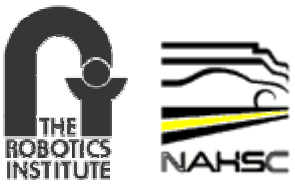

Sensor Models for AHS Simulations

Cem Ünsal

CMU-RI-TR-98-01

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania
15213-3980

February 1998

This work was supported by US Department of Transportation under Cooperative Agreement number DTFH61-94-X-00001 as part of the National Automated Highway System Consortium.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expresses or implied, of the U.S. government.

Table of Contents

1 Introduction

Sensor technology plays a critical role in the operation of the Automated Highway System (AHS). The proposed concepts depend on a variety of sensors for positioning, lane-tracking, range and vehicle proximity. Since large subsystems of the AHS will be designed and evaluated in simulation before deployment, it is important that simulators make realistic sensor assumptions.

The sensor models presented here are part of the functional sensor hierarchy, incorporating geometric models, abstract noise characteristics, and can be used directly with current AHS tools. These models capture the aspects of sensing technology that are important to AHS concept design such as occlusion, and field of view restrictions, while ignoring physical-level details such as electromagnetic sensor reflections. Since the functional sensor models operate at the same level of granularity as the simulation platform, complete integration is assured. The hierarchy classifies sensors into functional groups. The models at a particular level incorporate characteristics that are common to all sensors in its subgroups. For example, range sensors have a parameter corresponding to a maximum effective range, while lane-trackers will include information pertaining to lateral accuracy.

Sensor models described in this document are implemented using SHIFT programming language [1]. The characteristics and general input/output structure of the sensors are compatible with Smart-AHS [2] simulation platform defined in [3], and therefore can be used with Smart-AHS package (current version 1.1).

Each of the classes described in the following sections is represented in the microsimulator by a functional sensor model. The functional sensor model contains three important parts. First, it implements algorithms that access the microsimulator's internal state to create the appropriate outputs (e.g., a range sensor should return the distance to the appropriate target). Second, it is responsible for realistically corrupting the sensor measurements. Parameters such as maximum range or field of view are represented explicitly when the microsimulator's fidelity can support them. In other cases, the characteristics will be captured in a more abstract manner (e.g., expected accuracy of lane tracking). Initial results obtained with the following sensor models as well as the initial sensor hierarchy envisioned for AHS applications are presented in [4].

The computer implementation of a functional sensor is a module that processes information from the simulation environment (“real world”) to create inputs suitable for the cognition system (“perceived world”). Additionally, the functional sensors are used to “corrupt” the actual state of the world, as represented in the microsimulator internals, into realistic sensor readings, useful for evaluating control methodologies.

The general structure of a range sensor is given in Figure 1 to illustrate our sensor implementation approach. The environment conditions are used to evaluate noise characteristics fort a specific sensor, although the initial noise model is defined at the setup phase with other sensor characteristics such as the level of detail and pertaining parameters. The information about the vehicles is preprocessed in order to extract necessary information, such as current sensor and vehicle positions and orientations, and the set of vehicle in the field of view. The preprocessing is followed by the computation of the desired sensor outputs. Fault modes of the sensors are defined as discrete states, and the parameters characterizing the sensor capabilities are updated while transitioning between these states.

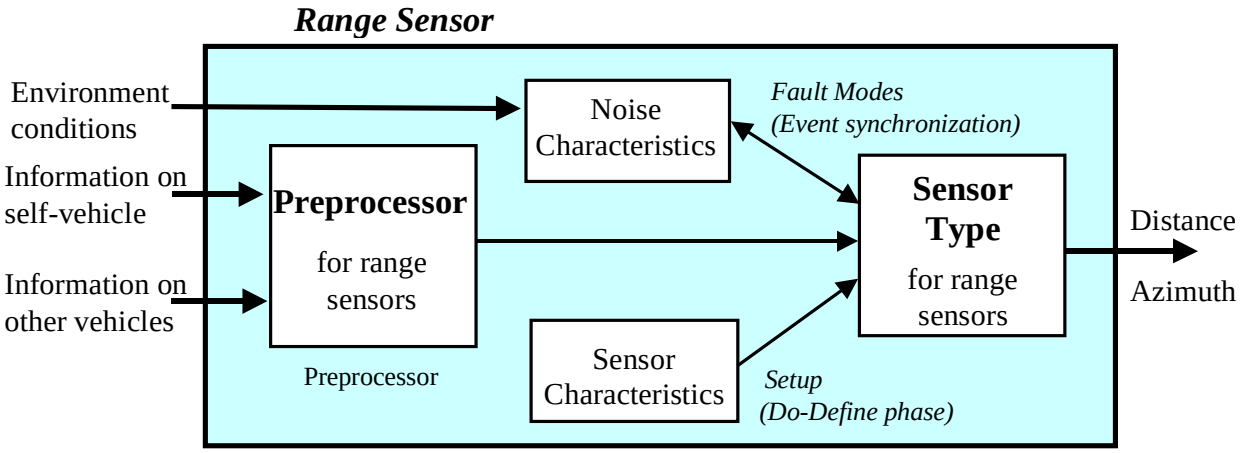

Figure 1. Implementation structure for a functional range sensor.

2 Range Sensors

There are currently four different implementations for on-board range sensors. These are described in the following sections. All sensors are implemented in two-dimensional space, although the simulation platform supports three-dimensional definitions. The extension of the models to 3-D will be carried out, as it becomes necessary.

Figure 2 illustrates the levels of detail considered for range sensor implementation. The simplest model for range sensing uses the position information on all vehicles to compute the distance to the closest vehicle (Figure 2a). This approach treats the vehicles as particles; therefore, the error in sensed distance can be large for shorter ranges. Furthermore, the exact position of the vehicle with respect to the sensor is not known.

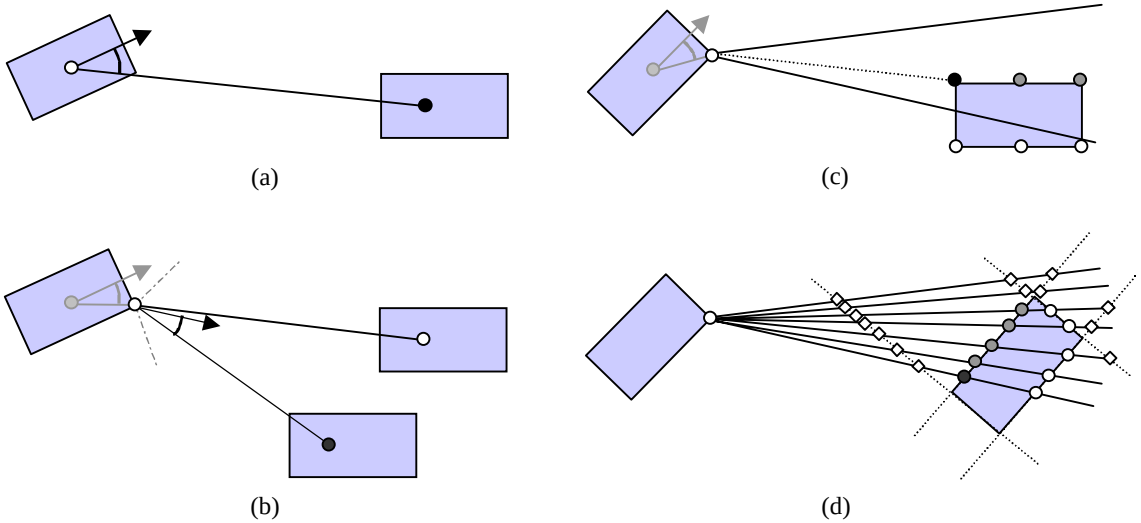

Figure 2. Four different levels of detail for generic range sensor: (a) point vehicles, (b) addition of sensor position and orientation, (c) pseudo-vertex calculations for vehicle orientation, (d) ray definitions.

A natural extension of the first sensor model is the addition of the sensor position and orientation in vehicle coordinate frame (Figure 2b). This enables the user to define more realistic sensors such as side-looking proximity sensors. The addition of the position and orientation information slightly increases the computational burden. The global position and orientation of the sensor must be computed at every time step using the vehicle information provided by simulation platform. For the

first two sensor models, it is possible to define a horizontal field of view by simply filtering the vehicles with azimuth readings beyond a predefined value. The definition of field of view is implemented for the second model.

The next two levels of detail for range sensor model are designed to provide additional information about the orientation of the sensed objects as well as more accurate information about the position and the range of the sensed vehicle. The model which employs "pseudo-vertex" definitions enables the user to define points on the rectangles describing the vehicles, and uses these points to evaluate the range and azimuth angle (Figure 2c). This approach is useful in long range sensor implementations or in platoon simulations where the road curvature is small; the returned azimuth angle may be drastically different from the actual value in close range.

The sensor module created for pseudo-vertices approach can return the range and azimuth angle of all defined points in the sensor range (with minor changes in the current external function), thus providing additional information about the orientation of the vehicle/obstacle surface. The price for this additional information is of course the complexity of computation: The length and width of all the vehicles, as well as their current orientation, must be fed to the sensor modules, since they are required for calculating the position of multiple pseudo-vertex points. An additional computation loop for all these points is needed for each vehicle in range (See external functions given in Section 13 for details).

The most complex range sensor definition includes ray definitions where the number of scanning rays are parameterized in addition to previous definitions of horizontal field of view, sensor position and orientation. In this model, the functional sensor computes the intersection of defined rays (for which the maximum range and the azimuth angle are known) and the lines defining the sensed vehicle. For each ray of the sensor, and for each line of all vehicles in range, the intersections are computed. Intersection points outside of the segments that constitute the vehicle ("diamonds" in Figure 2d) are discarded as well as those that are in the opposing side of the vehicle ("white circles" in Figure 2d). The closest intersection point is returned as the range.

All sensor models presented here use periodical updates where the sampling frequency is a user-defined parameter that can be as small as the simulation time step. Gaussian measurement noise can be injected into range and/or azimuth readings; the mean and variance of the noise are also user-defined parameters.

Furthermore, it is possible to define different operating modes with minor changes in the program code. For example, the noise characteristics of a range sensor may depend on the weather conditions on the road segment that the vehicle travels, or the user may want to introduce operating modes that emulate faults in the system. Examples of such sensors are given in later sections.

2.1 Simple Range Sensor

This functional sensor type uses the global set of vehicles to obtain the position of all vehicles in the global reference frame. The sensor is assumed to be located at the center of gravity of the vehicle (Figure 2a). The sensor detects the position of other vehicles in 2-D space. It returns the distance to the center of gravity of the sensed vehicle and the azimuth angle from vehicle x-axis that is directed toward the front of the vehicle. This evaluation does not take the shapes of vehicles into account.

As seen in Figure 3, range sensor's main axis is assumed to be aligned with the x-axis of the vehicle coordinate frame. The rotation of the two-dimensional vehicle coordinate frame with respect to global coordinate frame is shown as θ_{vg}. The orientation of the line connecting to vehicles with respect to the global coordinate frame is θ_{sv}. The first angle can be evaluated using the vehicle-to-global alignment matrix available from Smart-AHS implementation *for vehicle roadway environment processor* (*VREP*) in SHIFT (For detailed information, see [3], and Section 12). The second value is calculated using the *atan2* function with vehicle coordinates. The azimuth angle can then be calculated as the difference between these two values:

$$\theta_{vg} = \text{atan}\,2(vgam12, vgam11)$$

(Eq. 1) $$\theta_{sv} = \text{atan}\,2(gyp(av) - gyp, gxp(av) - gxp)$$

$$\alpha = -(\theta_{vg} - \theta_{sv})$$

where *vgam11* and *vgam12* are the first and second elements of the "vehicle-to-global alignment matrix" *VGAM*, and gxp and gyp indicates the global positions of the vehicles. The negative sign in (Eq. 1) is required due to the right-handed definition of the vehicle coordinate frame while the global frame is right-handed.

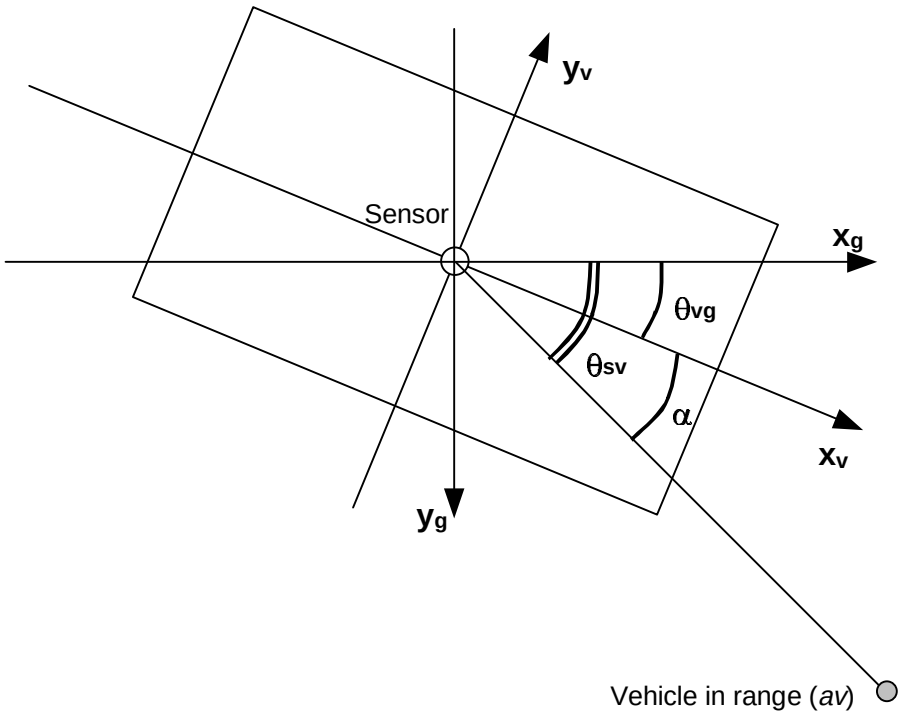

Figure 3. Alignment of sensor main axis with global and vehicle coordinate frames.

2.1.1 I/O Structure and Parameters

The type *RangeSensor* has the following inputs:

- *gxp, gyp*: Global position of the vehicle/sensor (m)
- *vgam11, vgam12*: First to elements of the vehicle orientation matrix *VGAM*.

The following values are returned as outputs:

- *distance*: Distance from the center of gravity of the sensed vehicle to the sensor. (m)
- *angle*: Azimuth angle defined between the sensor main axis (i.e., vehicle x-axis) and the line connecting the sensor and the COG of the sensed vehicle (deg).
- *closest*: Identification number of the closest vehicle in sensor range.

The following are user-defined parameters or state variables (evaluated during processing) for the range sensor type:

- *vehicle*: Vehicle associated with the sensor.
- *maxrange*: Maximum range (m).
- *procspd*: Processing speed for the sensor (sec).
- *inrange*: Set of the vehicles in range.

- *t*: Time (sec).

The set of vehicles is defined globally as shown in Section 11.

2.1.2 Source Code

The SHIFT code for the simplest range sensor is given below:

<rangesensor.hs>

The function that evaluates the distance to the vehicle in range is implemented as an external C subroutine, and is linked as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

2.2 Range Sensor Type 2

This type is similar to the previous one except the fact that the location and the orientation of the sensor with respect to the vehicle coordinate frame can be defined. Again, all vehicle positions in global vehicle frame are obtained from the global set of vehicles.

The sensor detects the positions of other vehicles in 2-D space. It evaluates the distance from the sensor location to and the azimuth angle from the main sensor axis for the closest vehicle in range. The shape of the vehicle is not taken into account. A set of vehicles that are in the sensing range and in the field of view is generated before selecting the closest vehicle.

As seen in Figure 4, the sensor's main axis deviates from the vehicle coordinate frame by a rotation of θ_{vs} degrees, a user-defined parameter. Again, the rotation of the of the vehicle coordinate frame with respect to the global coordinate frame (in 2-D) is given by the angle θ_{vg} whose value is obtained from VREP. These values are used to calculate θ_{sg}, the orientation of the sensor main axis with respect to the global coordinate frame. Furthermore, the angle θ_{sv} between the global x-axis and the line connecting the locations of the sensor and detected vehicle can be calculated using the position information provided by VREP. Therefore, azimuth angle α from the sensor axis can be evaluated as:

(Eq. 2) $$\alpha = \theta_{sv} - \theta_{sg} = \theta_{sv} - \left(\theta_{vg} - \theta_{vs}\right) = \theta_{sv} - \theta_{vg} + \theta_{vs}$$

It is important to remember that the angle θ_{vs} is subtracted from the angle θ_{vg} although the figure indicates a summation. This is due to the right-handed definition of the vehicle coordinate frame; the value of the angle must be negated.

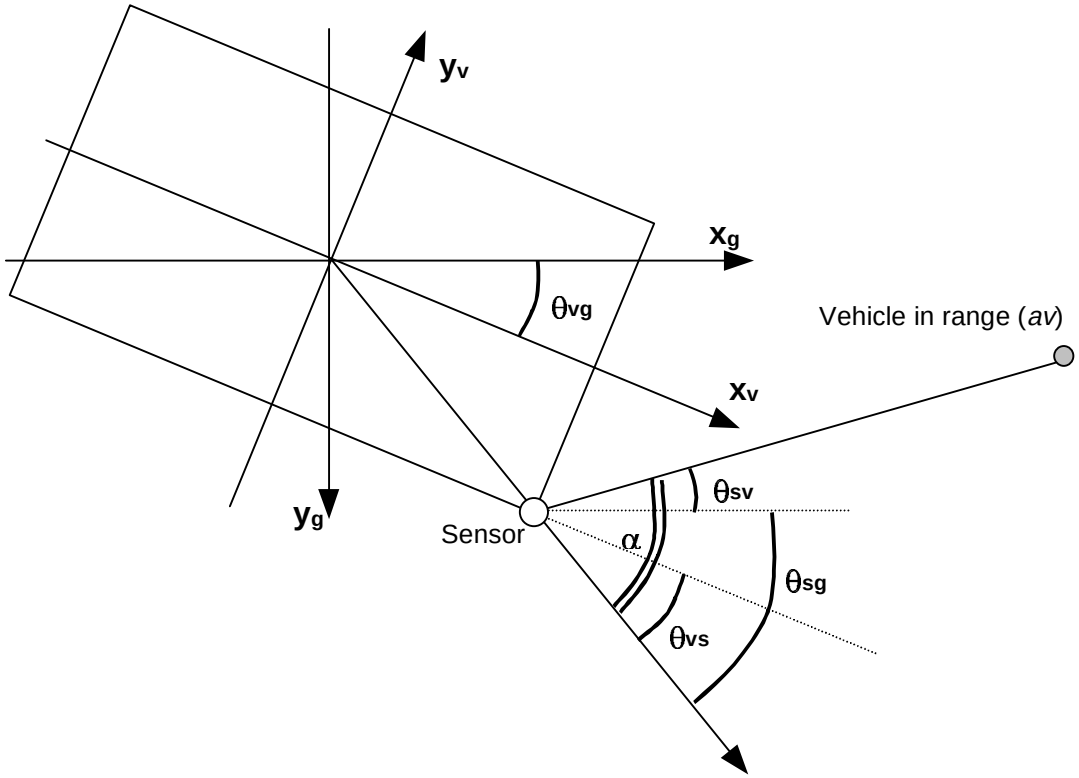

Figure 4. Alignment of sensor axis with global and vehicle coordinate frames.

Using the vehicle-to-global alignment matrix *VGAM*, the position of the sensor can be evaluated as:

$$\text{(Eq. 3)} \qquad \begin{bmatrix} x_{sg} & y_{sg} \end{bmatrix} = \begin{bmatrix} x_{sv} & y_{sv} \end{bmatrix} \cdot \begin{bmatrix} vgam11 & vgam12 \\ vgam21 & vgam22 \end{bmatrix} + \begin{bmatrix} gxp & gyp \end{bmatrix}$$

where x_{sg} and y_{sg} are the global position of the sensor while x_{sv} and y_{sv} denote the position of the sensor in vehicle coordinate frame. The vectors are given in row format for clarity since the alignment matrix is given in transpose form in the code (See SHIFT file *vrep.hs* in Smart-AHS package for details).

For the evaluation of the closest vehicle in range and in the field of view, a set of vehicles in range (*inrange*) is created first from the set of all vehicles except the self-vehicle. Then, a subset of vehicle in the horizontal field of view (*infield*) is evaluated from the set *inrange*. The vehicle returning the closest distance in the set *infield* is indicated as the *closest* vehicle.

2.2.1 I/O Structure and Parameters

The type *RangeSensor2* has the following inputs:

- *gxp, gyp*: Global position of the vehicle/sensor (m)
- *vgam11, vgam12*: Elements of the vehicle orientation matrix *VGAM*.
  vgam21, vgam22

The following values are returned as outputs:

- *distance*: Distance from the center of gravity of the sensed vehicle to the sensor (m).
- *angle*: Azimuth angle defined between the sensor main axis (i.e., vehicle x-axis) and the line connecting the sensor and the COG of the sensed vehicle (deg).
- *closest*: Identification number of the closest vehicle in sensor range.

The following are user-defined parameters or state variables (evaluated during processing) for the range sensor type:

- *vehicle*: Vehicle associated with the sensor.
- *maxrange*: Maximum range (m).
- *fov*: Angle defining the horizontal field of view (degrees).
- *procspd*: Processing speed for the sensor (sec).
- *xs*, *ys*: Sensor position in vehicle coordinate frame (m).
- *sor*: Sensor orientation in vehicle coordinate frame (degrees).
- *infield*: Set of vehicle in the field of view.
- *inrange*: Set of the vehicles in range.
- *t*: Time (sec)

The set of vehicles is defined globally as shown in Section 11.

2.2.2 Source Code

The SHIFT code for the second range sensor type is given below:

<rangesensor2.hs>

The function that evaluates the distance to the vehicle in range is implemented as an external C subroutine, and is linked to as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

2.3 Range Sensor with Pseudo-vertex Definitions

Range sensor with pseudo-vertex definitions uses the information from the global set of vehicles to obtain the position of all vehicles in the global reference frame. The position and the orientation of the sensor can also be defined with respect to the vehicle coordinate frame. The method used for distance and angle measurements is based on the definitions of multiple points on the edges of the rectangular space occupied by the (sensed) vehicle. For vehicle detection, range sensor checks these points instead of the center of gravity as in the previous sections. For each vehicle in range, the distance and the angle to these (currently six) points are calculated; values associated with the vehicle pseudo-vertex point with shortest range are returned (Figure 2d).

At present, the sensor position, and orientation, as well as the position and orientation of all vehicles are passed to an external function, which calculates the distance and the angle to six pseudo-vertex points for each vehicle. The distance and the azimuth angle to the closest pseudo-vertex point in the horizontal field of view are returned. Several parameters such as vehicle length, vehicle width, maximum sensor range, and field of view are also sent to external subroutine since these values may change from vehicle to vehicle. If the values for multiple points for a specific vehicle are needed, minor changes are required in the subroutine.

The position with respect to the center of gravity and the number of the pseudo-vertex points are defined in the external C function (See function *dist6* in Section 13). Depending on the application at hand, these values can be changed. For example, in a platooning simulation, 3 or more points can be defined on the “rear edge” of the vehicles for sensor readings close to actual values.

2.3.1 I/O Structure and Parameters

The type *RangeSensor_PV* has the following inputs:

- *gxp*, *gyp*: Global position of the vehicle center of gravity (m)
- *vgam11*, *vgam12*: Elements of the vehicle orientation matrix *VGAM*
  vgam21, *vgam22*

The following are returned as outputs:

- *distance*: Distance from the closest pseudo-vertex point to the sensor (m)
- *angle*: Azimuth angle defined between the sensor main axis and the line connecting the sensor and the pseudo-vertex point on the sensed vehicle (deg).

The following are user-defined parameters or state variables (evaluated during processing) for the range sensor type:

- *maxrange*: Maximum range (m)
- *hfov*: Half of the horizontal field of view (degrees)
- *xs, ys*: Sensor position in vehicle coordinate frame (m)
- *sor*: Sensor orientation in vehicle coordinate frame (degrees)
- *procspd*: Processing speed for the sensor (sec)
- *inrange*: Set of the vehicles in range
- *t*: Time (sec)
- *vl, vw*: Vehicle length and width (m)

The set of vehicle is defined globally as shown in Section 11.

2.3.2 Source Code

The SHIFT code for the range sensor with pseudo-vertex definitions is given below:

<rangesensor_pv.hs>

The function that evaluates the distance and the azimuth angle to the closest pseudo-vertex point is implemented as an external C subroutine, and is linked as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

2.4 Range Sensor with Ray Definitions

Range sensor with ray definitions uses the information from the global set of vehicles to obtain the position of all vehicles in the global reference frame. The position and the orientation of the sensor can be defined with respect to vehicle coordinate frame. The method used for this sensor model is based on a subroutine that calculates the intersection of two straight lines in two-dimensional space.

The user defines the resolution of the range sensor by the number of scanning rays and the maximum horizontal field of view. Since the position and orientation of the sensor with respect to the vehicle coordinate frame, and the position and orientation of the vehicle with respect to global coordinate frame are known, the equation of the lines representing the rays can be evaluated. The same is true for the line segments defining the sensed vehicles (Figure 5).

Using the definitions of Figure 5 and (Eq. 1), the slope/orientation θ_i of a scanning ray is given as:

(Eq. 4)
$$slope_i = sensor\ orientation + ray\ orientation$$
$$\theta_i = \left(\theta_{vg} - \theta_{vs}\right) + \left(hfov - \frac{2 \cdot hfov}{r-1} \cdot \left(i-1\right) \right)$$

where θ_{vg} is the orientation of the vehicle with respect to the global coordinate frame, θ_{vs} is the orientation of the sensor with respect to the vehicle coordinate frame (the minus sign is required since the vehicle coordinate frame is right-handed), *hfov* is the angle defining half of the horizontal field of view, *r* is the number of scanning rays, and *i* is the index of the ray.

At present, the sensor position, and orientation, as well as the position and orientation of all vehicles and the deviation of the rays from the sensor main axis, are passed to an external function. This function calculates the intersection points (four for each vehicle— before discarding the points outside the segments defining the sensed vehicle) of the scanning rays with the vehicles, and evaluates the distance to the closest intersection point for each ray. A vector of ranges is then returned.

The minimum value in this vector is taken as the range, and the index of the minimum is used to calculate the azimuth angle. Of course, there may be a slight deviation from the actual range and azimuth angle values due to low resolution of the scanning rays.

The model employs a second external subroutine for calculating the minimum of an array and its index. This additional subroutine is chosen due to complexity of realizing the same in SHIFT language. This subroutine will be replaced with its SHIFT counterpart in the future.

The vertex points for the segments representing the vehicles are defined in the external C function (See Function *dist5* in Section 13.1.3). The vertex points are currently chosen to define rectangular vehicle shapes. With the addition of more vertex points, defining other vehicle shapes is possible though computationally tasking.

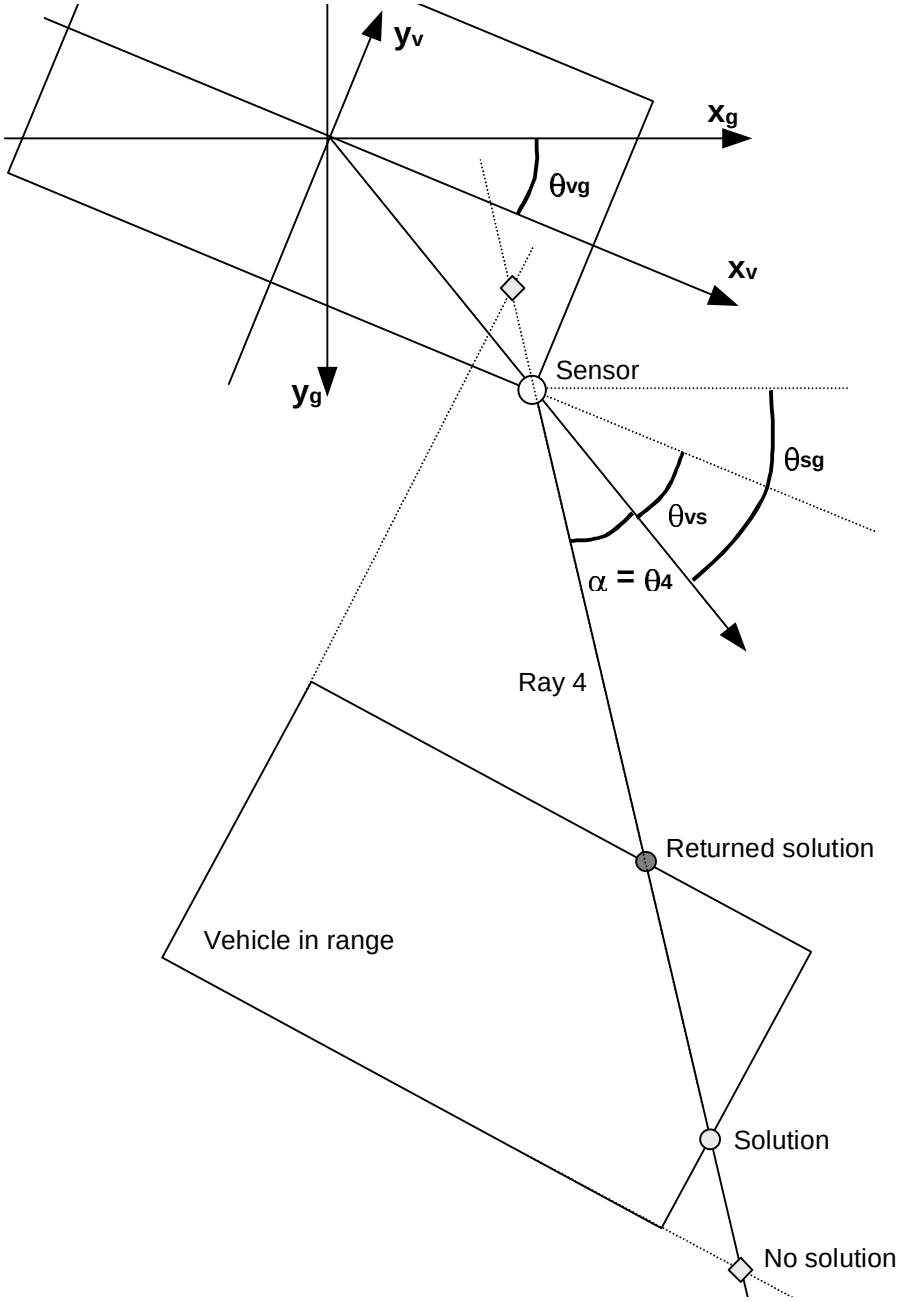

Figure 5. Definition of the sensor and scanning rays.

2.4.1 I/O Structure and Parameters

The type *RangeSensor_R* has the following inputs:

- *gxp*, *gyp*: Global position of the vehicle/sensor (m)
- *vgam11*, *vgam12*, *vgam21*, *vgam22*: Elements of the vehicle orientation matrix *VGAM*.

The following values are returned as outputs:

- *distance*: Distance from the edge of the sensed vehicle to the sensor (m)
- *angle*: Azimuth angle (*i.e.*, the angle associated with the intersecting ray; degrees).

The following are user-defined parameters or state variables (evaluated during processing) for the range sensor type:

- *vehicle*: Vehicle associated with the sensor.
- *maxrange*: Maximum range (m)
- *hfov*: Angle defining half the horizontal field of view (degrees).
- *nray*: Number of scanning rays.
- *procspd*: Processing speed for the sensor (sec).
- *xs*, *ys*: Sensor position in vehicle coordinate frame (m).
- *sor*: Sensor orientation in vehicle coordinate frame (degrees).
- *rayvec*: Array of ray slopes/orientations
- *inrange*: Set of the vehicles in range.
- *range*: Returned readings for all rays (m).
- *t*: Time (sec)
- *vl*, vw: Vehicle length and width (m).

The set of vehicles is defined globally as shown in Section 11.

2.4.2 Source Code

The SHIFT code for the range sensor with scanning ray definitions is given below:

<rangesensor_r.hs>

The function that evaluates the distance to the closest intersection point is implemented as an external C subroutine, and is linked as shown in the first few lines of the program code above. Another function for finding the minimum valued element of an array and its index is also implemented. For the description of these functions, see Section 13 on page 39.

2.5 Addition and Subtraction of Angles: Problems around ±Π

All sensor models (and/or their external C functions) include the following line of codes or similar lines:

```
number diff := atan2(vgam12,vgam11)-sor;
number sen_or := if abs(diff) < 2*PI-abs(diff)
                            then diff
                           else -(2*PI-abs(diff))*signum(diff);
```

The reason for this is the definition of the variable indicating angles and/or slopes. All such variables are defined in the interval $[-\Pi, \Pi]$. When two such values are added or subtracted, the result may be outside of the predefined interval. *If-then* condition given above is actually implementing a function mapping the resulting values back to the interval when there is a need for such an adjustment (Figure 6 on page 14).

2.6 Preprocessing for Range Sensors

Initially, all sensor modules are designed to take the global set of vehicles as input for evaluating vehicles in range. This approach results in extensive computational effort –especially for large number of vehicles–, and needs to be eliminated. Instead of checking the whole set of existing vehicle on the simulation platform, it is sufficient to consider a subset of vehicles that are in the sensor range of the vehicle in question. The limiting area for a "sensor sweep" is illustrated in Figure 7. The size of the rectangular filtering area is then $(vl + 2mr) \times (vl + 2mr)$ where *vl* is the vehicle length, and *mr* is the maximum sensor range. Two approaches currently used in Smart-AHS projects are:

- Grids with user-defined sizes (part of the communications project).
- Definition of *cells* in road segments for use with *sensor environment processor SEP*.

For detailed description of these methods, see the latest Smart-AHS distribution. As of December 1997, a global grid definition is added to this sensor suite. The details of the grid, and its cell structure, as well as the sensor environment processor associated with these definitions are given in Section 9.

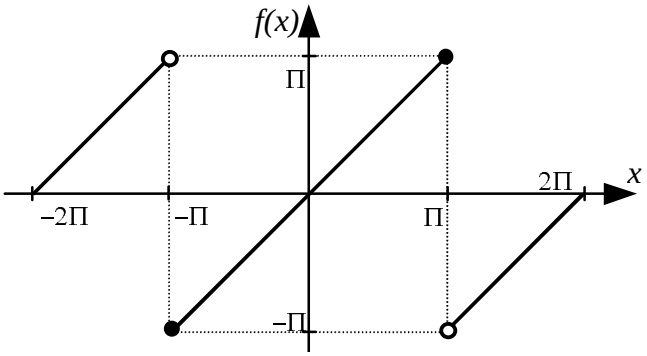

Figure 6. Function mapping resulting values back into the defined interval after summation and/or subtraction.

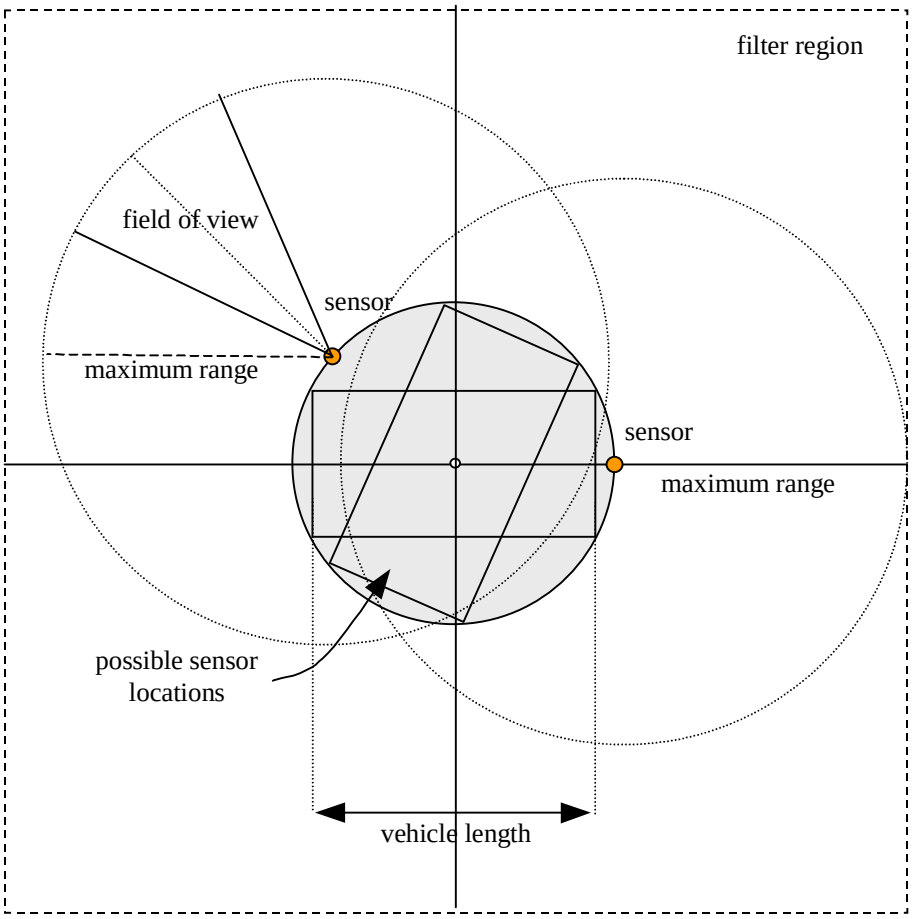

Figure 7. Sensor range definitions for filtering the vehicle set.

3 Range Sensors for Human Driver Models

There are currently four different implementations of functional range sensors specifically used for human driver models in SHIFT (See [5] for details of human driver models). These are actually two different models: same lane detection, and adjacent lane detection. The first model is used for front- and rear-looking sensors while the second approach is needed for side sensors.

The main difference between these sensors and the sensors described in Section 2 is the identification procedure for vehicles in range. Sensors in Section 2 can also be used for other applications such as multiple autonomous robot simulations while those described in this section are implemented as human perception modules, and require specific Smart-AHS definitions such as *VREP*, and highway components.

Instead of "filtering" the vehicles in sensor range and/or field of view, these sensors use the Smart-AHS highway definitions in SHIFT, and consider only the vehicles in a specific lane and its up and down connecting lanes. Since *vehicle roadway environment processor* (*VREP*) provides the lane, segment and section information [3], the evaluation of the adjacent lanes is straightforward.

Sensor range is still a user-defined variable for these sensor models, and indicates the longitudinal distance from the sensor/vehicle location. As seen in Figure 8, "longitudinal" distances are calculated using the information about the position of the vehicles on a road section, and therefore indicate the distance between vehicle COGs on the circular curve representing the left edge of the road section.

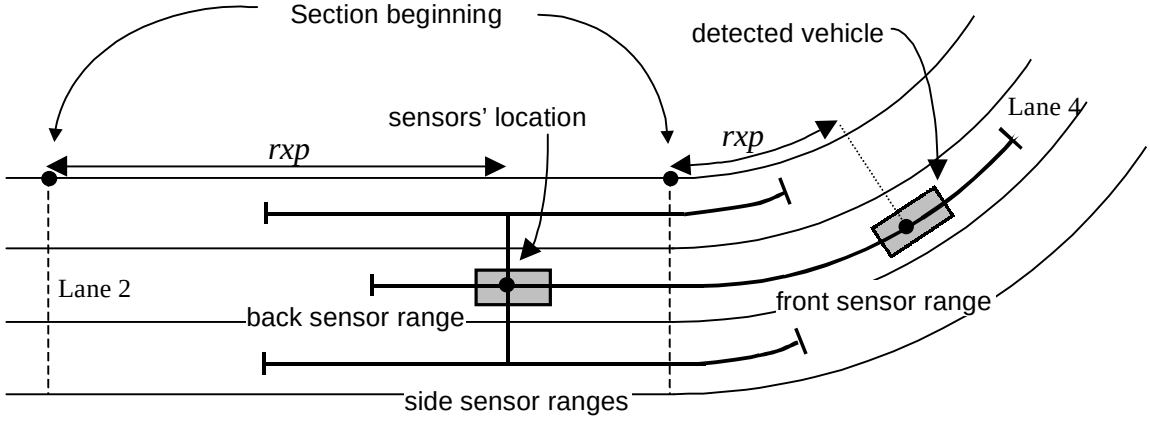

Figure 8. Definition of the sensor ranges for front, back, left and right sensors.

The detection subroutine is slightly different for four sensor types described in detail below. First, up and/or down stream connecting lanes to the lane in question are evaluated depending on the sensor type. For example, the left-sensor module checks for up *and* down stream lanes connected the lane on the left side of the current vehicle lane. Then, the vehicles, which occupy the lane in question or its connected lanes, are filtered based on the maximum sensor range(s) defined by the user.

Since vehicle positions in the road section, namely the variable *rxp*, is used for evaluations, the connecting up and/or down stream sections must also be considered: the sensor range may extend beyond the limits of the current section. For example, front sensor module looks for vehicles that satisfy range conditions and are (a) in the same section and in the same lane or (b) in the section connected downstream to the current section and in the lane connected down stream to the current lane. For the second case, the vehicle road position *rxp* cannot be used directly for calculating the distance between vehicles due to its definition (See source codes in Section 3.1.2 and the figure above for details).

The sensors are designed to provide data at user-defined frequencies. However, due to drastic changes in the sensor environment during lane changes, they are forced to "sample" range information when the vehicle moves over the adjacent lane. This "forced sampling" is obtained by synchronizing

the sampling state transition of the sensor(s) to the external *Driver* event *new_lane* which is in turn synchronized to the exported events *updateLaneRight* and *updateLaneLeft* of type *VREP*. See source codes given below for details.

Sensor models described here use the highway descriptions of Smart-AHS in SHIFT (See [2, 3] for details). Currently, we assume that there is only one element in arrays *downSection* and *upSection*. The Smart-AHS platform is capable of supporting multiple definitions in these arrays; if multiple sections are connected, the source code needs to be changed. See source codes given below for details.

3.1 Front and Back Sensors

Range sensors for human driver models use the roadway information provided by the simulation platform and the vehicle position information from the set of vehicles. Distance calculations are based on the road (section) coordinates of the vehicles providing their relative positions from the beginning of road section that they are traveling. Euclidean distance and azimuth angle are not computed; only the "longitudinal" difference between the vehicle COGs is returned.

The vehicles that are in the same lane or in the up/downstream lane connected to the self-vehicle's lane are filtered as the *vehicles in range*. The calculation of the distance is slightly different for vehicles on a connected section; the length of the section must also be taken into account because of the definition of the parameter *rxp*. The vehicle with closest distance to the sensor/self-vehicle is chosen from the set of vehicles in range. Vehicle ID as well as its longitudinal distance are returned. If there are no vehicles in the current lane and/or up/downstream lane satisfying the range condition, a sensor reading of –1 is returned.

3.1.1 I/O Structure and Parameters

The types *FrontSensor* and *BackSensor* have the following inputs:

- *rxp*: Longitudinal position of the vehicle with respect to the beginning of the road section (m).
- *gxp*, *gyp*: Global position of the vehicle/sensor (m)
- *lane*: Current lane.
- *section*: Current section.

The following values are returned as outputs in both types:

- *closest*: Identification number of the closest vehicle in sensor range.
- *distance*: COG-to-COG distance from the closest vehicle in range to the self-vehicle (m)

The following are user-defined parameters or state variables (evaluated during processing) for both range sensor types:

- *vehicle*: Vehicle associated with the sensor.
- *range*: Maximum range of the sensor (m)
- *sr*: Sampling rate (processing speed) for the sensor (sec).
- *t*: Time (sec)
- *lanesDown*: Set of downstream lanes connected to the current lane (for *FrontSensor;* the parameter is *lanesUp* for *BackSensor*).
- *inrange*: Set of vehicles in range.
- *vir*: Temporary variable for closest vehicle.

The set of vehicles is defined globally as shown in Section 11.

3.1.2 Source Codes

The SHIFT code for the front-looking range sensor is given below:

<frontsensor.hs>

The SHIFT code for the rear-looking range sensor is given below:

<backsensor.hs>

3.2 Left and Right Sensors

Left and right range sensors for human driver models use the roadway information provided by the simulation platform and the vehicle position information from the set of vehicles. Distance calculations are based on the road (section) coordinates of the vehicles providing their relative positions from the beginning of the road section that they are traveling. Euclidean distance and azimuth angle are not computed; only the longitudinal distance between the vehicle COGs is returned.

The vehicles that are in the adjacent left/right left or in the up and downstream lane connected to the left/right lane are filtered as the *vehicles in range*. The calculation of the distance is slightly different for vehicles on a connected section; the length of the section must also be taken into account because of the definition of the parameter *rxp*. The vehicles with closest longitudinal distance on the front and at the back are chosen from the set of vehicles in range. Vehicle Ids as well as their longitudinal distances to the sensor location are returned. If there are no vehicles in the current left lane and/or up and downstream lanes satisfying the range condition, a sensor reading of –1 is returned. Sensor output is -2 if there is no adjacent lane on the left/right.

3.2.1 I/O Structure and Parameters

The types *LeftSensor* and *RightSensor* have the following inputs:

- *rxp*: Longitudinal position of the vehicle with respect to the beginning of the road section (m).
- *gxp, gyp*: Global position of the vehicle/sensor (m)
- *lane*: Current lane.
- *section*: Current section.

The following values are returned as outputs in both types:

- *fclosest*: *bclosest*: Identification numbers of the closest vehicles in sensor range (front and back).
- *fdistance*: *bdistance*: COG-to-COG distances from the closest vehicles in range to the self-vehicle.

The following are user-defined parameters or state variables (evaluated during processing) for both range sensor types:

- *vehicle*: Vehicle associated with the sensor.
- *frange, brange*: Maximum range of the sensor in meters (front and back).
- *sr*: Sampling rate (processing speed) for the sensor (sec).
- *t*: Time (sec)
- *lanesDownLeft*: *lanesUpLeft*: Set of downstream lanes connected to the current lane (for *LeftSensor;* the parameters are *lanesDownRight* and *lanesUpRight* for *RightSensor*).
- *inrangef*: *inrangeb*: Sets of vehicles in range (front and back).

- *virf*, *virb*: Temporary variables for closest vehicles (redundant).

The set of vehicles is defined globally as shown in Section 11.

3.2.2 Source Codes

The SHIFT code for the left-side range sensor is given below:

<leftsensor.hs>

The SHIFT code for the right-side range sensor is given below:

<rightsensor.hs>

3.3 Possible Extensions to HDM Range Sensor Models

There are three possible extensions to the range sensors that use roadway information from the Smart-AHS platform. These are:

- Extension of the side sensor capabilities to the second lane on the right/left:
  In order to provide information for more complex path decisions, the sensors (human perception models) can be assumed to detect vehicles beyond the adjacent lane. The implementation of this addition is straightforward.
- Addition of rate information to sensor outputs:
  Besides the distance and vehicle ID, the rate of change of the measured distance can be useful to human driver models in making intelligent path decisions. If this capability is not implemented in the associated HDM model (see examples in [5] for a straightforward implementation), the rate of change can be provided by the range sensor. An example is given in Section 3.3.1.
- Addition of the second closest vehicle to sensor outputs:
  Since human drivers try to and can look beyond the vehicle in front, the second closest vehicle must be returned by the sensor module in order to implement human perception more accurately. Implementation of this capability is slightly difficult then the previous one, though feasible. A simple approach would be deleting the vehicle returned as closest vehicle from the set of vehicles in range, and repeat the detection subroutine on the resulting set. An example if this is given in Section 3.3.2.

3.3.1 Providing Rate of Change in Headway: *FrontSensor_Rate*

The sensor type *FrontSensor_Rate* is a direct extension of the type *FrontSensor*. The only addition is the evaluation of the rate of change in measured distance by comparing two consecutive readings. The source code of *FrontSensor_Rate* is given below:

<frontsensor_rate.hs>

3.3.2 Providing Information on Second Closest Vehicle: *FrontSensor2*

The sensor type *FrontSensor2* is a direct extension of the type *FrontSensor*. The only addition is the definition of another subset of vehicles in range in adjacent lane in question by eliminating closest vehicle from the initial set of vehicles. By applying the same detection condition to this new set, second closest vehicle is obtained. The SHIFT code for *FrontSensor2* is given below:

<frontsensor2.hs>

4 Roadside Range Sensors

Two roadside sensor models given in this section are similar to previous sensor models except the fact that there are associated with road segments instead of vehicles. Their position and/or orientation are defined at the beginning of the simulation run based on the parameters given by the user. Input/output structure and evaluation of vehicles in range is adapted from previous sensor models given in Section 2.

The first sensor model can return the Euclidean distance to the vehicles in range. The orientation of the sensor is not important. The second sensor model can also return the azimuth angle from the main sensor axis, which is defined with respect to the *x*-axis of the road coordinate frame at the sensor position.

4.1 Roadside Sensor

This functional sensor type uses the information from the global set of vehicles to obtain the position of all vehicles in the global reference frame. The sensor position on a road segment is given by its distance from the beginning of the road segment along the arc defining the left edge of the road, and its lateral deviation from the left edge of the road segment. Associating sensor position with a road segment enables the user to define sensor positions easily and more clearly. The transformation of these definitions into global coordinates requires additional computations based on the road segment's global position, orientation, and curvature.

If the position of a sensor with respect to a road segment is defined with *sx* and *sy* in two-dimensional space, then the global position $[xgs, ygs]^T$ of the sensor is given by:

(Eq. 5)
$$\begin{bmatrix} xgs \\ ygs \end{bmatrix} = \begin{bmatrix} gxa \\ gya \end{bmatrix} + \begin{bmatrix} \cos(\beta) & \sin(\beta) \\ -\sin(\beta) & \cos(\beta) \end{bmatrix} \cdot \begin{bmatrix} \overline{xs} \\ \overline{ys} \end{bmatrix}$$

where *gxa*, *gya* are the global position of the left edge of the road segment at its beginning when looking downstream, β is the global orientation of the road segment, and $\overline{xs}$ and $\overline{ys}$ are given as:

(Eq. 6)
$$\begin{bmatrix} \overline{xs} \\ \overline{ys} \end{bmatrix} = \begin{cases} \begin{bmatrix} sx \\ sy \end{bmatrix} & \text{if } \rho = 0 \\ \begin{bmatrix} \frac{1}{|\rho|} \cdot \sin(sx \cdot |\rho|) + sy \cdot \sin(sx \cdot |\rho|) \cdot sign(-\rho) \\ -\frac{1}{|\rho|} \cdot (1 - \cos(sx \cdot |\rho|)) \cdot sign(-\rho) + sy \cdot \cos(sx \cdot |\rho|) \end{bmatrix} & \text{otherwise} \end{cases}$$

where ρ is the curvature of the road segment. Its sign is positive for left turns, and negative for right turns. Figure 9 illustrates the parameters described above for a generic case. Multiple negative signs in the equations are kept in order to facilitate the understanding of the evaluation equations, and debugging in later stages.

Once the global position of the sensor is calculated, the evaluation of the distance to the vehicle in range is straightforward. The evaluation of the closest vehicle in the sensor range is the same as in the sensors described in Section 2.

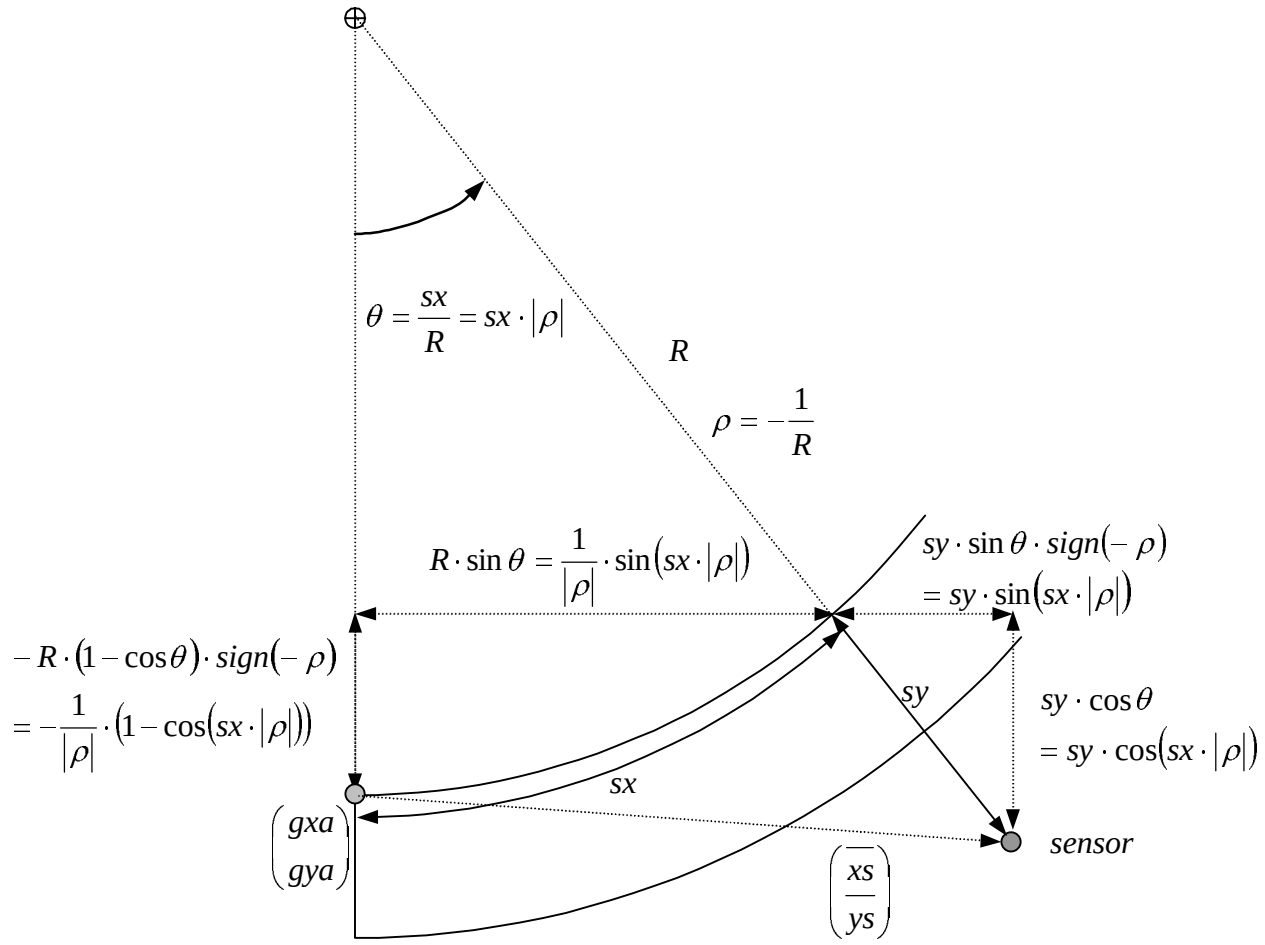

Figure 9. The global position of the roadside sensor relative to the road segment.

4.1.1 I/O Structure and Parameters

The type *RoadSideSensor* has no inputs.

The following values are returned as outputs:

- *distance*: Distance from the center of gravity of the sensed vehicle to the sensor (m).
- *closest*: Identification number of the closest vehicle in sensor range.

The following user-defined parameters or state variables (evaluated during processing) for the roadside sensor type:

- *segment*: Segment associated with the sensor
- *sx, sy*: Sensor position on the segment (m)
- *maxrange*: Maximum range (m)
- *procspd*: Processing speed for the sensor (sec)
- *inrange*: Set of the vehicles in range.
- *t*: Time (sec)
- *xgs, ygs*: Global position of the sensor (m)
- *rho*: Curvature of the road segment (1/m)
- *R*: Radius of curvature of the segment (m)
- *beta*: Orientation of the segment (rad)
- *sxbar, sybar*: Sensor position relative to the road segment (in global coordinate frame; m)

The set of vehicles is defined globally as shown in Section 11.

4.1.2 Source Code

The SHIFT code for the simplest roadside sensor is given below:

<roadsidesensor.hs>

The function that evaluates the distance to the vehicle in range is implemented as an external C subroutine, and is linked as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

4.2 Roadside Sensor Type 2

This type is similar to the previous type except the fact that the orientation of the sensor as well as its position is a user-defined parameter. However, there is no field of view definition for this sensor. The orientation of the sensed vehicle with respect to the main sensor axis (defined with respect to the road coordinate frame at the sensor position) is returned in degrees.

The position of the sensor is again defined on the associated road segment, the global position is evaluated at the setup phase of the simulation. Orientation of the sensor is defined with respect to the road segment; it is the difference between the main sensor axis and the *x*-axis of the road coordinate frame at the position of the sensor. Clockwise direction is positive for azimuth angle calculations. Figure 10 illustrates the parameters related to sensor orientation.

As seen from Figure 10, global orientation θ of the roadside sensor can be calculated using:

$$\begin{aligned} Sensor\ orientatio_{(Global)} &= Segment\ orientation_{(Global)} \\ &\quad + Orientation\ due\ to\ road\ \ psoition \\ &\quad + Sensor\ orientation_{(Road)} \\ \theta &= \beta + \alpha + \sigma \end{aligned} \qquad \text{(Eq. 7)}$$

For the sensor shown in Figure 10, a road orientation of $\frac{\Pi}{2}$ radians will cause the sensor to return an azimuth angle of zero for a vehicle that is positioned at *sx* meters from the beginning of the segment. For the same vehicle the closest possible distance reading would be *ryp*-(-*sy)* = *ryp*+*sy*.

Once the global orientation of the sensor is known, the calculation of the azimuth angle to the vehicle center of gravity is straightforward: the evaluation is the same as in the sensors described in Section 2.

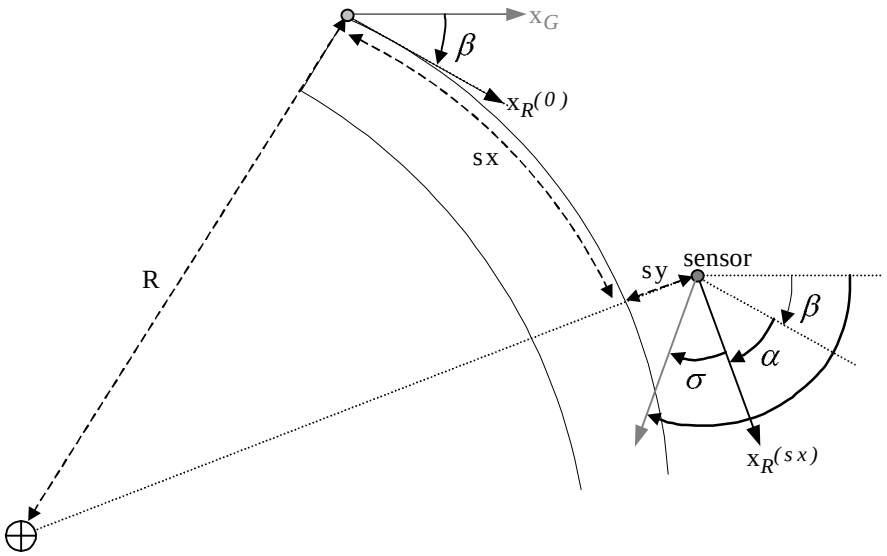

Figure 10. Global proentation of the roadside sensor located at (*sx*, *sy*) on the road segment.

4.2.1 I/O Structure and Parameters

The type *RoadSideSensor2* has no inputs.

The following values are returned as outputs:

- *distance*: Distance from the center of gravity of the sensed vehicle to the sensor (m).
- angle: Azimuth angle defined between the sensor main axis and the line connecting the sensor and the COG of sensed vehicle (degrees).
- *closest*: Identification number of the closest vehicle in sensor range.

The following user-defined parameters or state variables (evaluated during processing) for the roadside sensor type:

- *segment*: Segment associated with the sensor.
- *sx, sy*: Sensor position on the segment (m)
- *sorseg*: Sensor orientation with respect to the road coordinate frame at the sensor position (rad)
- *maxrange*: Maximum range (m)
- *procspd*: Processing speed for the sensor (sec)
- *inrange*: Set of the vehicles in range.
- *t*: Time (sec)
- *xgs, ygs*: Global position of the sensor (m)
- *sen_or:* Global orientation of the sensor (rad)
- *rho*: Curvature of the road segment (1/m)
- *R*: Radius of curvature of the segment (m)
- *beta*: Orientation of the segment (rad)
- *sxbar, sybar*: Sensor position relative to the road segment (in global coordinate frame; m)
- diff1, diff2: Temporary variables for angle summation (taking care of the problem around $\pm\Pi$).

The set of vehicles is defined globally as shown in Section 11.

4.2.2 Source Code

The SHIFT code for the roadside sensor is given below:

<roadsidesensor.hs>

The function that evaluates the distance to the vehicle in range is implemented as an external C subroutine, and is linked as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

4.3 Other possibilities

Roadside sensors similar to HDM sensor models given in Section 3 are also possible. These sensors will return only the longitudinal distance between the point that the sensor is "attached" to the road segment and the COG of the vehicle in range.

The most detailed roadside sensor should include parameters such as height of the sensor with respect to the road surface, and scanning ray definitions, and must take into account three-dimensional positioning information about the roadway and the vehicle.

5 Speed sensor

This functional sensor type uses the information provided by the *vehicle roadway environment processor* (*VREP*) to model a simple implementation. The sensor is assumed to be reading the current speed of the vehicle with a small Gaussian zero mean error distribution. In other words, the vehicle longitudinal speed provided by *VREP* is taken, and corrupted with Gaussian noise, which is generated by a separate function. The mean and variance of the measurement error are user-defined parameters.

The speed sensor has more than one operation mode (currently two; this number can be increased with minor changes in the source code). The percentage value of the precipitation on the road segment that the vehicle is traveling is taken and used to deteriorate the speed measurement. This is implemented by using two different "operation states" for the sensor; the only difference between the two states is the variance of the measurement error which is set during state transitions.

The variance and mean characteristics of the measurement error can also be a continuous function of the precipitation percentage. The model given here is used as an initial test of the different operation modes. The main use of these operational states is the implementation of fault mode for complex sensor types. More detailed information about the vehicle environment and/or condition can be relayed by environment processor VREP.

5.1 I/O Structure and Parameters

The type *SpeedSensor* has the following inputs:

- *xDot*: Actual longitudinal speed of the vehicle (m/s)
- *precip*: Percentage precipitation on the current road segment (%)

The following value is returned as output:

- *spd_reading*: Sensor measurement (m/s)

The following are user-defined parameters or state variables (evaluated during processing) for the speed sensor:

- *vehicle*: Vehicle associated with the sensor.
- *mean*: Mean value of the measurement error
- *var*: Variance of the measurement error
- *error_signal*: Measurement error.

- *tnoise*: Gaussian noise generator with two independent noise signals.

The states of the sensor type and the transition conditions between them are given in Table 1.

Table 1. Speed sensor operation modes and transition conditions

From ↓ to → when…	*Normal*	*problem*
Normal	-	*precip* ≥ 10
problem	*precip* < 10	-

5.2 Source Code

The SHIFT code for the simplest speed sensor is given below:

<speed.hs>

6 Position sensors

There are currently two simple implementations for position sensors: one for dead reckoning, the other for global positioning system. The position of the vehicle provided by the simulation environment is corrupted with user-defined noise model based on the sensor characteristics, and the desired output is generated.

6.1 Dead Reckoning

A positioning system using dead reckoning devices such as encoder is implemented as noisy measurements of the actual vehicle speed provided by the simulation environment. The variance of the error is assumed to be relatively smaller than a GPS measurement, the dead reckoning error grow as the measurement error due to bumps and cracks on the road accumulates over time.

The distance traveled D by a vehicle with an encoder attached to one of its wheel shafts is simply given by the equation:

(Eq. 8) $$D = \phi \cdot R_a$$

where ϕ is the wheel rotation, and R_a is the effective (actual) wheel radius. The rotation ϕ can be expressed in terms of encoder counts as:

(Eq. 9) $$\phi = \frac{2\Pi \cdot N}{C}$$

where N is the number of counts detected, and C is the encoder count per wheel revolution. Using the above equation, and massaging them into another form, we obtain the following "ideal" equation:

(Eq. 10) $$\dot{D} = \dot{\phi} \cdot R_a = \varpi \cdot R_a = \frac{v}{R_a} \cdot R_a$$

where v is the speed of the vehicle. Introducing the measurement noise, and errors in estimated wheel radius, we have:

(Eq. 11) $$\dot{D} = \dot{\phi} \cdot R_e = \varpi \cdot R_e = \left(\frac{v}{R_a} + N(m,v) \right) \cdot R_e$$

where R_e is the estimated wheel radius, R_a is the actual wheel radius (a decreasing function of time with an initial value of R_e), and $N(m,v)$ is the measurement noise with a mean of m and variance of v.

Besides the measurement and estimation errors, bumps and cracks on the road also affect the distance measurements. The effects of a bump and a crack on the distance evaluation are illustrated in Figure 11.

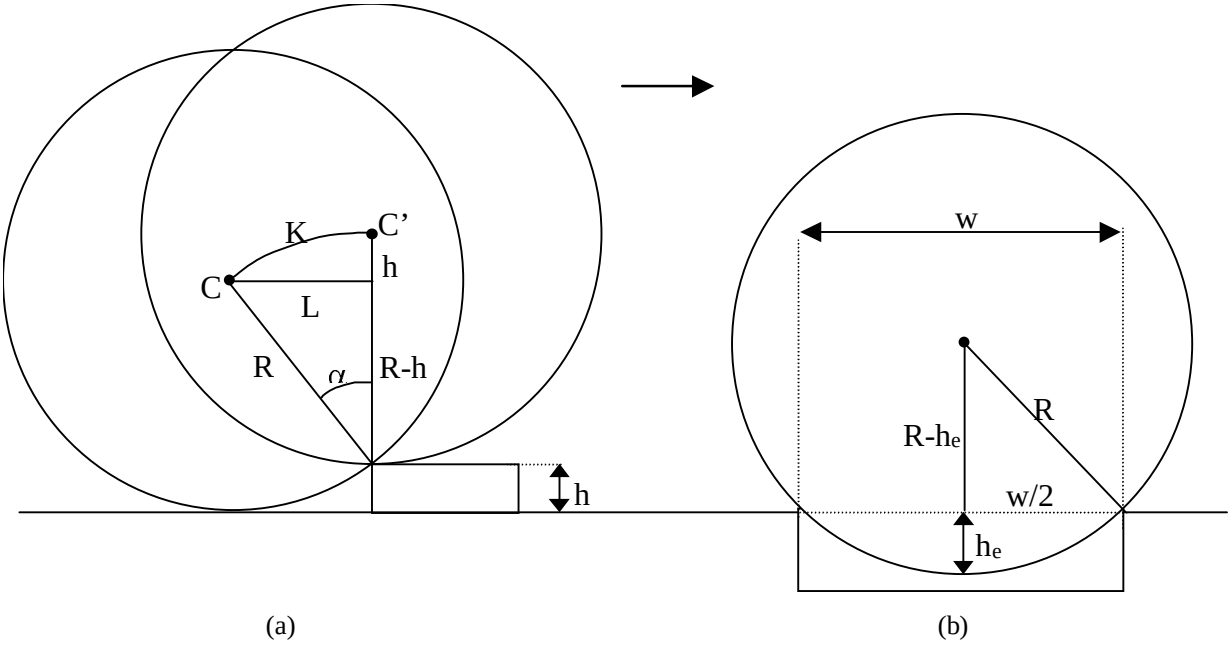

Figure 11. Distance measurement errors introduced by (a) bumps and (b) cracks on the road surface (Adapted from [9]).

When the wheel is traveling over a bump of height *h*, the measured distance is K while in fact the displacement of the wheel center is only L. Therefore the error is measurement can be evaluated as:

(Eq. 12)
$$L = \sqrt{(2R - h) \cdot h}$$
$$K = R \cdot \alpha = R \cdot \arcsin\left(\frac{L}{R}\right)$$

and:

(Eq. 13)
$$error = 2 \cdot (K - L)$$
$$= 2 \cdot \left(R \cdot \arcsin\left(\frac{\sqrt{(2R - h) \cdot h}}{R} \right) - \sqrt{(2R - h) \cdot h} \right)$$

For a crack on the road surface, given its width *w*, the effective height of the crack can be evaluated as:

(Eq. 14)
$$h_e = R - \frac{\sqrt{4R^2 - w^2}}{2}$$

(Eq. 12) and (Eq. 13) can then be evaluated using the effective height for the crack.

6.1.1 I/O Structure and Parameters

The type *DR_encoder* has the following input:

- *speed*: Actual speed of the vehicle

The following value is returned as output:

- *distance*: Measured distance

The following are user-defined parameters or state variables (evaluated during processing) for the position sensor:

- *re:* Estimated/model wheel radius (m).
- *r:* Actual wheel radius (m).
- *m_noise, v_noise:* Measurement noise mean and variance
- *arrival_bump:* Random arrival time for bumps (t)
- *arrival_crack:* Random arrival time for cracks (t)
- *bmin, bmax, cmin, cmax:* Parameters for arrival times
- *hm, hv, wm, wv:* Mean and variances for height and widths of the bumps and cracks.
- *noise:* Gaussian noise generator providing two independent signals
- *n1, n2:* Noise signals
- *t_b, t_c:* Time variables
- *theta:* Wheel revolutions (rad)
- *w, h:* Calculated width of the crack and height of the bump.
- *he:* Effective height of the crack.
- *md_c, md_b:* Variables used in effective distance error measurements.

6.1.2 Source Code

The SHIFT source code for dead reckoning is given below:

<encoder.hs>

6.2 Global Positioning System

A positioning system using a satellite network can be implemented by simply adding Gaussian measurement noise to the actual position of the vehicle. The data provided for GPS (and GLONASS) [6], and the data for differential global positioning systems obtained by Navlab vehicles [7] suggest that the longitude and latitude readings can be characterized by independent Gaussian distributions.

GPS position sensor has more than one operation mode (currently three modes; this number can be increased with minor changes in the source code). The percentage value of the precipitation on the road segment that the vehicle is traveling is taken and used to deteriorate the position measurement. This is implemented by using three different “operation states” for the sensor; the only difference between these states is the variance of the measurement error which is set during state transitions.

6.2.1 I/O Structure and Parameters

The type *PositionSensor_GPS* has the following inputs:

- *gxp, gyp*: Actual global position of the vehicle (m)
- *precip*: Percentage precipitation on the current road segment (%)

The following value is returned as output:

- *pos_x, pos_y*: Sensor readings (m)
- *err_x, err_y*; Sensor measurement errors (m)
- *s:* Measurement signal (1 = signal, 0 = no signal).

The following are user-defined parameters or state variables (evaluated during processing) for the position sensor:

- *vehicle*: Vehicle associated with the sensor.

- *mean*: Mean value of the measurement errors.
- *var*: Variance of the measurement errors (Same mean and variance for both readings).
- *error_signals*: Measurement errors.
- *tnoise*: Gaussian noise generator with two independent noise signals.

The states of the sensor type and the transition conditions between them are given in Table 2.

Table 2. GPS position sensor operation modes and transition conditions.

From ↓ to → when...	*normal*	*problem*	*nodata*
normal	-	$10 \le precip < 60$	$precip \ge 60$
problem	$precip < 10$	-	$precip \ge 60$
nodata	$precip < 10$	$10 \le precip < 60$	-

6.2.2 Source Code

The SHIFT code for GPS position sensing is given below:

<gps.hs>

7 Time Clock

A global time clock is created to make it available to all sensor types. The type *Clock* has a single output *t*. The SHIFT code for time sensing is given below:

<clock.hs>

8 Noise models

A Gaussian signal generator is implemented in order to corrupt sensor readings with measurement noise. The type *Gaussian* generates two uncorrelated signals using two uniformly distributed random variables. Both signals return values that are normally distributed around 0 with a variance of 1.

The algorithm for Gaussian noise generation using uniform distributions is taken from [8]. The uniform distributions are obtained using the SHIFT function *random()*. This functions is found to behave similar to Matlab©'s *rand* function on a Sun SparcStation© 4m running SunOS© 4.1.4[1]. Given two uniform distributions x_1 and x_2 between 0 and 1, two independent normal distributions around zero with a variance of 1 can be obtained using:

(Eq. 15)
$$y_1 = \sqrt{-2\ln(x_1)} \cdot \cos(2\Pi x_2)$$
$$y_2 = \sqrt{-2\ln(x_1)} \cdot \sin(2\Pi x_2)$$

The outputs of the noise signal generator, distributed around 0 mean with unit variance, can be used to obtain new normal distributions around different means with different variance values. Given a normal distribution *y* around zero with a variance of 1, a new normal distribution *z* around *m* with a variance *v* is obtained using the equation:

(Eq. 16) $$z = \sqrt{v} \cdot y + m$$

[1] SHIFT version 3.1 installed on the same machine returned erroneous values during run-time, possibly due to the fact that version 3.1 –designed under Solaris 2.*- does no longer support SunOS 4.1* implementations.

8.1 I/O Structure and Parameters

The type *Gaussian* has the following outputs:

- *sg1*, *sg2*: Gaussian signals with zero mean and unit variance.

The following are state variables evaluated during processing:

- *x1*, *x2*: Uniform distributions in [0, 1].

8.2 Source Code

The SHIFT code for the Gaussian signal generator is given below:

<noise.hs>

The function that evaluates the natural logarithm of a variable is implemented as an external C subroutine, and linked as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

9 Global Grid for Vehicle Detection

The section and segment descriptions in Smart-AHS libraries include cell definitions to be used with range sensors and/or vehicle detection algorithms. This definition enables the sensing and detection algorithms to work on a small subset of vehicles. The cells are defined using the existing definitions of segments and sections in Smart-AHS (See file `vrep.hs` in Smart-AHS library for details).

Here, we will define another cell structure forming a global grid for vehicle detection. Global environment that includes the highway elements and the vehicles is divided into same-size squares (cells) that facilitate vehicle detection for sensors. We call this structure the “grid” (Figure 12).

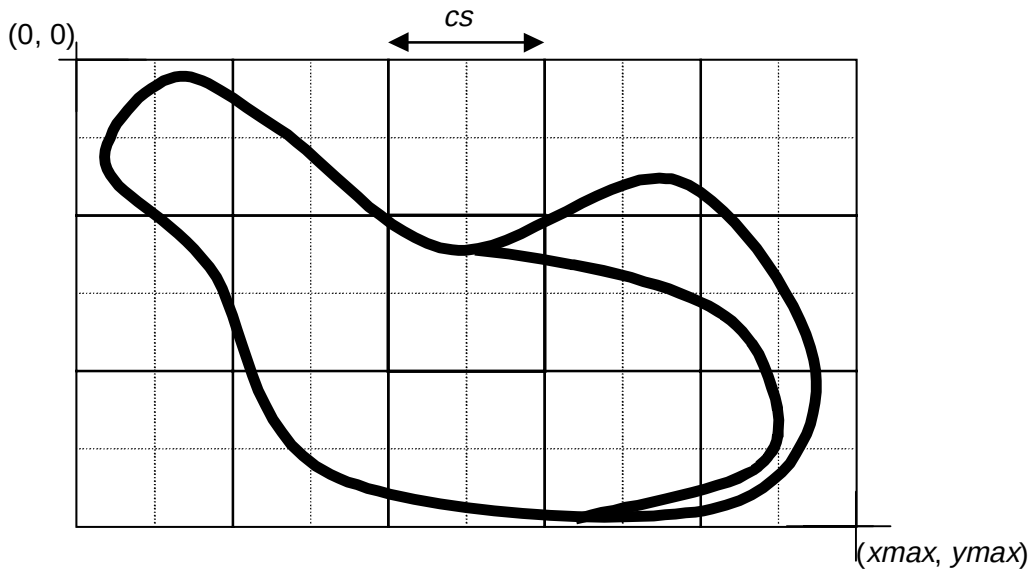

Figure 12. Definition of the grid.

The grid formed by the cells, is defined between origin point (0, 0) at the upper left corner of the 2-D representation of the world (as defined in TkShift GUI). It extends to the maximum coordinates (*xmax*, *ymax*) defined (See file `grid.hs`) by the user. These values must be chosen so that the global grid includes all vehicles and the highway sections for proper operation. It may be possible to automate the definitions of *xmax* and *ymax* in the future.

The size of the grid regions is defined with the variable *cs* (for 'cell size'). This variable must be chosen in accordance with the simulation complexity and the sensor range descriptions. It may be also possible to automate the choice of this variable based on the sensor ranges and the highway structure defined in the SHIFT description files. For all the cells defined in the grid, neighboring cells are indicated with a set *NC* of cells, which includes at most eight neighboring cells. For cells that are located at the edge of the global space definition, the number of neighbors is of course less than eight.

It is also possible to defined a smaller set of cells in order to define a grid only over the portions of the two-dimensional space with roadway definitions as shown in Figure 13. In this case, the user has to define the cell positions, sizes and neighboring cells manually in a file similar to `grid_simpler.hs`.

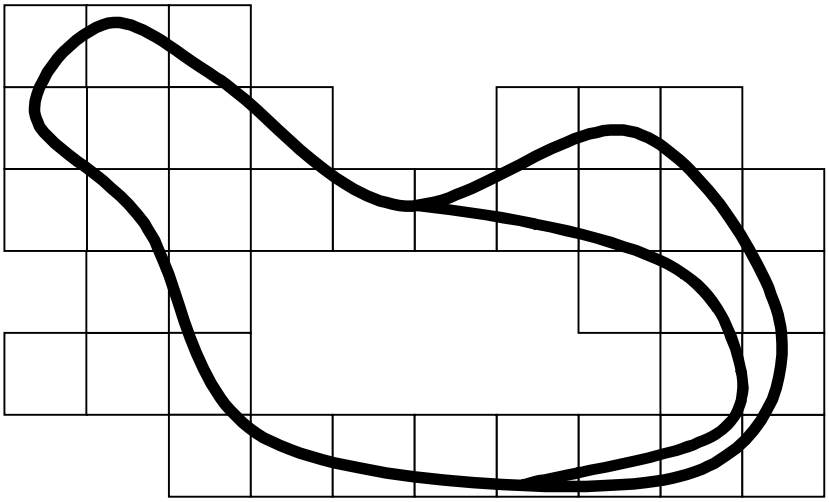

Figure 13. Simplified grid structure.

9.1 The use

Sensors types given in previous sections use the global set of *Vehicles* to detect vehicles in range. Using the grid structure, sensors will now be able to consider only a subset of the vehicles for range calculations. *Sensor environment processor SEP* will keep track of the current grid cell the vehicle/sensor is travelling in, and the set of vehicles in a specific cell can be polled for evaluation.

SEP is synchronized with the updates of the current grid cell. Whenever the cell updates its set of vehicles, all *SEPs* in this cell check to see whether the vehicle leaving the cell (if there are any) is their associated vehicle (ego-vehicle). If this is the case, then *SEP* updates its current cell parameter.

Range sensors use this information about the current cell to define a subset of vehicles to be checked for range measurements. From the parameter *currentcell* (provided by associated *SEP*), the set of cells including the current cell and all the neighboring cells, is defined. Using this set of cells, a new set of vehicles –minus the ego-vehicle- is generated, and used for distance and angle measurements.

Sampling rate for the cells is directly related to how much tolerance we have on the cell boundaries. If there is a new vehicle entering a cell, and we do not detect it for –for example- 4 seconds, it will not be taken into account by the range sensors that are checking the new cell this vehicle entered. During that time, (a) the vehicle may not even be in the sensor range (e.g., good application of cell size), or (b) the vehicle is in the sensor range, but we do not detect it until the cell updates its set of vehicles (possible problems). Hopefully, the definition of the cells relative to the sensor ranges will guarantee case (a); cell sizes and the update rates can be chosen carefully by considering the maximum speed the vehicles will attend during the simulation. On the other hand, it is always possible to increase the rate at which the cells check for leaving vehicles.

9.2 Vehicle tracking

The global grid consists of square cells for which the adjoining cells are known. The neighboring cells are defined for each cell at the beginning of the simulation right after cell creation. All the cells obtain the list of vehicles currently travelling from the source. (Therefore, the type *Source* needs to be changed as given in file `source_grid.hs`. The vehicles/sensors also obtain the cell they are traveling in initially from their source.

When a vehicle leaves a cell, this fact is detected by the cell that checks for vehicles leaving with a user-defined frequency. If the cell type finds vehicles that moved outside of its range, it then updates its set of vehicles while exporting *vehicle_leaving*. At the same time, all neighboring cells, synchronized to this exported value, check to see if the vehicle entered their region. Only the adjoining cell that finds the leaving vehicle(s) in its range, updates its set of vehicles; all other cells add null set to their set of vehicles (See code in file `grid.hs`). As described above, exported parameter *vehicle_leaving* is also used to update the current cell definition in sensor environment processors (SEPs).

If the frequency of checking for leaving vehicles is low, the number of calculations over the set of vehicles for sensor range evaluations must be much less than the number with global set of vehicles. Again, the vehicles/sensors do not check the global set of vehicles at every (sensor) iteration, but only a smaller set of vehicles in a specific number of cells (at least one). On the other hand, all the cells in the global coordinate system check a smaller set of vehicles to see if there is a vehicle leaving the cell at a predefined frequency, which is smaller than sensor frequency. Once a vehicle is found to be leaving a cell, all the neighboring cells, and all the vehicles in that specific cell must evaluate multiple sets to update their set of vehicles, or cell value. For a large number of vehicles, this grid method may prove to be more efficient than searching the whole set of vehicles.

9.3 Source Codes

9.3.1 Cell and Grid Definition Files

The SHIFT codes for the cell and grid definitions are given below:

<cell.hs>

<grid.hs>

<grid_simpler.hs>

9.3.2 Changes/Updates Required in other Smart-AHS Elements

A new sensor environment processor SEP described in the previous section is required. The SHIFT code for the environment processor is given below:

sep.hs>

Initial position and initial cell for the vehicle (and its sensors) are provided by the source. Therefore, the source definition file is changed as given below:

<source_gid.hs>

There are several changes to be made in the range sensors described in Section 2. These are:

- Addition of following line to sensor inputs:

```
input   Cell currentcell;
```

- Addition and changes given below to the evaluation of the vehicles in range:

```
/* Subset of Vehicles from SEP */
set(set(Vehicle)) vset1 := {vset(k) : k in NC(currentcell)};

/* Set of the vehicles in range
 * Instead of the whole set of vehicles, only the vehicle in the current cell
  * and neighboring cells (minus the ego-vehicle) are considered */
 set(Vehicle) inRange
      := {x : v1 in vset1, x in v1} + vset(currentcell) - {vehicle};

/* set of vehicles in the field of view */
 set(Vehicle) inField
      := {z : z in inRange
            | abs(atan2(gyp(z)-sen_y,gxp(z)-sen_x)-sen_or) < fov
              or
              2*PI-abs(atan2(gyp(z)-sen_y,gxp(z)-sen_x)-sen_or) < fov};
```

Furthermore, the following additions are to be made in the definition of the vehicle:

- Include new range sensor and sensor environment processor files:

```
#include <NewRangeSensorType>
```

- Change the definition of the outputs as follows:

```
output NewRangeSensorType sensor1, sensor2, [..];
       SEP  sep;
```

- Add the following lines to *setup* phase:
  - Define:

```
NewRangeSensorType tsensor1 := create(NewRangeSensorType, <parameters> );
...
SEP tsep := create(SEP, mycell := scell(source), myveh := self);
```

  - Do:

```
sensor1 := tsensor1;
...
sep := tsep;
```

  - Connect:

```
currentcell(tsensor1) <- mycell(tsep);
currentcell(tsensor2) <- mycell(tsep);
[...]
```

- Similar changes need to be made for the range sensors. Below, SHIFT code for type *RangeSensor2_Grid* (altered version on *RangeSensor2*) is given:

<rangesensor2_grid.hs>

10 Additional Files for Sensor Simulations

In order to test the sensor models described in this document, several Smart-AHS/SHIFT files are created. Some of these files are given in this section to facilitate the understanding of the sensor models. Inspection of these files will clarify the input/output structure of the models, and give a better understanding of the vehicle creation methods in the simulation platform.

10.1 2-D Kinematic Vehicle Model

Two-dimensional kinematic definition of a vehicle is basically a bicycle model in vehicle coordinate frame. It takes the steering angle at the front wheel (*steering*) and the acceleration of the front wheel (*acc*) as inputs, and generates the angular speed (*thetaDot*) and the longitudinal speed of the vehicle

(*xDot*) in its own coordinate frame. The distance between the front wheel and the center of the rear axle (i.e., wheelbase) is a user-defined parameter. While using this model, vehicle's center of gravity is implicitly assumed to be at the center of the rear axle.

10.1.1 Source Code

The SHIFT code for the two-dimensional kinematic vehicle model is given below:

<2dkineveh.hs>

10.2 2-D Vehicle Controller Model

The controller for the two-dimensional kinematic vehicle is realized in two parts: the longitudinal and lateral controller. The longitudinal controller is a proportional feedback controller for unit step inputs of desired speed value. The lateral controller uses a predefined look-ahead distance to generate desired lateral deviation for the pursuit point assumed to be followed by the vehicle. Based on the lateral deviation, required steering angle is evaluated. The details of the controller models are given in the following sections.

10.2.1 Longitudinal Controller

The longitudinal controller tries to track the desired speed input using a proportional control law. As long as the desired value is characterized as unit input, P-type controller successfully tracks the desired value and drives the error to zero. The error in speed is multiplied with the longitudinal controller gain (k_1) and the control input (longitudinal acceleration) to the system (vehicle) is generated. The control input to the system is limited by maximum and minimum values. The overall control is given as:

$$s \equiv k_1 \cdot \left(v_{desired} - v_{actual}\right)$$

(Eq. 17)
$$a = \begin{cases} -d_{\max} & if \;\; s < -d_{\max} \\ s & -d_{\max} \le s \le a_{\max} \\ a_{\max} & s > a_{\max} \end{cases}$$

See source code given in Section 10.2.4.1 for details.

10.2.2 Lateral Controller

The lateral controller first evaluates the relative orientation of the vehicle on the roadway by using the vehicle-to-global, and roadway-to-global orientation matrices:

(Eq. 18) $$\theta_{actual} = \operatorname{atan2}\left(vgam12, vgam1\right) - \operatorname{atan2}\left(rgam12, rgam1\right)$$

Using this value, and the curvature information about the roadway, the lateral deviation of the pure-pursuit point on the lane/roadway is calculated. The deviation of the pursuit point from the lane center given as:

(Eq. 19) $$lyp_{pursuit} = lyp_{COG} + L \cdot \sin\theta_{actual} - L \cdot \tan\left(L \cdot \frac{\rho}{2}\right)$$

where L is the look-ahead distance, and ρ is the curvature of the road at the vehicle center of gravity, defined as:

(Eq. 20) $$\rho = \frac{sign(\rho_{seg})}{\left|\frac{1}{\rho_{segt}}\right| - sign(\rho_{seg}) \cdot ryp}$$

where ρ_{seg} is the curvature of the road segment the vehicle is traveling on, and *ryp* is the lateral position of the vehicle on that segment. These values are used to correct the curvature value for the lateral vehicle position.

The first term in (Eq. 19) is the effect of the vehicle orientation on the lateral deviation of the pursuit point while the second term is due to the road curvature. The value obtained using (Eq. 19) gives the "actual" position of the pursuit point. Given the desired value for the location of this point, the curvature of the arc to be followed by the vehicle can be calculated as:

(Eq. 21) $$\rho_{travel} = \frac{2 \cdot \left(lyp_{desired} - lyp_{pursuit}\right)}{L^2}$$

The steering angle required to drive the error to zero is:

$$s \equiv k_2 \cdot a\tan\left(\rho_{travel} \cdot W\right)$$

(Eq. 22) $$\alpha = \begin{cases} -\alpha_{\max} & s < -\alpha_{\max} \\ s & -\alpha_{\max} \le s \le \alpha_{\max} \\ \alpha_{\max} & s > \alpha_{\max} \end{cases}$$

where α_{max} denotes the maximum possible steering, W is the wheelbase, and k_2 is the controller gain. The choice of the gain k_2 as well as the look-ahead distance *L* affects the behavior of the vehicle. See the source code given in Section 10.2.4.1 for details. Figure 14 illustrates the definitions given in this section.

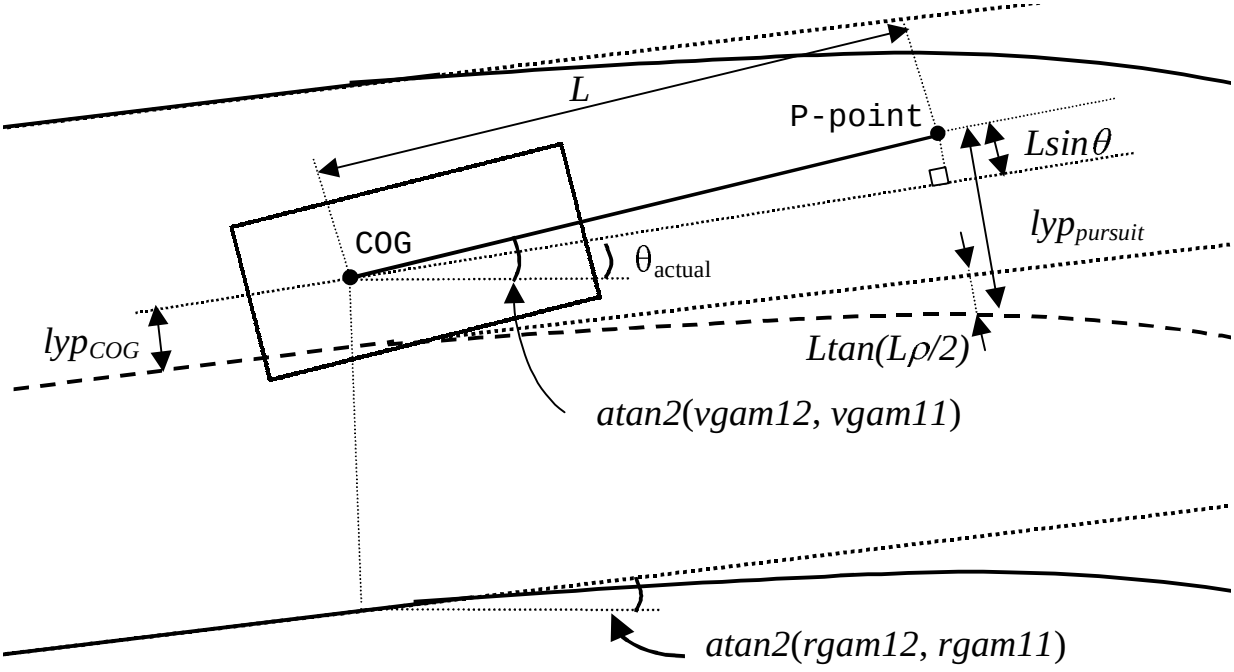

Figure 14. Pure-pursuit point definitions.

10.2.3 Tracking the Pursuit Point

An additional subroutine (SHIFT type) *Pursuit* is used to take the target lane position information from the driver (or the higher decision level in the control hierarchy), and relay it to the (lateral) vehicle controller with a predefined rate of change. This subroutine is required in order to make sure that the lateral controller is able to track the pursuit point.

A second task for this type is to guarantee smooth transitions for the desired lane position (lateral deviation) input to the lateral controller during lane changes. As seen in the highway and vehicle environment processor *VREP* descriptions in Smart-AHS [3], the lateral lane deviation value *lyp* "jumps" from $-lw/2$ to $lw/2$ during left lane changes, or vice versa (*lw* indicates the lane width). The driver model (higher control level in the hierarchy) currently uses a lane deviation value of $\pm lw$ to indicate a desired lane change. Due to the change in the actual value at transition from one lane to the other, the value of the desired deviation also needs to be changed to guarantee smooth operation. The type *Pursuit* uses a conditional transition waiting for the lane change to take care of this problem.

This module takes target lane position from driver module, current lane position and current lane for *VREP* as inputs, and generates desired pursuit point as output. The rate of change for the desired lane position and its maximum value are user-defined parameters. See the source code given in Section 10.2.4.2 for details.

10.2.4 Source codes

10.2.4.1 Controller

The SHIFT code for lateral and longitudinal 2-D kinematic vehicle controller is given below:

<2dkinectrl.hs>

he function that evaluate the inverse sin of a variable is implemented as an external C subroutine, and is linked as shown in the first few lines of the program code above. For the description of the external function, see Section 13 on page 39.

10.2.4.2 Pursuit Point

The SHIFT code for pursuit point evaluation is given below:

<pursuit.hs>

11 Scenario and Vehicle Description Files

Some of the highway/scenario and vehicle description files created for sensor modeling are given to clarify the input/output structure of the models, and facilitate the use of the models for other application and/or AHS simulations. Investigation of the files in this section will give the reader a good idea on how to connect several modeling blocks.

11.1 Highway Descriptions

11.1.1 Straight Road with Three Sections

Figure 15 illustrates the 3-section roadway definition given in file `3section.hs`. There are three *sources* the beginning of the first section, located at the middle of each lane, except the one in the left lane. A user-defined parameter *initiallanedev* is used to introduce a slight deviation from the lane center for the first source. For details, see the source code below.

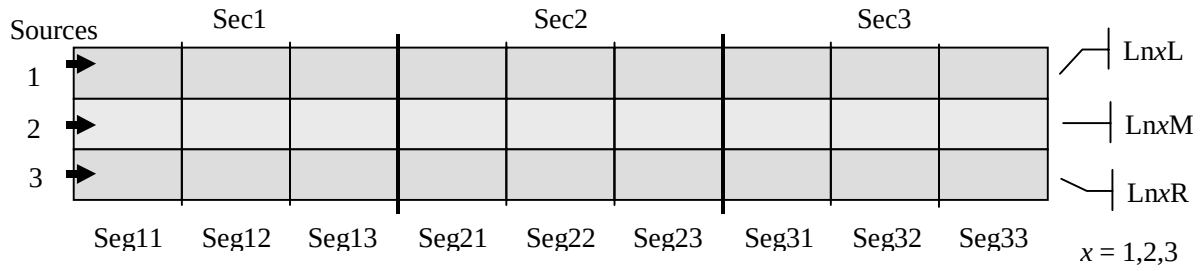

Figure 15. Straight roadway with 3 sections, 9 segments and 3 lanes.

The SHIFT code for the 3-section roadway description is given below:

<3section.hs>

11.1.2 Circular Highway

Figure 16 illustrates the 4-section circular highway given in file `circular.hs`. The file includes only one source at the beginning of lower left section. For details, see the source code below and the web page http://www.cs.cmu.edu/~unsal/research/shift/circular.html.

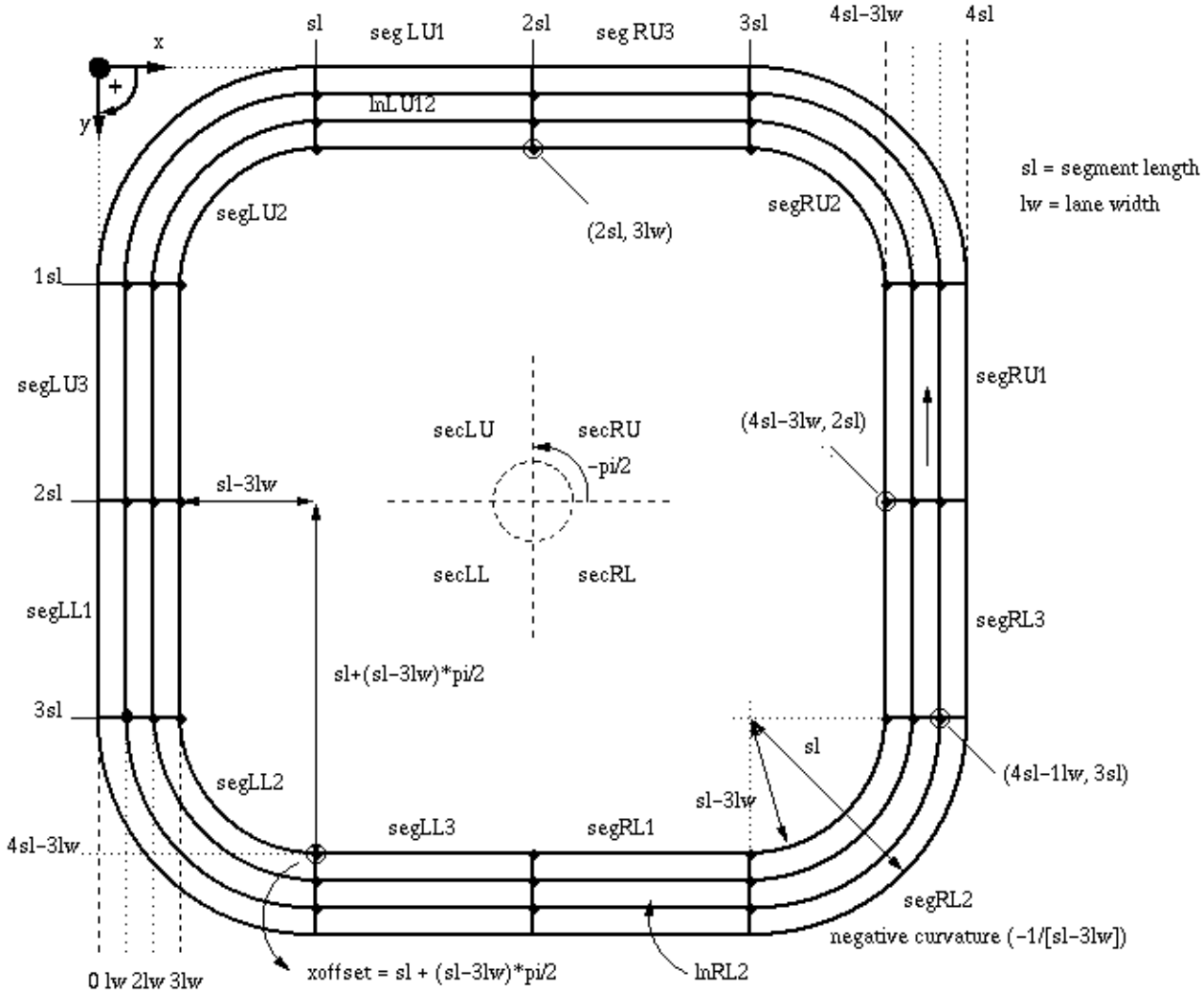

Figure 16. Circular roadway with 4 sections, 4x3=12 segments, and 3 lanes.

The SHIFT code for the circular highway description is given below:

<circular.hs>

11.1.3 Racetrack

Figure 17 illustrates the 4-section racetrack given in file `racetrack.hs`. The file includes only one source at the beginning of lower left section. For details, see the source code below and the web page http://www.cs.cmu.edu/~unsal/research/shift/track.html.

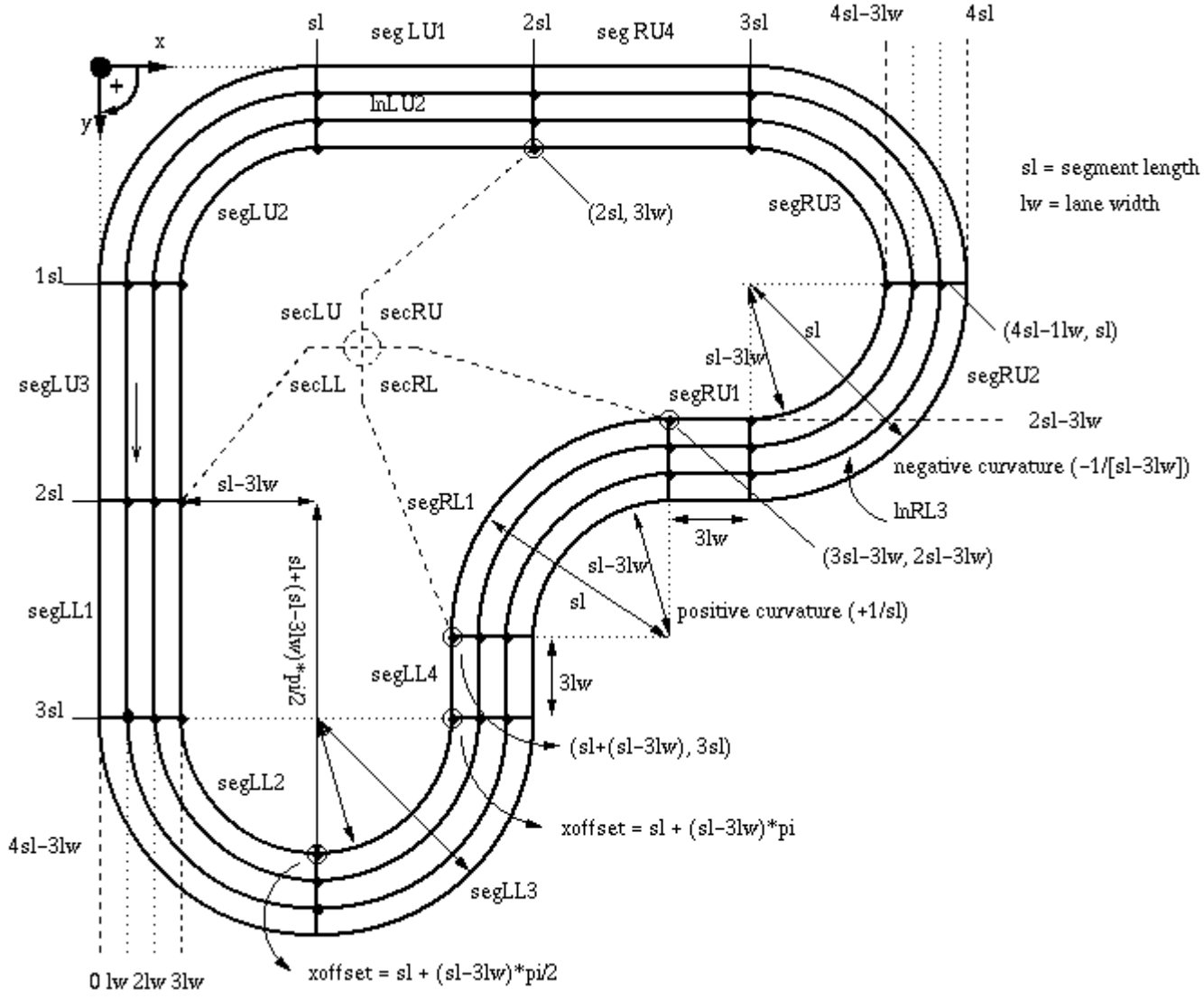

Figure 17. Racetrack with 4 sections, 3+4+1+4 =12 segments, and 3 lanes.

The SHIFT code for the racetrack description is given below:

<racetrack.hs>

11.2 Vehicle Description

An example file (`ExVehicle2.hs`) is included in this section to illustrate subtype definitions for an vehicle simulation in SHIFT/Smart-AHS platform. Figure 18 gives possible links between the subtypes constituting the vehicle.

As seen in the SHIFT code below, the set of vehicles is defined globally. Sub-modules (types) in the vehicle require inclusion of additional files for the SHIFT compiler. A generic type *Vehicle* includes a two-dimensional vehicle model (subtype *Vehicle_Kinematics*), a lateral and longitudinal control models (*Controller*), the pursuit subroutine (*Pursuit*), a simple driver model (*Driver*), and several sensor models (e.g., *RangeSensor_PV*) as well as vehicle roadway environment processor (*VREP*), source and/or a sink (*Source*, *Sink*). Global position, the velocity and the width and length of the vehicle are defined as outputs for the vehicle.

In the setup phase, sensor, controller, and other model parameters are defined; the newly created vehicle is added to the set of vehicles. The initial location of the vehicle is inherited from the associated source.

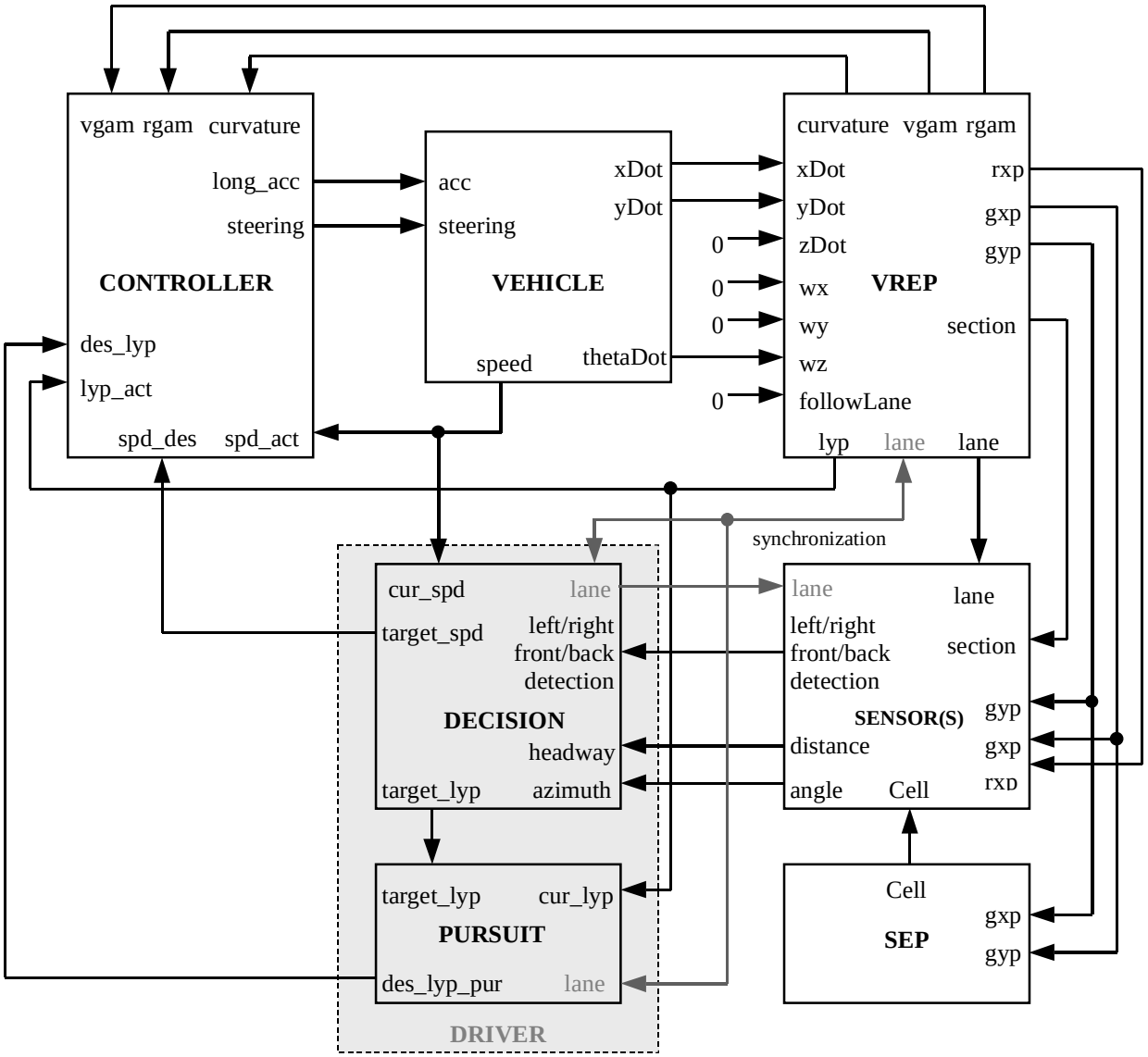

Figure 18. Connections between the subtypes for the type *Vehicle*.

The SHIFT code for the example vehicle description is given below:

<ExVehicle2.hs>

12 Coordinate frames in Smart-AHS

Three different coordinate frames are used to describe the positions of components in Smart-AHS [3, 2]. These are:

- vehicle,
- road, and
- global

coordinate (or reference) frames.

Vehicle coordinate frame is assumed to be attached to the vehicle at its center of gravity. The *x*-axis of this frame is aligned with the forward direction of motion. The *y*-axis points toward the left side of the vehicle, while the *z*-axis is directed away from the ground. This frame is in accordance with the "right hand rule." Vehicle movement is defined in this frame (e.g., lateral and longitudinal speeds).

Road coordinate frame has its origin at the point where the left edge and the line defining the beginning of a section intersect. The *x*-axis is the tangent to the left edge, and is pointing toward the direction of traffic flow. The *y*-axis is parallel to the radius of curvature, pointing to the right when looking down the direction of vehicle movement. The *z*-axis is directed away from the ground. This frame is "left-handed." It is used to define the vehicle position on the road/lane (*rxp*, *ryp*, *rzp*, and *lyp*).

Global coordinate frame is assumed to be attached to a point on the ground. We assume this frame to be a "left-handed frame" with *z*-axis pointing away from the ground. This definition is in accordance with the description of the vehicle-to-global alignment matrix (*VGAM*). The frame is used to define the vehicle's global position (*gxp*, *gyp* and *gzp*).

12.1 Relation between coordinate frames

Figure 19 shows the global, road and vehicle coordinate frames for a (very simple) highway description. As seen in the figure, the road coordinate frame is aligned with the global coordinate frame for this specific example (The segment's orientation is zero radian). Both coordinate frames are left-handed. On the other hand, the vehicle coordinate frame is not completely aligned with the global, or the road coordinate frames due to the definition of the transformation matrix (See type *VREP* in file *vrep.hs*.) Vehicle coordinate frame (for a vehicle following the road segment as shown in Figure 19) is a left-handed frame where only the *x*- and *z*-axes are aligned.

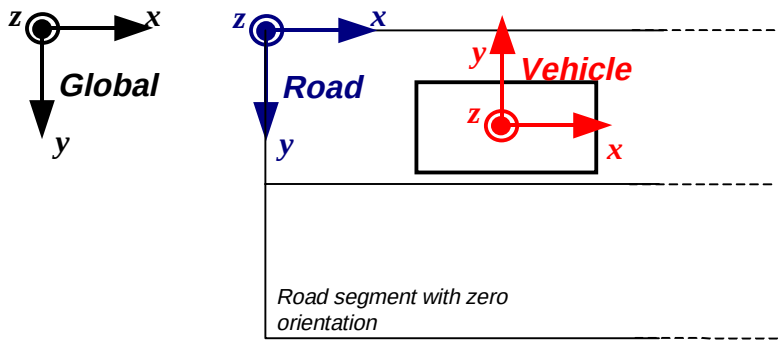

Figure 19. Reference frames in Smart-AHS.

The transformation between the vehicle and global coordinate frames cannot be described by using “standard” matrices used in robotics applications, because the former is right-handed while the latter is

left-handed. For a two-dimensional system, the transformation, given by the matrix *VGAM*, can be written as:

$$[gxp \quad gyp \quad gzp] = [xDot \quad yDot \quad zDot] \cdot VGAM$$

$$= [xDot \quad yDot \quad zDot] \cdot \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ \sin(\alpha) & -\cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{(Eq. 23)}$$

where a is the angle between x-axes of the vehicle and global reference frames. The matrix *VGAM* differs from a standard coordinate frame transformation matrix in two ways:

1. The position/velocity vectors are written as row vectors resulting in a "transposed" transformation matrix definition, and
2. The difference in the definition of the coordinate frames forces the second column of the matrix *VGAM* to be the negative of the second row of a standard transformation matrix[2].

13 External C functions

Most of the sensor models defined in this document use external functions implemented as C subroutines. It is also possible to implement some of these functions in SHIFT, depending on the sensor structure and computational effort. The external C functions are combined into a single file (`ext-func.c`) for use with the C compiler.

13.1 Function descriptions

Some of the functions given in file `ext-func.c` are described in this section. Simpler functions such as *eucdist*, *natlog*, and *arcsin* are not described due to their straightforward implementations.

13.1.1 Function dist6

This function is used by the range sensor with pseudo-vertex definitions; it is used to evaluate the distance and the azimuth angle to the closest vehicle in range, given the set of vehicles. The following are provided by the range sensor type in SHIFT as inputs:

- *senx*, *seny*: Global sensor position
- *senor*: Global sensor orientation
- *vehx*[], *vehy*[]: Global positions of the vehicles in range
- *ca*[], *sa*[]: Global orientations of the vehicles in range (Elements of *VGAM*)
- *n_of_veh*: Number of vehicles
- *vl*, *vw*: Vehicle length and width
- *maxrange*: Maximum sensor range
- *hfov*: Half the horizontal field of view

The output of the function *dist6* is the two-element array *reading*[]. The first element is the distance to the closest vehicle in range, the second is the azimuth angle.

Pseudo-vertex points are defined at the corners and at the middle of the long sides of the rectangle defining the vehicle. There are two loops for evaluation. The outer loop checks for all the vehicles

[2] The standard transformation matrix for a rotation of a radians around the z-axis is given as:

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(defined by the position, orientation vectors, and the parameter *n_of_veh*). The inner loop is completed for all the pseudo-vertex points for a given vehicle. If (a) the distance to a vertex point is less than the maximum sensor range, (b) the azimuth angle is less than half the horizontal field of view, and (c) the distance is less than the minimum range reading, then the current values are assigned to the output vector. If no such point is found, a maximum range of 1000m, and an azimuth angle of zero are returned.

13.1.2 Function *min_of_array*

This function is used by the range sensor with ray definitions. It is used to evaluate the value and the index of the minimum element in an array. The array and its size are provided by the range sensor type in SHIFT as inputs. The output is an array *result*[] that combines the minimum-valued element and its index. If there are more than one minimal value in the array, this function will return the index of the last element encountered.

This function could be replaced with its SHIFT counterpart in the future.

13.1.3 Function *dist5*

This function is used by the range sensor with ray definitions; it is used to evaluate the vector of range readings for the scanning rays, given the set of vehicles. The following are provided by the range sensor is inputs:

- *senx*, *seny*: Global sensor position
- *senor*: Global sensor orientation
- *vehx*[], *vehy*[]: Global positions of the vehicles in range
- *ca*[], *sa*[]: Global orientations of the vehicles in range (Elements of *VGAM*)
- *n_of_veh*: Number of vehicles
- *vl*, *vw*: Vehicle length and width
- *rayvec*[]: Vector defining the rays (azimuth angles from the sensor normal).

The output of the function *dist5* is an array of range readings whose length is equal to the number of scanning rays.

The vehicles are defined as rectangular regions described by four vertex points. Using the position and orientation information provided by the range sensor, it is possible to define the four lines describing the region occupied by the vehicle. The information on sensor position and orientation along with the ray definition vector *rayvec* is used to define the scanning rays. There are three loops in the subroutine. One for rays, one for vehicles and one for lines defining the vehicles. If (a) there is an intersection between a line defining the scanning ray and a line defining the vehicle, and (b) this intersection occurs in the line segment constituting an edge of the region occupied by the vehicle, and (c) the distance is less than the minimum range reading obtained for that specific ray, then the distance to the current intersection point is assigned to the corresponding element of the output vector. If no such point is found, a value of 10000 meters is returned. The azimuth angle to the closest vehicle in range is calculated using the index of the minimum-valued element of the distance vector.

13.2 Source Code

The C code for the external functions is given below:

<ext-func.c>

14 Additional Information

All source code files listed below as well as this documentation are provided at `http://www.cs.cmu.edu/~unsal/research/shift/index.html`

- 2dkinectrl.hs
- 2dkineveh.hs
- 3section.hs
- ExVehicle2.hs
- backsensor.hs
- backsensor2.hs
- cell.hs
- circular.hs
- clock.hs
- delayer.hs
- encoder.hs
- ext-func.c
- frontsensor.hs
- frontsensor2.hs
- frontsensor_rate.hs
- gps.hs
- grid.hs
- grid_simpler.hs
- humandrivers.hs
- leftsensor.hs
- leftsensor2.hs
- noise.hs
- pursuit.hs
- sep.hs
- racetrack.hs
- rangesensor.hs
- rangesensor2.hs
- rangesensor2_grid.hs
- rangesensor_pv.hs
- rangesensor_r.hs
- rightsensor.hs
- rightsensor2.hs
- roadsidesensor.hs
- roadsidesensor2.hs
- roadsidesensor2_grid.hs
- sep.hs
- source_grid.hs
- speed.hs

Sensor models and related files described here are compatible with Smart-AHS versions 0.45 and 0.60 [2], and were tested under SHIFT version 2.12 [1] using Sun SPARC© Station 4 running SunOS 4.1.4.

The following text must included in the directory where the source files are located, under the name CONDITIONS:

```
/*****************************************************************\
* The files in this directory are distributed under the following    *
* conditions:                                                        *
*                                                                    *
* 1. The recipient shall refrain from disclosing the software,       *
*    in any form, to third parties without prior written             *
*    authorization from Carnegie-Mellon University.  The             *
*    recipient shall have the right to use and copy the              *
*    software on, or in connection with the operation of, any        *
*    computer system owned or operated by it.  In addition,          *
*    the recipient shall have the right to modify or merge           *
*    the software to form updated works.                             *
*                                                                    *
* 2. If the recipient receives a request from any third party        *
*    to furnish all or a portion of the software to any third        *
*    party,  it will refer such a request to Carnegie-Mellon         *
*    University.                                                     *
*                                                                    *
* 3. Carnegie-Mellon University shall not be held liable for any     *
*    damages resulting from the use or misuse of the software        *
*    provided by it.  Furthermore, Carnegie-Mellon University        *
*    remains without obligation to assist in its installation        *
*    or maintenance.                                                 *
*                                                                    *
* 4. The recipient agrees to acknowledge Carnegie-Mellon             *
*    University in appropriate citations appearing in public         *
*    literature when reference is made to the software provided      *
*    above.                                                          *
*                                                                    *
* 5. If the recipient develops any enhancements to the software      *
```

```
*    which materially improves its operation, the recipient          *
*    agrees to make such enhancements available to Carnegie-         *
*    Mellon University without charge, provided Carnegie-            *
*    Mellon University agrees in writing to receive such             *
*    enhancements in confidence, if requested to do so.              *
*                                                                    *
* 6. This header comment must remain attached to the source          *
*    code of the provided software.                                  *
*                                                                    *
*  Bug reports and suggestions can be mailed to Cem Unsal by         *
*  electronic mail addressed to: "unsal@ri.cmu.edu". As mentioned    *
*  in condition 3 above, the author is not obligated to fix any      *
*  such bugs, or even to acknowledge receipt of the bug report.      *
*                                                                    *
\********************************************************************/
```

15 Contact Information

The author of this document can be contacted at:

Cem Ünsal
Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890
(412) 268-5594 unsal@ri.cmu.edu
(412) 268-5571 (fax) http://www.cs.cmu.edu/~unsal/

16 References

[1]The SHIFT Team, “SHIFT, the Hybrid System Simulation Programming Language,” California PATH/University of California-Berkeley, http://www.path.berkeley.edu/shift (September 19, 1997).
[2] The SHIFT Team, “California PATH Smart-AHS,” California PATH/University of California-Berkeley, http://www.path.berkeley.edu/smart-ahs (September 19, 1997).
[3] A. Deshpande, “AHS Components in SHIFT,” California PATH/University of California-Berkeley Report, http://www.path.berkeley.edu/shift/doc/ahs.ps.gz (September 19, 1997).
[4] C. Ünsal, R. Sukthankar, C. Thorpe, “Functional Sensor Models for AHS Simulations,” SPIE International Symposium on Intelligent Systems and Advanced Manufacturing, Technical Conference on Intelligent Transportation Systems, Pittsburgh, PA, October 1997, http://www.cs.cmu.edu/~unsal/publications/spie97son.pdf (September 13, 1997).
[5] C. Ünsal , “Human Driver Models for AHS Simulations,” Technical Report CMU-RI-TR-98-01, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, February 1998, http://www.cs.cmu.edu/~unsal/research/shift/hdm.pdf.
[6] P.N. Misra, M. Pratt, R. Muchnik, B. Burke, and T. Hall, “GLONASS Performance: Measurement Data Quality and System Upkeep,” *Proceedings of ION GPS-96*, pp. 261-270, The Institute of Navigation, 1996.
[7] T. Jochem, D. Pomerleau, B. Kumar, and J. Armstrong, “PANS: A Portable Navigation Platform,” *IEEE Symposium on Intelligent Vehicles*, Detroit, MI, USA, September 25-26, 1995.
[8] W.H. Press, S.A. Teukolsky, and B. Flannery, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*, Cambridge University Press, Cambridge, MA, 1992.
[9] H.R. Everett, *Sensors for Mobile Robots: Theory and Application*, A. K. Peters, Ltd., Weslesley, MA, 1995.