10: Using Turtlesim with CRAM
Lecture 3: Turtlesim with CRAM; implementing plans to move a turtle

www.cognitiverobotics.net

The CRAM Beginner Tutorials

Based on CRAM tutorials
http://cram-system.org/tutorials

Implementing simple plans to move a turtle

Based on Implementing simple plans to move a turtle
http://cram-system.org/tutorials/beginner/simple_plans

Implementing simple plans to move a turtle

Now let's learn how to write and implement a simple plan to move a turtle from waypoint to waypoint

We'll do this in three steps:

1. Design, implement, and test a function `calculate-angular-cmd` to compute the angle to the goal in the turtles frame of reference

   We will use this to re-orient the turtle towards the goal position

2. Test `calculate-angular-cmd` by moving the turtlebot to a goal position

3. Use `calculate-angular-cmd` to write a plan to move to a waypoint

Implementing simple plans to move a turtle

Step 1

1. Design, implement, and test a function `calculate-angular-cmd` to compute the angle to the goal in the turtles frame of reference

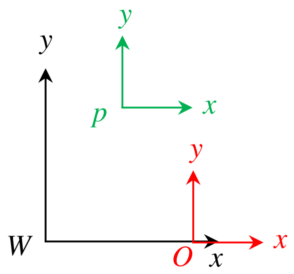
   How do we compute the angle to the goal in the turtles frame of reference?

Recall: Specifying Pose in ROS

How would we determine the pose of p w.r.t. O?

$$
\begin{aligned}
{}^{O}p &= {}^{O}W * {}^{W}p \\
&= ({}^{W}O)^{-1} * {}^{W}p \\
&= (\text{Trans}(2, 0, 0))^{-1} * \text{Trans}(1, 2, 0) \\
&= \text{Trans}(-2, 0, 0) * \text{Trans}(1, 2, 0) \\
&= \text{Trans}(-1, 2, 0)
\end{aligned}
$$

← w.r.t. O



Recall: Specifying Pose in ROS

Some more pose operations:

```
CL-USER > (transform
            (transform-inv (pose->transform O))
            p)
#<POSE
#<3D-VECTOR (-1.0d0 2.0d0 0.0d0)>
#<QUATERNION (0.0d0 0.0d0 0.0d0 1.0d0)>>
```

$({}^{W}O)^{-1}$

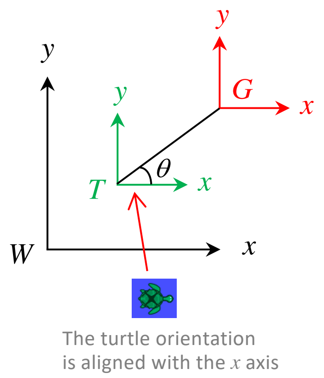
$({}^{W}O)^{-1} * {}^{W}p$

Here: Specifying Pose in ROS

We can use the same approach for determining the pose of the goal with respect to the turtle

$$^{T}G = {}^{T}W * {}^{W}G$$

$$= ({}^{W}T)^{-1} * {}^{W}G$$



Here: Specifying Pose in ROS

```
CL-USER > (transform
            (transform-inv (pose->transform pose-msg))
            goal)
#<POSE
#<3D-VECTOR (-3.0d0 4.0d0 0.0d0)>
#<QUATERNION (0.0d0 0.0d0 0.0d0 1.0d0)>>
```

$({}^{W}T)^{-1}$

$({}^{W}T)^{-1} * {}^{W}G$

We will implement this in `calculate-angular-cmd`

This gives us the coordinates of the goal with respect to the turtles frame of reference

The direction to the goal is given by atan2(y, x)

Implementing simple plans to move a turtle

As before, when developing new code, we need to

- (Update the dependencies in package.xml) ← We don't need to do this as there are no new packages being used
- Update the dependencies in cram-my-beginner-tutorial.asd ← We need to do this because we are going to put the new code in a separate file
- (Update the dependencies in package.lisp) ← We don't need to do this as there are no new packages being used
- Add the new code to simple-plans.lisp ← We will place the new code is a separate Lisp file
- Test the code
  - Run the ROS master
  - Run the Lisp REPL, loading the new program, creating a ROS node
  - Run turtlesim
  - Run turtlesim_teleop
  - Call the new functions

Implementing simple plans to move a turtle

Update the ASDF dependencies

Make sure you are in the cram_my_beginner_tutorial sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial
~/workspace/ros/src/cram_my_beginner_tutorial$
```

You should be there already
from the previous step

Implementing simple plans to move a turtle

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

```
~/workspace/ros/src/cram_my_beginner_tutorial$ emacs cram-my-beginner-tutorial.asd
```

Implementing simple plans to move a turtle

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

Add the `(:file "simple-plans" ...)` line below:

```
(defsystem cram-my-beginner-tutorial
  :depends-on (roslisp cram-language
               turtlesim-msg turtlesim-srv
               cl-transforms geometry_msgs-msg)
  :components
  ((:module "src"
            :components
            ((:file "package")
             (:file "control-turtlesim" :depends-on ("package"))
             (:file "simple-plans" :depends-on ("package" "control-turtlesim"))))))
```

The file should now look like this

Add this line

Be careful to ensure the open and closing brackets match

Implementing simple plans to move a turtle

Create a new Lisp file for the plan code

Make sure you are in the cram_my_beginner_tutorial/src sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial/src
~/workspace/ros/src/cram_my_beginner_tutorial/src$
```

Implementing simple plans to move a turtle

Create a new Lisp file for the plan code

Edit simple-plans.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs simple-plans.lisp
```

Implementing simple plans to move a turtle

Update the Lisp package to include the code for the simple plan

Edit simple-plans.lisp

Add the code on the next slide …

```
(in-package :tut)

(defun pose-msg->transform (msg)
  "Returns a transform proxy that allows to transform into the frame given by x, y, and theta of `msg'."

  (with-fields (x y theta) msg
    (cl-transforms:make-transform
     (cl-transforms:make-3d-vector x y 0)
     (cl-transforms:axis-angle->quaternion
      (cl-transforms:make-3d-vector 0 0 1)
      theta))))

(defun relative-angle-to (goal pose-msg)
  "Given a `pose-msg' as a turtlesim-msg:pose and a `goal' as cl-transforms:3d-vector,
   calculate the angle by which the pose has to be turned to point toward the goal."

  (let ((diff-pose (cl-transforms:transform-point
                     (cl-transforms:transform-inv
                       (pose-msg->transform pose-msg))
                     goal)))
    (atan
      (cl-transforms:y diff-pose)
      (cl-transforms:x diff-pose))))

(defun calculate-angular-cmd (goal &optional (ang-vel-factor 8))
  "Uses the current turtle pose and calculates the angular velocity command to turn towards the goal."

  (* ang-vel-factor
     (relative-angle-to goal (value *turtle-pose*))))
```

```lisp
(in-package :tut)

(defun pose-msg->transform (msg)
  "Returns a transform given by the x, y, and theta fields of message 'msg'"

  (with-fields (x y theta) msg
    (cl-transforms:make-transform
     (cl-transforms:make-3d-vector x y 0)
     (cl-transforms:axis-angle->quaternion
      (cl-transforms:make-3d-vector 0 0 1)
      theta))))

(defun relative-angle-to (goal pose-msg)
  "Given a `pose-msg' as a turtlesim-msg:pose and a `goal' as cl-transforms:3d-vector,
   calculate the angle by which the pose has to be turned to point toward the goal."

  (let ((diff-pose (cl-transforms:transform-point
                    (cl-transforms:transform-inv
                      (pose-msg->transform pose-msg))
                    goal)))
    (atan
      (cl-transforms:y diff-pose)
      (cl-transforms:x diff-pose))))

(defun calculate-angular-cmd (goal &optional (ang-vel-factor 8))
  "Uses the current turtle pose and calculates the angular velocity command to turn towards the goal."

  (* ang-vel-factor
     (relative-angle-to goal (value *turtle-pose*))))
```

Make a transform from a message

Qualify the fields so that we don't have to prefix with msg

Make a transform from a message
(could also have used `make-pose` ... check this)

Euler axis is aligned with the *Z* axis

```lisp
(in-package :tut)

(defun pose-msg->transform (msg)
  "Returns a transform given by the x, y, and theta fields of message 'msg'"

  (with-fields (x y theta) msg
    (cl-transforms:make-transform
     (cl-transforms:make-3d-vector x y 0)
     (cl-transforms:axis-angle->quaternion
      (cl-transforms:make-3d-vector 0 0 1)
      theta))))

(defun relative-angle-to (goal pose-msg)
  "Given a `pose-msg' as a turtlesim-msg:pose and a `goal' as cl-transforms:3d-vector,
   calculate the angle by which the pose has to be turned to point toward the goal."

  (let ((diff-pose (cl-transforms:transform-point
                    (cl-transforms:transform-inv
                      (pose-msg->transform pose-msg))
                    goal)))
    (atan
      (cl-transforms:y diff-pose)
      (cl-transforms:x diff-pose))))

(defun calculate-angular-cmd (goal &optional (ang-vel-factor 8))
  "Uses the current turtle pose and calculates the angular velocity command to turn towards the goal."

  (* ang-vel-factor
     (relative-angle-to goal (value *turtle-pose*))))
```

This function calculates the angle between the turtle orientation and the goal

This implements a version of the formula we derived in the previous slides for computing the coordinates of the goal in the turtle's frame for reference

This computes the angle from the coordinates
This angle represents the orientation error between the turtles current orientation and the desired heading to the goal

```
(in-package :tut)

(defun pose-msg->transform (msg)
  "Returns a transform given by the x, y, and theta fields of message 'msg'"

  (with-fields (x y theta) msg
    (cl-transforms:make-transform
     (cl-transforms:make-3d-vector x y 0)
     (cl-transforms:axis-angle->quaternion
      (cl-transforms:make-3d-vector 0 0 1)
      theta))))

(defun relative-angle-to (goal pose-msg)
  "Given a `pose-msg' as a turtlesim-msg:pose and a `goal' as cl-transforms:3d-vector,
   calculate the angle by which the pose has to be turned to point toward the goal."

  (let ((diff-pose (cl-transforms:transform-point
                     (cl-transforms:transform-inv
                       (pose-msg->transform pose-msg))
                     goal)))
    (atan
      (cl-transforms:y diff-pose)
      (cl-transforms:x diff-pose))))

(defun calculate-angular-cmd (goal &optional (ang-vel-factor 8))
  "Uses the current turtle pose and calculates the angular velocity command to turn towards the goal."

  (* ang-vel-factor
     (relative-angle-to goal (value *turtle-pose*))))
```

Let's look at what `diff-pose` is doing more closely ...

3. Compute a 3d-vector by applying the transform to the goal 3d-vector

2. Compute the inverse of the transform

1. Make pose-msg a transform

3d-vector

```lisp
(in-package :tut)

(defun pose-msg->transform (msg)
  "Returns a transform given by the x, y, and theta fields of message 'msg'"

  (with-fields (x y theta) msg
    (cl-transforms:make-transform
     (cl-transforms:make-3d-vector x y 0)
     (cl-transforms:axis-angle->quaternion
      (cl-transforms:make-3d-vector 0 0 1)
      theta))))

(defun relative-angle-to (goal pose-msg)
  "Given a `pose-msg' as a turtlesim-msg:pose and a `goal' as cl-transforms:3d-vector,
   calculate the angle by which the pose has to be turned to point toward the goal."

  (let ((diff-pose (cl-transforms:transform-point
                     (cl-transforms:transform-inv
                       (pose-msg->transform pose-msg))
                     goal)))
    (atan
      (cl-transforms:y diff-pose)
      (cl-transforms:x diff-pose))))

(defun calculate-angular-cmd (goal &optional (ang-vel-factor 8))
  "Uses the current turtle pose and calculates the angular velocity command to turn towards the goal."

  (* ang-vel-factor
     (relative-angle-to goal (value *turtle-pose*))))
```

This function calculates and returns the required angular velocity drive the orientation error to zero

It uses an optional gain parameter with a default value

Multiply the gain by the relative angle between the goal direction and the current turtle orientation

Implementing simple plans to move a turtle

Step 2

2. Test `calculate-angular-cmd` by moving the turtlebot to a goal position

Implementing simple plans to move a turtle

Before using these functions, we first need to recompile the code

There are several options to do this

- Type C-c C-c with the cursor on the function to recompile only the function
- Type C-c C-k to recompile the file
- Reload the complete ASDF system

```
CL-USER> (ros-load:load-system "cram_my_beginner_tutorial" :cram-my-beginner-tutorial)
CL-USER> (in-package :tut)
```

Implementing simple plans to move a turtle

Run the following to call `send-vel-cmd` 100 times

```
TUT> (dotimes (i 100)
          (send-vel-cmd
            1.5 ; linear speed
            (calculate-angular-cmd (cl-transforms:make-3d-vector 1 1 0)))
          (wait-duration 0.1))
```

The turtle should now move towards the bottom left corner and finally rotate around the goal until the loop finishes

Why does the turtle continue to rotate? Because the goal position and the turtle position are exactly the same, the turtle translates by a small amount, recalculates the orientation error, and rotates accordingly

Implementing simple plans to move a turtle

Step 3:

3. Use `calculate-angular-cmd` to write a plan to move to a waypoint

We will write a simple plan that recalculates & executes the velocity command until we reach the goal

Later, as an exercise, we'll implement both the divide-and-conquer and MIMO algorithms we met earlier in the course

Implementing simple plans to move a turtle

Edit the simple-plans.lisp file

Make sure you are in the cram_my_beginner_tutorial/src sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial/src
~/workspace/ros/src/cram_my_beginner_tutorial/src$
```

Implementing simple plans to move a turtle

Edit the simple-plans.lisp file

~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs simple-plans.lisp

Add the code on the next slide …

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

We use `def-cram-function` because we're going to use CRAM language features, specifically `pursue`

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

The distance threshold allows the program to end even if the robot position is not exactly equal to the goal position

```lisp
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

This fluent network is complicated.
Let's walk through it ...

```lisp
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                   (fl-funcall
                                    #'cl-transforms:translation
                                     (fl-funcall
                                      #'pose-msg->transform
                                      *turtle-pose*))
                                   goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

Make a transform from the pose message ...
the value of which depends on the fluent

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
              1.5
              (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

Access the translation slot

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                   (fl-funcall
                                    #'cl-transforms:translation
                                    (fl-funcall
                                     #'pose-msg->transform
                                     *turtle-pose*))
                                   goal)
                       distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

← Compute the Euclidean distance of the translation

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

The reached-fl fluent returns T if the Euclidean distance is less than the threshold

```lisp
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

The pursue form terminates whenever one of the two forms in the body terminates

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                   (fl-funcall
                                    #'cl-transforms:translation
                                    (fl-funcall
                                     #'pose-msg->transform
                                     *turtle-pose*))
                                   goal)
                       distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

Wait until the turtle arrives at the goal

```lisp
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                  (fl-funcall
                                   #'cl-transforms:translation
                                   (fl-funcall
                                    #'pose-msg->transform
                                    *turtle-pose*))
                                  goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

Send a velocity command
... and wait while that's executed

```
(def-cram-function move-to (goal &optional (distance-threshold 0.1))
  "Sends velocity commands until `goal' is reached."

 (let ((reached-fl (< (fl-funcall #'cl-transforms:v-dist
                                   (fl-funcall
                                    #'cl-transforms:translation
                                    (fl-funcall
                                     #'pose-msg->transform
                                     *turtle-pose*))
                                   goal)
                      distance-threshold)))
    (unwind-protect
         (pursue
           (wait-for reached-fl)
           (loop do
             (send-vel-cmd
               1.5
               (calculate-angular-cmd goal))
             (wait-duration 0.1)))
      (send-vel-cmd 0 0))))
```

Send a velocity command to stop the turtle

Implementing simple plans to move a turtle

Again, we first need to recompile the code

There are several options to do this

- Type C-c C-c with the cursor on the function to recompile only the function
- Type C-c C-k to recompile the file
- Reload the complete ASDF system

```
CL-USER> (ros-load:load-system "cram_my_beginner_tutorial" :cram-my-beginner-tutorial)
CL-USER> (in-package :tut)
```

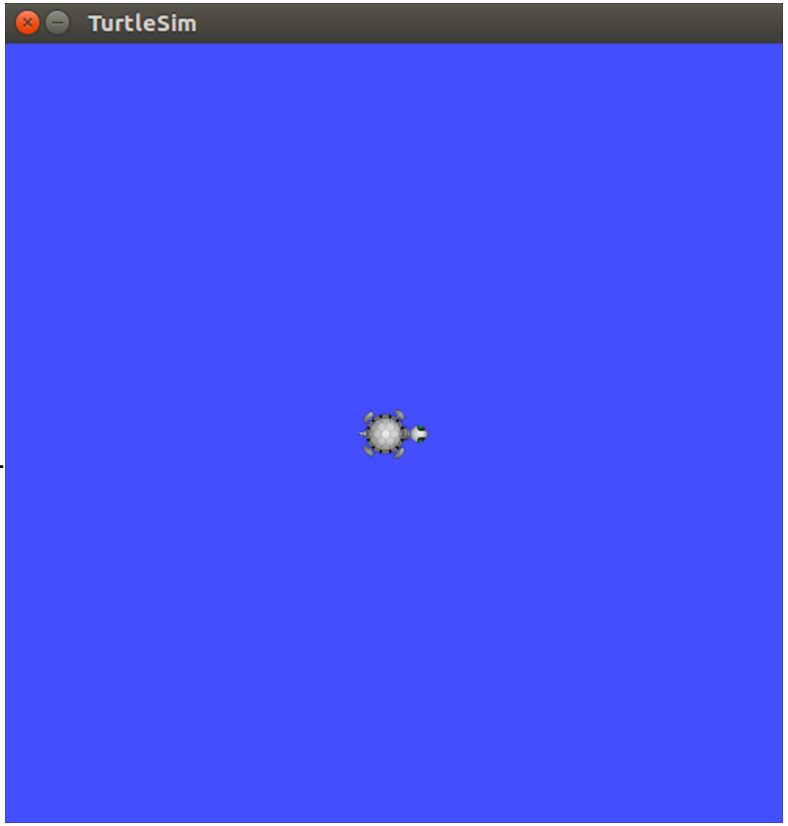
Implementing simple plans to move a turtle

Clear the Turtlesim environment

The simplest way is just to kill the process in the terminal

in which it was started and restart

~$ rosrun turtlesim turtlesim_node
[INFO] [1586708039.479694452]: Starting turtlesim with node name /turtlesim
[INFO] [1586708039.489055920]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544
^C ← Kill using <cntrl>-C
~$ rosrun turtlesim turtlesim_node



Implementing simple plans to move a turtle

Better:

- If the turtlesim environment gets a bit messy,
  you can clear the background by entering the following from a terminal

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ rosservice call /clear
```

- Or you can reset it completely by entering the following from a terminal
  (this creates a new turtle in the default pose)

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ rosservice call /reset
```

Implementing simple plans to move a turtle

Best:

You might even create a new function in control-turtlesim.lisp to reset (you might do the same for clear)
Here's the code:

```
(defvar *reset-srv* nil "name of ROS service for resetting the simulator")
...

;; add this to (defun init-ros-turtle (name) ...)
(setf *reset-srv* (concatenate 'string "reset"))
...

(defun call-reset ()
  "Function to call the reset service."
  (call-service *reset-srv* 'std_srvs-srv:empty))
```

Implementing simple plans to move a turtle

Best:

You might even create a new function in control-turtlesim.lisp to reset (you might do the same for clear)

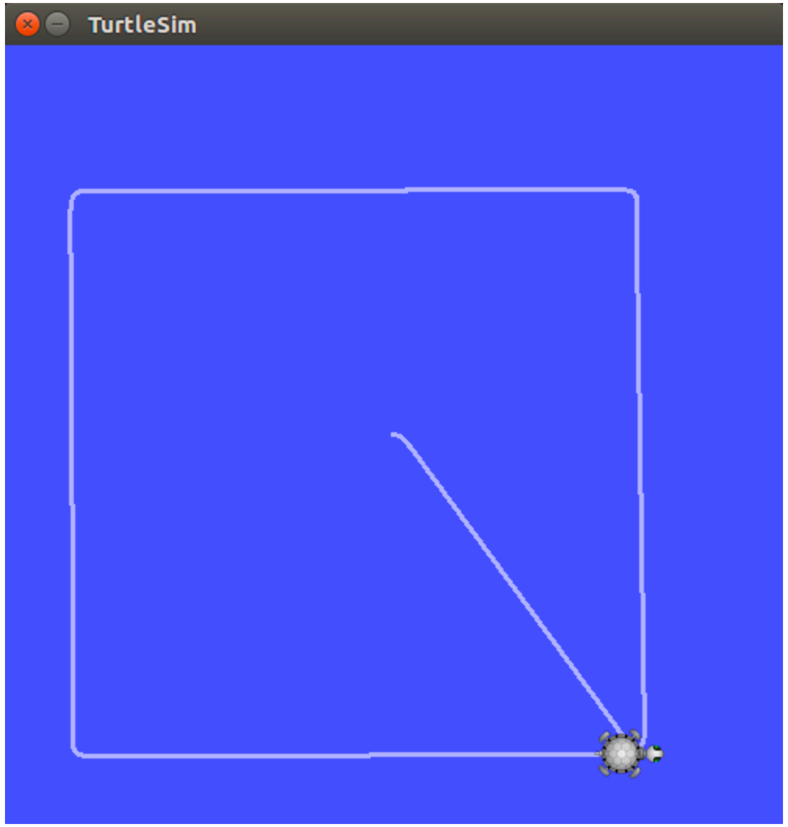
Now, to reset Turtlesim:

```
TUT> (call-reset)
```

Implementing simple plans to move a turtle

Run the following

```
TUT> (top-level
          (dolist (goal '((9 1 0) (9 9 0) (1 9 0) (1 1 0) (9 1 0)))
             (move-to (apply #'cl-transforms:make-3d-vector goal))))
```



The turtle should now move to these four waypoints, as shown

Implementing simple plans to move a turtle

Run the following

To execute CRAM Plan Language code (e.g. pursue or def-cram-function) we need to either call it from a function that was defined with def-top-level-cram-function or we need to wrap it in a top-level form

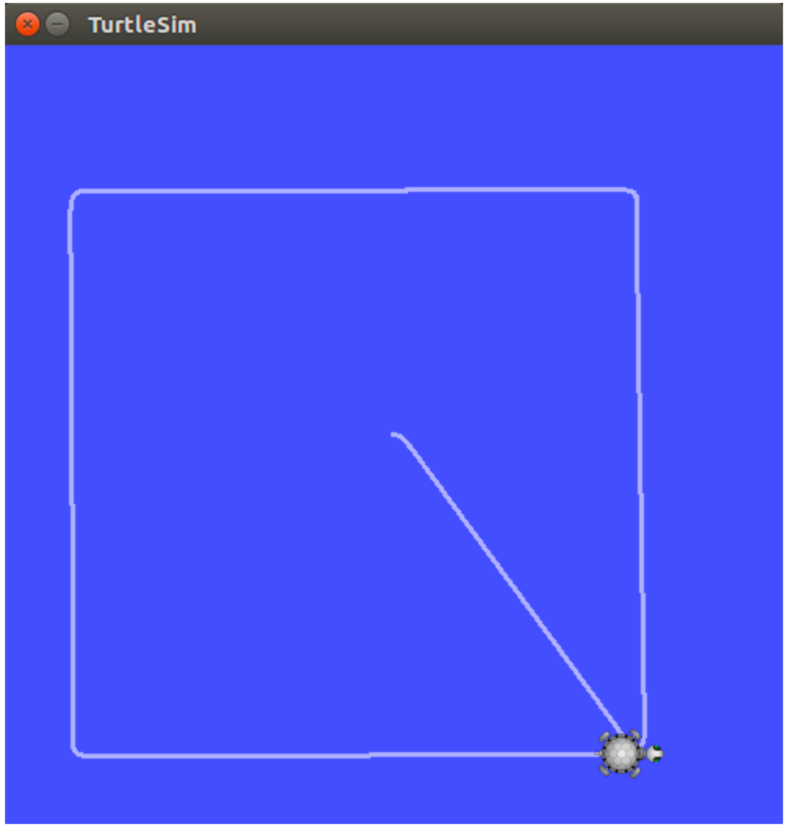
Quoted list of four waypoint coordinates

```
TUT> (top-level
          (dolist (goal '((9 1 0) (9 9 0) (1 9 0) (1 1 0) (9 1 0)))
             (move-to (apply #'cl-transforms:make-3d-vector goal))))
```

Recall: the dolist macro iterates through the elements of a list

Apply the sharp-quoted function to create an argument of the appropriate type (a 3D vector) from each goal list in the dolist

The turtle should now move to these four waypoints, as shown



Implementing simple plans to move a turtle

Experiment with this by changing waypoint coordinates

- You don't have to re-type the entire form each time
- You can copy and paste the text from the previous slides, edit it, and run it
- Or you can get REPL to replicate previously entered text:
  - Positioning the cursor over the text you want
  - Press enter to have it copied to the current prompt
  - Edit the text
  - press enter to run it

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs package.lisp
```

CRAM Beginner Tutorials

Create a CRAM Package	http://cram-system.org/tutorials/beginner/package_for_turtlesim
Controlling turtlesim from CRAM	http://cram-system.org/tutorials/beginner/controlling_turtlesim_2
Implementing simple plans to move a turtle	http://cram-system.org/tutorials/beginner/simple_plans

Background Reading

G. Kazhoyan, Lecture notes: Robot Programming with Lisp 7. Coordinate Transformations, TF, ActionLib, slides 5-8.
`https://ai.uni-bremen.de/_media/teaching/7_more_ros.pdf`

`http://wiki.ros.org/tf/Overview/Transformations`

T. Rittweiler, CRAM – Design and Implementation of a Reactive Plan Language, Bachelor Thesis, Technical University of Munich, 2010.
`https://common-lisp.net/~trittweiler/bachelor-thesis.pdf`