

Wheel_Locomotor Kinematics and Implementation

Overview

Vehicle models

- Fully steered
- Partially steered
- Skid steered

Algorithms

- Position driving with crab and arc motions: 1-arc, 2-arc motions
- Velocity driving

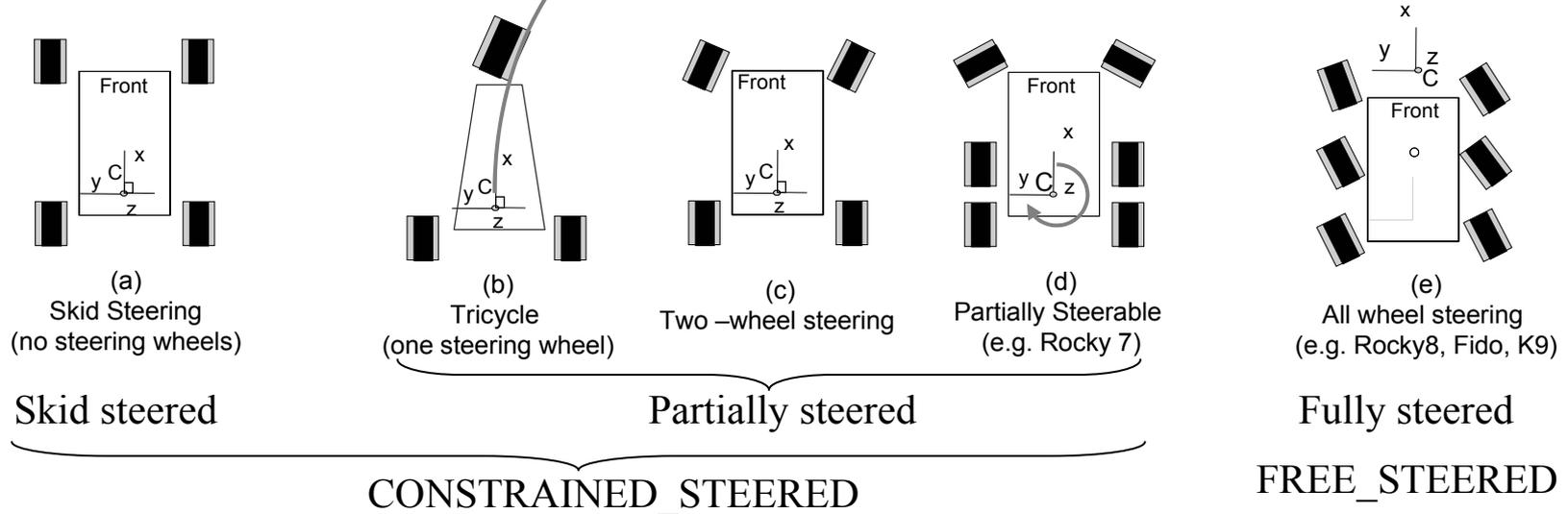
Implementation

- UML Overview
- Software Scenarios

Position driving: background information

1. I. Nesnas, M. Maimone, H. Das, "Rover Maneuvering for Autonomous Vision-Based Dexterous Manipulation", Proc. IEEE International Conference on Robotics and Automation", San Francisco, CA. 2000.
2. B. Shamah, "Experimental Comparison of Skid Steering Vs. Explicit Steering for a Wheeled Mobile Robot", Master's thesis, Tech. Report CMU-RI-TR-99-06, Robotics Institute, Carnegie Mellon University, March, 1999.
3. R. Volpe, "Inverse Kinematics for All-wheel Steered Vehicles", JPL internal report.

Vehicle models



Model Restrictions/Assumptions

1. All wheels are independently driven
2. All wheels are mounted to (real or virtual) axles
3. All axles are rigidly attached and perpendicular to the vehicle body heading axis
4. All axles lie in the same plane, all wheels have the same wheel diameter.
5. Axles may have any number of wheels; different axles can have different numbers of wheels
6. Wheels do not have to be symmetrically mounted on an axle
7. All wheels on an axle must be either steered or non-steered.
8. Steered wheels are independently steered and rotate about a vertical axis through their ground contact point and through a point fixed to the respective axle.
9. Model wheel-ground interaction as a thin rigid disk with point contact rotating without slipping in the rolling direction on a perfectly hard and flat surface.

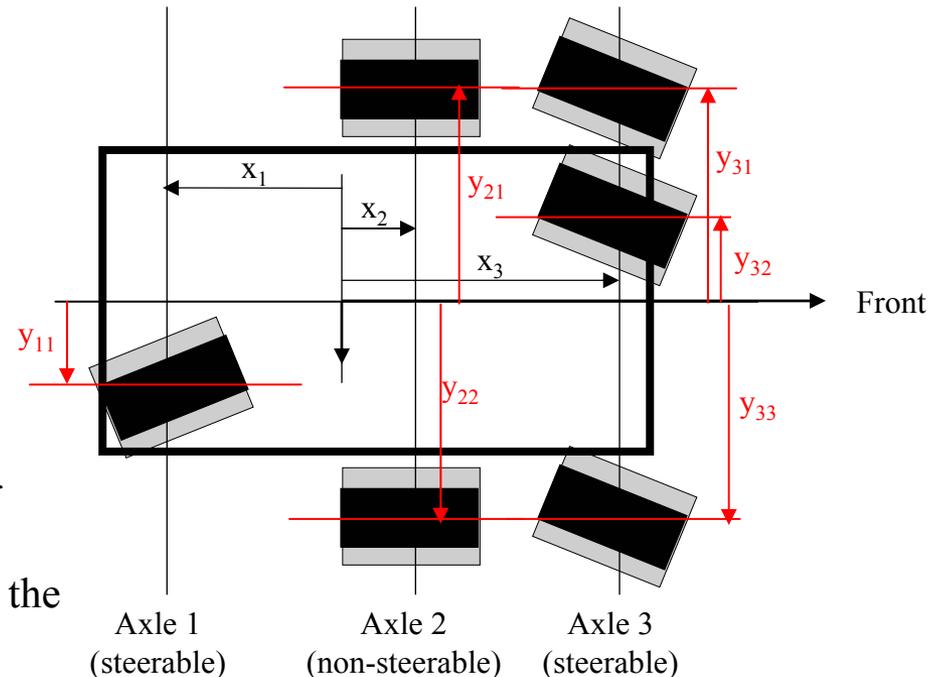
Vehicle kinematic parameters

Parameters to specify vehicle kinematics

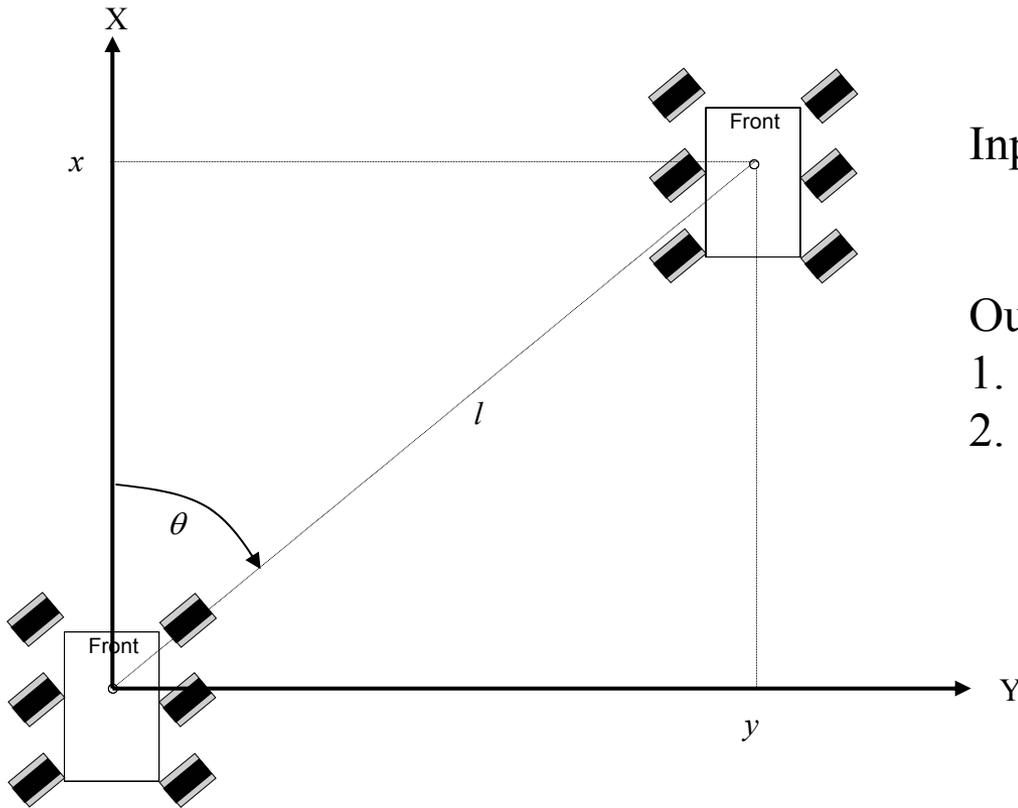
1. Number of axles.
2. Which axles have steerable wheels
3. Location of the intersections of the axles and the x-axis of the vehicle-based coordinate frame
4. Number of wheels on each axle.
5. Locations of the wheels on each axle

Given the above data:

1. the vehicle type (skid, partially steered or fully steered) can be determined
2. The locations (x and y coordinates) of all the wheels can be determined with respect to the vehicle coordinate frame



Position driving: fully steered crab driving



Input:

x, y

Output motion sequence:

1. Steer wheels to appropriate angles
2. Drive wheels the appropriate distance while maintaining steer angles

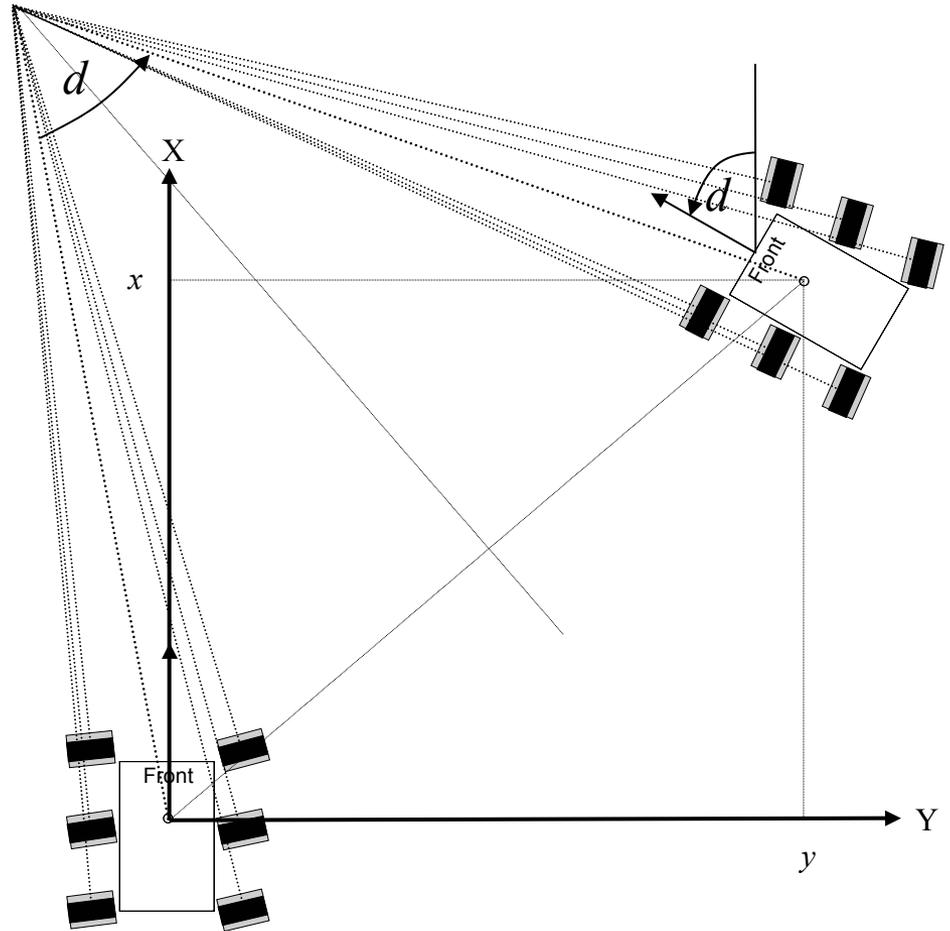
Position driving: fully steered arc driving

Input:

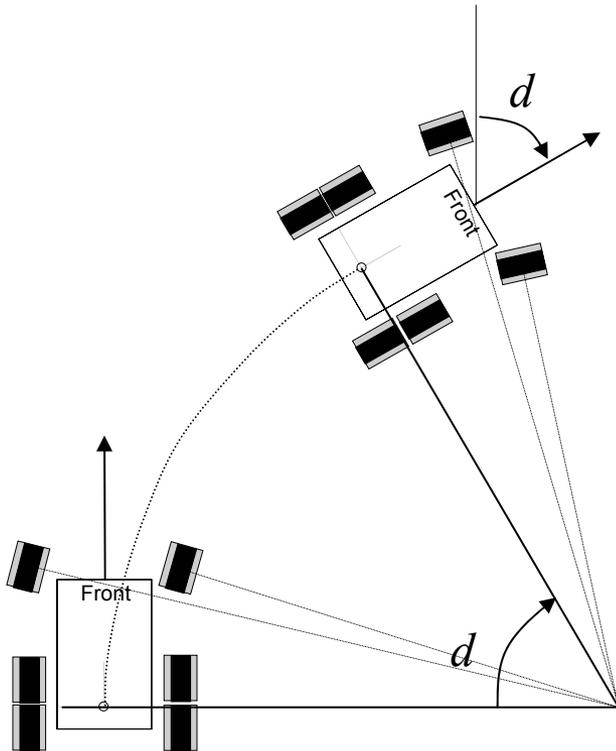
x, y, d

Output motion sequence:

1. Steer wheels to appropriate angles
2. Drive wheels the appropriate distance while maintaining steer angles



Position driving: partially steered 1-arc driving



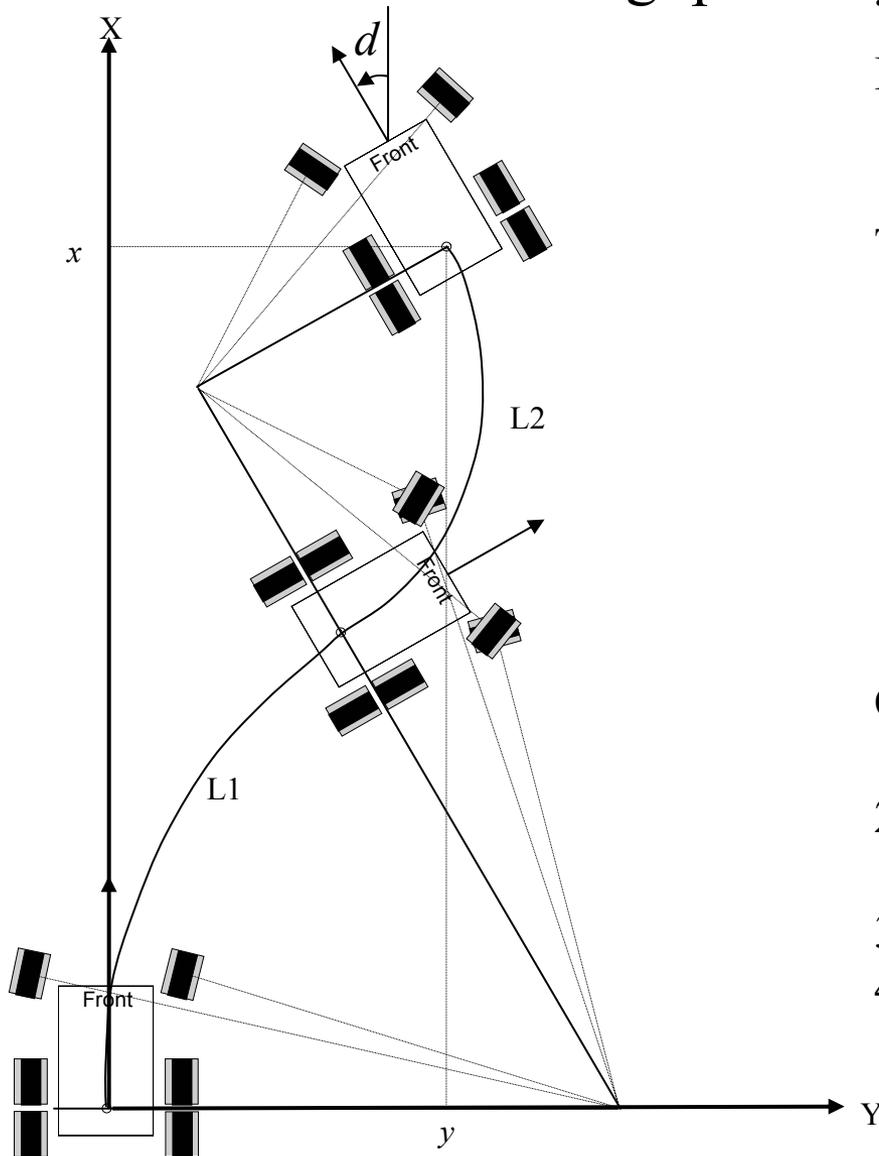
Input:

l, d

Output motion sequence:

1. Steer wheels to appropriate angles
2. Drive wheels the appropriate distance while maintaining steer angles

Position driving: partially steered 2-arc driving



Input:

x, y, d

There are an infinite number of 2-arcs sets that will get the partially steered vehicle to the desired position and heading. Define a cost function to pick an “optimal” 2-arc set. Cost function =

$$L1 + L2 + \text{abs}(L1-L2)$$

Output motion sequence:

1. Steer wheels to 1st set of steer angles
2. Drive wheels the appropriate distance while maintaining steer angles
3. Steer wheels to 2nd set of steer angles
4. Drive wheels the appropriate distance while maintaining steer angles

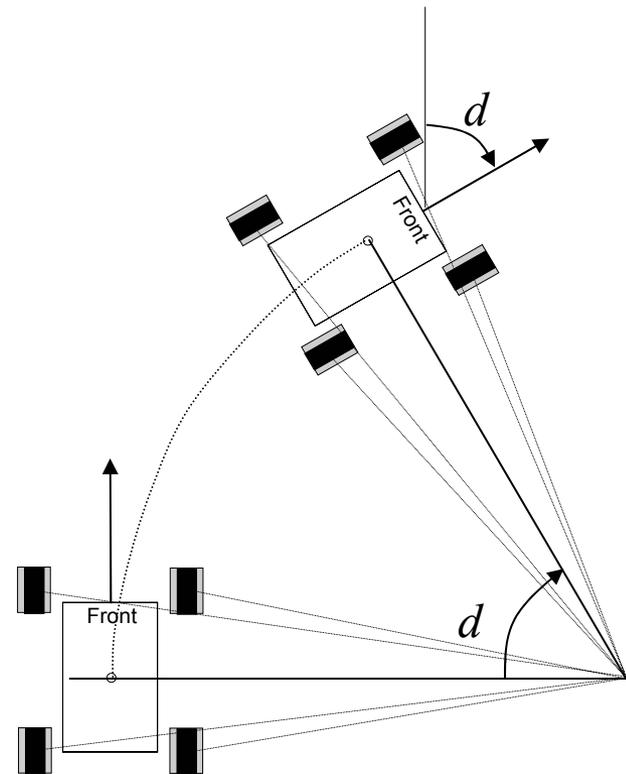
Position driving: skid steered arc driving

Input:

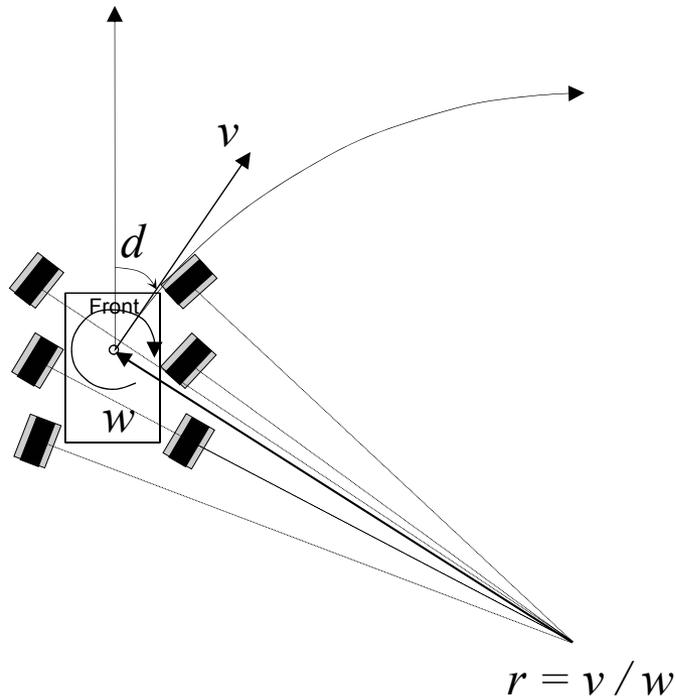
l, d

Output motion sequence:

1. Drive wheels the appropriate distance



Velocity driving for fully steered vehicles

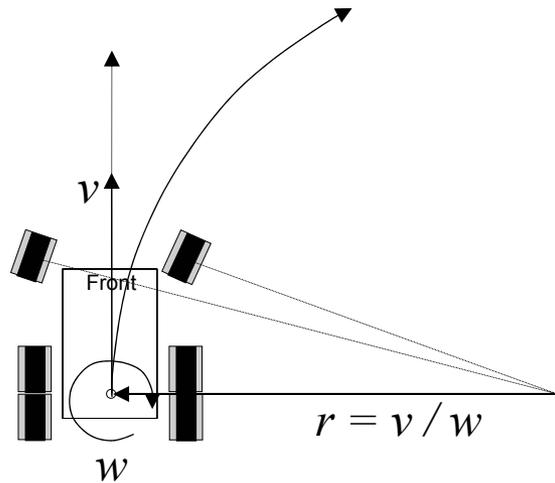


Input (with respect to the vehicle frame):
 v, d, w

Output motion sequence:

1. If error between desired steer angles and current steer angles is greater than some programmable threshold, steer wheels to appropriate angles
2. Steer wheels to the desired angles and drive wheels at the appropriate velocity

Velocity driving for partially and skid steered vehicles

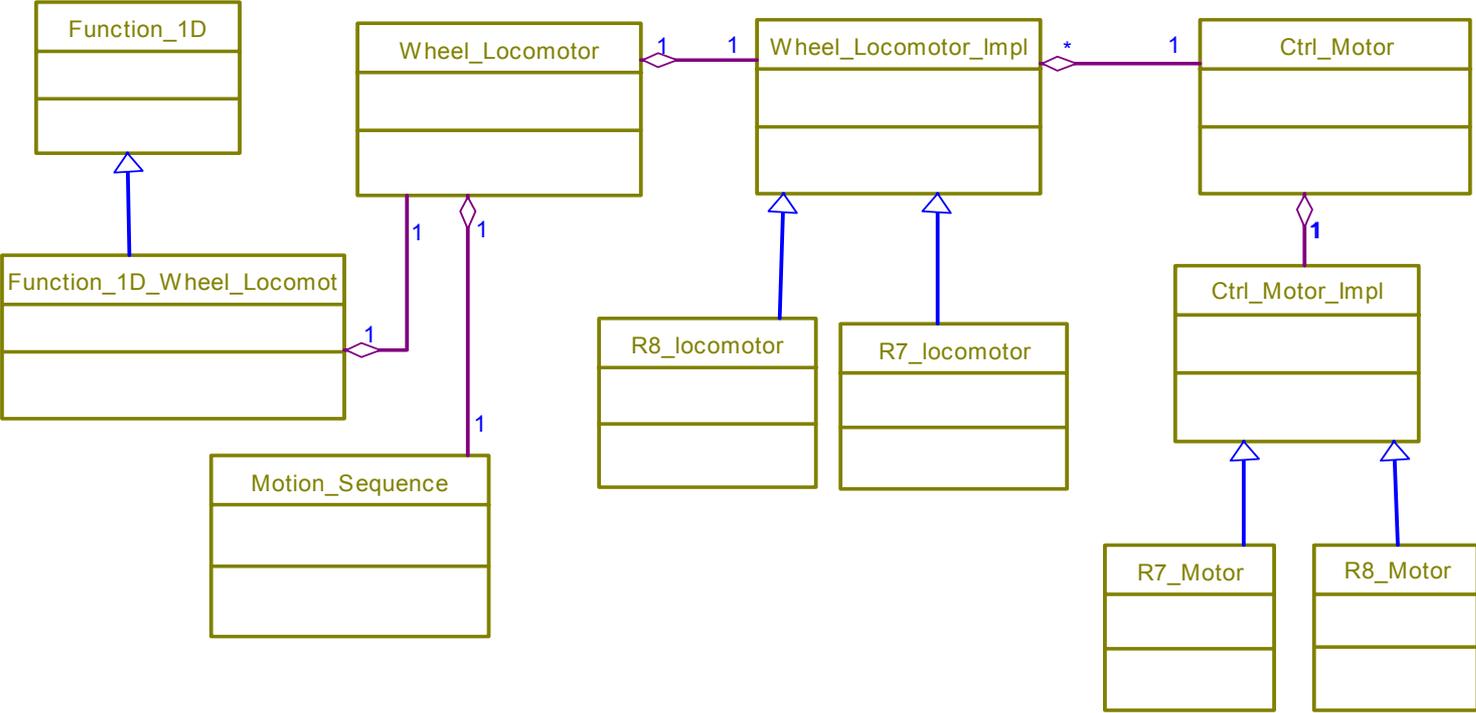


Input (with respect to the vehicle frame):
 v, w

Output motion sequence:

1. If error between desired steer angles and current steer angles is greater than some programmable threshold, steer wheels to appropriate angles
2. Steer wheels to the desired angles and drive wheels at the appropriate velocity

UML Overview – Wheel_Locomotor and associated classes



Software Scenarios

The next few examples illustrate how different motions are made.

The examples have simplified the code to make it easier to present. Intermediate function calls and computations are not covered in detail.

Software Scenarios – **move(length, 0.0)**

```
void move(length, 0.0)
{
    // If vehicle is moving, stop it
    // If a motion sequence exists, de-allocate it

    // Generate a new motion sequence using the
    // _straight_line_to_steer_drive_angles(double path_length) function

    // Drive along the motion sequence using the
    // position_drive() function
}

void _straight_line_to_steer_drive_angles(double path_length)
{
    // This is a 2-segment move.
    // Segment 1: Steer all steer wheels to zero angles (STEERING_IN_PLACE move_mode)
    // Segment 2: Control drive motors to the desired distance (DRIVING_STRAIGHT_LINE move mode)

    // Set the move mode (used by the local and external pose estimation algorithm)
    // Save the geometry of the motion (for use by the local pose estimation algorithm)

    // Allocate the motion sequence and populate it (steer angles and drive distances)
}
```

Software Scenarios – move(a, b)

```
void move(const Dpoint & a, const Dpoint & b)
{
    // If vehicle is moving, stop it
    // If a motion sequence exists, de-allocate it

    // If FREE_STEERED perform crab motion to move from a to b with no heading change
    // Compute the corresponding path_length and crab angle to drive at
    // Generate a new motion sequence using the
    // _crab_line_to_steer_drive_angles(double path_length, double angle) function

    // If CONSTRAINED_STEERED perform arc move to from a to b with heading change
    // Compute the corresponding radius_x (= ref_axel_position), radius_y
    // and heading to drive at
    // Generate a new motion sequence using the
    // _arc_circle_to_steer_drive_angles(radius_x ,
    // double radius_y, double heading) function

    // Drive along the motion sequence using the
    // position_drive() function
}

void _crab_line_to_steer_drive_angles(double path_length , double angle)
{
    // This is a 2-segment move.
    // Segment 1: Steer all steer wheels to computed steer angles (STEERING_IN_PLACE move_mode)
    // Segment 2: Control drive motors to the desired distance (DRIVING_CRAB_LINE move mode)

    // Set the move mode (used by the local and external pose estimation algorithm)
    // Save the geometry of the motion (for use by the local pose estimation algorithm)

    // Allocate the motion sequence and populate it (steer angles and drive distances)
}
```

Software Scenarios – **move(a, b)** – cont.

```
void _arc_circle_to_steer_drive_angles(double radius_x, double radius_y, double heading)
{
    // This is a 2-segment move.
    // Segment 1: Steer all steer wheels to computed steer angles (STEERING_IN_PLACE move_mode)
    // Segment 2: Control drive motors to the desired distance (DRIVING_CRAB_LINE move mode)

    // Set the move mode (used by the local and external pose estimation algorithm)
    // Save the geometry of the motion (for use by the local pose estimation algorithm)

    // Compute the corresponding steer angles and drive distances for each wheel.

    // Allocate the motion sequence and populate it (steer angles and drive distances)
}
```

Software Scenarios – move(a, b, heading)

```
void move(const Dpoint & a, const Dpoint & b, const & heading)
{
    // If vehicle is moving, stop it
    // If a motion sequence exists, de-allocate it

    // If FREE_STEERED, perform arc motion to move from a to b with heading change
    // Compute the corresponding radius_x, radius_y and heading to drive at
    // Generate a new motion sequence using the
    // _arc_circle_to_steering_drive_angles(double radius_x,
    // double radius_y, double heading) function

    // If CONSTRAINED_STEERED, perform 2-arc move to from a to b with heading change
    // First guess of 2nd path radius = distance to mid-point between a and b
    // Minimize cost function = fabs(L1) + abs(L2) + fabs(fabs(L1) - fabs(L2))
    // Compute the arc radii (radius_x, radius_y) and heading change for
    // each arc
    // Generate a new motion sequence for the first arc
    // path1 = _arc_circle_to_steering_drive_angles(double radius_x,
    // double radius_y, double heading) function

    // Generate a new motion sequence for the second arc
    // path2 = _arc_circle_to_steering_drive_angles(double radius_x,
    // double radius_y, double heading) function

    // Add the motion sequences path = path1 + path2

    // Drive along the motion sequence using the
    // position_drive() function
}
```

Software Scenarios – `position_drive()`

```
void position_drive()
{
    // Set semaphore to indicate that we are spawning a drive task
    // Fill in a global data structure containing a pointer to the Wheel_Locomotor
    // position_drive_loop function to be spawned and a pointer to
    // this Wheel_Locomotor

    // Spawn the globally defined start_thread() function with the global data struct
    // as its argument
    // start_thread() in turn calls the Wheel_Locomotor position_drive_loop function
}
void position_drive_loop()
{
    // If there is not path, reset the semaphores and return

    // for the number of segments in the motion sequence
        // wait until the current motor command is completed
        // update the pose (for crude pose estimation)
        // reset the home position for the drive motors
        // if stop was issued
            // break out of the for loop
        // set a counter to indicate the path segment being executed
        // set the motor motion parameters (position, velocity, accel)
        // start driving
    // end for
    // wait until the current motor command is completed
    // update the pose (for crude pose estimation)
    // reset the home position for the drive motors
    // Reset the semaphores
}
```

Software Scenarios – `drive_continuous(v, w)`

```
void drive_continuous(double v, double w)
{
    // If a previous velocity task is executing
    // wait until it is done

    // If w = 0.0
    // do straight line velocity drive with
    // _straight_line_to_steering_drive_velocities(v)
    // Else
    // do arc velocity drive with
    // _arc_line_to_steering_drive_velocities(v, 0.0, w)
}

void _straight_line_to_steering_drive_velocities(double v)
{
    // Allocate a 2-segment motion sequence and fill in the steer angles (=0) for
    // the first segment then wheel drive velocities (corresponding vehicle body
    // velocity) for the second segment

    // Drive at the velocity using the velocity_drive() function
}

void _arc_line_to_steering_drive_velocities(double v, double d, double w)
{
    // Allocate a 2-segment motion sequence
    // Compute the instantaneous arc parameters corresponding to the velocity command
    // Compute the corresponding wheel steer angles and drive velocities
    // Fill in the motion sequence with the computed wheel steer angles and drive
    // velocities

    // Drive at the velocity using the velocity_drive() function
}
```

Software Scenarios – **drive_continuous(v, d, w)**

```
void drive_continuous(double v, double d, double w)
{
// If a previous velocity task is executing
    // wait until it is done

    // If w > 0.0
        // do arc velocity drive with
        // _arc_line_to_steering_drive_velocities(v, d, w)

    // Else if d > 0.0
        // do straight line velocity drive with
        // _crab_line_to_steering_drive_velocities(v, d)
    // Else
        // do arc velocity drive with
        // _arc_line_to_steering_drive_velocities(v)
}

void _crab_line_to_steering_drive_velocities(double v, double d)
{
    // Allocate a 2-segment motion sequence and fill in the steer angles (=d) for
    // the first segment then wheel drive velocities (corresponding vehicle body
    // velocity) for the second segment

    // Drive at the velocity using the velocity_drive() function
}
```

Software Scenarios – **velocity_drive()**

```
void velocity_drive()
{
    // Wait until the previous velocity_drive_update() returns

    // Set semaphore to indicate that we are spawning a drive task

    // Reset the command_interval_timer for monitoring command update intervals

    // If a velocity_monitor task is not active
        // Set a semaphore to indicate that it is being activated
        // Fill in a global data structure containing a pointer to the
        // Wheel_Locomotor_velocity_monitor() function to be spawned and
        // a pointer to this Wheel_Locomotor
        // Spawn the _velocity_monitor() task

    // Fill in a global data structure containing a pointer to the
    // Wheel_Locomotor velocity_drive_update() function to be spawned and
    // a pointer to this Wheel_Locomotor

    // Spawn the globally defined start_thread() function with the global data struct
    // as its argument

    // start_thread() in turn calls the Wheel_Locomotor velocity_drive_update() function
}
```

Software Scenarios – **velocity_drive_update()**, **velocity_monitor()**

```
void velocity_drive_update()
```

```
{  
    // Set the steer motor params and drive the steer motors to their  
    // steer angles as specified in the motion sequence  
  
    // for all steer motor  
        // While the error between the actual steer angles and the  
        // commanded steer angles are larger than a threshold value  
            // If a stop motion was commanded  
                // Reset the semaphores and return  
            // sleep for 0.1 secs  
    // for all drive motors  
        // change velocity to the commanded velocity  
  
    // De-allocate the motion sequence  
    // Reset the semaphores  
}
```

```
void _velocity_monitor()
```

```
{  
    // Set a local counter to 0  
    // While command_interval_timer time is less than time-out interval (20 secs)  
        // Sleep for 0.1 secs  
        // Increment counter  
        // If counter == 10  
            // Print out time elapsed  
            // reset counter  
  
    // stop motion  
    // reset semaphores  
}
```