Phobetor: Robust, Parallellized Computer Vision in a Real-time Autonomous System

Joshua Newman[a], Han Zhu[a], Brenton A. Partridge[a], Laszlo J. Szocs[b], Solomon O. Abiola[b], Ryan M. Corey[a], Srinivasan A. Suresh[b], and Derrick D. Yu[a]

[a]Dept. of Electrical Engineering, Princeton University, Princeton, NJ, USA;
[b]Dept. of Mechanical & Aerospace Engineering, Princeton University, Princeton, NJ, USA

ABSTRACT

In this paper we present *Phobetor*, an autonomous outdoor vehicle originally designed for the 2010 Intelligent Ground Vehicle Competition (IGVC). We describe new vision and navigation systems that have yielded 3x increase in obstacle detection speed using parallel processing and robust lane detection results. *Phobetor* also uses probabilistic local mapping to learn about its environment and Anytime Dynamic A* (AD*) to plan paths to reach its goals. Our vision software is based on color stereo images and uses robust, RANSAC-based algorithms while running fast enough to support real-time autonomous navigation on uneven terrain. AD* allows *Phobetor* to respond quickly in all situations even when optimal planning takes more time, and uses incremental replanning to increase search efficiency. We augment the cost map of the environment with a potential field which addresses the problem of "wall-hugging" and smoothes generated paths to allow safe and reliable path-following. In summary, we present innovations on *Phobetor* that are relevant to real-world robotics platforms in uncertain environments.

Keywords: robotics, IGVC, stereo vision, lane detection, obstacle, occupancy map, path planning

1. INTRODUCTION

The Intelligent Ground Vehicle Competition (IGVC)* is an annual collegiate robotics competition held at Oakland University in Rochester, MI. Teams gather to compete in four challenges: the Autonomous Challenge consists of lane-following and obstacle avoidance, the Navigation Challenge is a timed open-field waypoint following and obstacle avoidance challenge, and the JAUS Challenge tests integration and compliance with the military's JAUS protocol. Finally, the Design Challenge consists of a technical paper submission and a presentation session.

Princeton University's undergraduate-lead robotics research group, *Princeton Autonomous Vehicle Engineering* (***PAVE***), has been participating in the competition since 2008. *Phobetor*, our entry into the 2010 IGVC, improves upon our previous entries with updated hardware and software systems that are described in this paper. While several of the robots systems have been engineered to the criteria of the IGVC, the computer vision, obstacle avoidance, and autonomous navigation systems onboard *Phobetor* have applications in many situations outside of competition, including search-and-rescue, exploration, law enforcement, or even household assistance.

2. HARDWARE

Phobetor's design was focused around meeting the challenges of the IGVC. At 31" wide, 37" long and 69.5" tall, the robot is a robust and reliable autonomous platform capable of navigating around its environment without any human aid. *Phobetor's* chassis is laid out in a bi-level design and constructed out of 80/20 T-slot aluminum framing. The lower section houses the drive and power systems; these consist of a pair Victor 885 24V motor controllers, and two NPC-T64 motors rated at 0.7 hp each, which ensure good mobility in uneven terrain. Six lead-acid batteries in a series-parallel configuration provide 24V DC power, and two step-down converters provide 12V and 5V power to the system's computer and sensors. The batteries are capable of supplying enough power for approximately two hours of autonomous navigation. *Phobetor* features a tricycle wheelbase with two rear

Send correspondence to Joshua Newman: joshuan@princeton.edu
*http://www.igvc.org/

wheels powered by the motors and a leading caster. Additionally, it is equipped with snowblower off-road tires to provide superior traction in a variety of terrains and may operated in the rain without damaging any of its electronics. The system also has a low center of gravity, allowing the robot to navigate inclines up to 10 degrees on grass.

Figure 1. *Phobetor* at the IGVC in Rochester, MI.

Phobetor's midsection houses all of its electrical modules. A LabJack UE9 allows the robot's single on-board computer to access all low-level electronic signals through a single interface. Two US Digital HB6M high-precision, low-drift wheel encoders and the motor controllers are routed through the LabJack which interfaces with the computer via ethernet, as well as Gladiator G50 MEMS gyroscope which is used to compensate for wheel slippage that the encoders are unable to detect. Additionally, a radio receiver located within the tower allows *Phobetor* to be driven using a conventional RC system.

Environmental sensing is carried out by a single Videre 15cm baseline stereo-on-chip (STOC) color camera. Its onboard Field Programmable Gate Array (FPGA) computes the three dimensional point cloud, reducing computational load on the computer and improving resolution and refresh rate. The camera is mounted at the top of the robot's tower so as to provide *Phobetor* with the viewing angle of a human. Three dimensional features may be detected as close as 20cm from the front caster and as far away as 20m. A Hemisphere A100 GPS unit provides location data but very low-accuracy heading, so an OceanServer OS5000 digital compass is used in tandem.

Although *Phobetor* was built around the guidelines of the IGVC, it was nevertheless intended to be a robust and highly applicable robotics platform that can demonstrate a wide range of autonomous behaviors. Its comprehensive and complementary sensor array and durable hardware make it an exceptionally reliable system that is capable a handling a wide range of possible tasks outside of its original design parameters.

3. SOFTWARE OVERVIEW

Figure 2 describes our software architecture. The red boxes represent the robot's hardware, including sensors and low-level interfaces. The blue boxes represent elements of our software architecture. Each system runs asynchronously to allow parallel processing, and the systems communicate through the Inter Process Communication (IPC)[1] library. The details of this implementation are similar to our previous robots [2, 3.1].

Lane detection is described in more detail in section 4.2; obstacle detection in section 4.1; occupancy mapping in section 4.3.1; state estimation in section 5.1; path planning in section 4.3.2, and path tracking and low-level control in 5.2. Algorithms are presented in Appendix A.

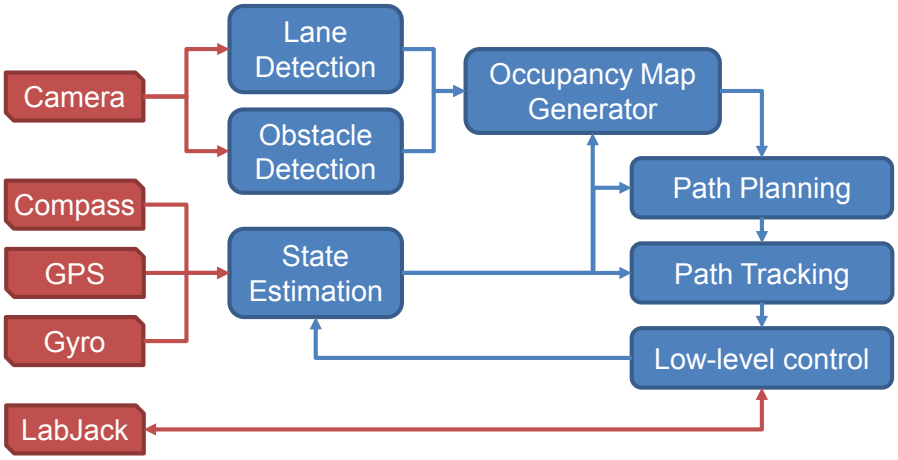


Figure 2. Block diagram of software architecture.

4. VISION

Obstacle detection is critical for any autonomous, mobile robot operating in an unfamiliar environment. While we designed *Phobetor* with the IGVC competition in mind, it is a reliable autonomous platform for outdoor applications, and any application from robot soccer player to unmanned military scout would require similar obstacle detection capabilities. We chose to address our sensory requirements using a single color, stereo camera. Compared to other sensors such as laser range-finders, stereo vision offers a strategic advantage as a passive sensor; can acquire a wider variety of scene information including colors, textures, and distances; and is more economical. We describe below two algorithms for obstacle detection as well as two robust approaches for lane detection.

4.1 Obstacle Detection

In the IGVC competition, the obstacles comprise a mix of cones, barrels, fences, and other objects with dimensions on the order of one meter. The terrain is mainly flat, with some ramps at approximately 15 degrees and some natural slopes. Considering this environment, we implemented a simple detector that identifies obstacle points by their height above the ground and corrects for non-flat regions using a robust ground plane detector. We also implemented a parallelized version of the obstacle detection algorithm given by Manduchi et al.[3] because the ground-plane fitting was inadequate for terrain with more complex topographies, which might be much more common outside of the IGVC.

4.1.1 Robust Ground Plane Fitting

A very simple approach to obstacle detection with a depth sensor such as a stereo camera is to determine the height of each point observed by the sensor, and classify anything higher than what the robot can safely traverse as an obstacle. An implementer would set the threshold some distance above the ground to minimize false-positives resulting from depth measurement errors. This naïve approach fails on non-flat surfaces or those with a varying ground plane, when knowledge of the camera's orientation with respect to the horizontal is uncertain. We present a robust alternative that locates the local ground plane and appropriately adjusts height calculations, improving the consistency of our detection rate.

Our ground plane fitting algorithm uses RANSAC (Random Sample Consensus)[4] (Algorithm 1). Each iteration j of the algorithm starts by selecting a random set of points I'_j from the point cloud P. The set I'_j are the initial inliers. We then fit a model m'_j (described below) to the the points I'_j. Next, we determine the set of points $I_j \subseteq P$ which fit model m'_j closely, and take these as the new inlier set. We fit another model m_j, our hypothesis, with the inlier points I_j and score the model m_j based on how well it fits the inlier set I_j. After a fixed number of iterations, the best hypothesis $m^* = \max_j(m_j)$ and corresponding inlier set I^* is chosen.

The models m_j and m'_j describe a plane and have the form $[a\ b\ c][x\ y\ z]^T = 0$, where $[a\ b\ c]$ is a unit vector and $[x\ y\ z]$ are the coordinate axes. The algorithm also computes the distance from the origin (a fixed point on the robot) to the ground plane, because on uneven ground or ramps we cannot assume that the robot is actually on the ground plane. Each model m_j is evaluated by the scoring function

$$s(m_j) = \sum_{p \in I_j} \frac{1}{1 + \left[d_{m_j}(p)\right]^2}$$

where $d(p)$ is the distance from the point to the ground plane m_j.

We evaluate the output of the algorithm to make sure it is reasonable. The model is discarded if its inlier set does not include a significant portion of the viewable area, because we assume that obstacles are relatively sparse in the robot's environment. A model that deviates significantly from the expected ground (that is, with slope greater than some threshold) is discarded. This may occur when the robot is very close to an obstacle in front of it, and that obstacle occupies a large portion of the robot's field-of-view.

After a model is found, obstacle detection thresholds distances of each point in the image from the ground plane to label individual pixels as obstacles. The number of iterations of the algorithm can be tuned to improve accuracy or improve response time, and the computational efficiency depends on the least squares implementation. According to Eberly,[5] the computational complexity of fitting is linear in the number of points, so for a fixed number of algorithm iterations the fitting process is linear in the number of pixels in the image.

Sample results of the ground plane fitting are shown in Figure 3. Note that this image was taken on a relatively flat surface, where the ground-plane is approximately constant in the field of view. We describe in section 4.1.2 below an alternative for when this assumption is not valid. Also, we discard camera measurements at distances where camera uncertainty becomes too large and the local ground plane assumption may no longer hold.

(a) Input image from stereo camera, which has a corresponding depth map.

(b) Output of obstacle detection. Pixels detected as obstacles are overwritten in blue.

Figure 3. Results of ground-plane obstacle detection.

4.1.2 Slope-Based Approach With Parallelization

However, we found in testing that the ground plane varied enough in the field of view that obstacle detection was often inaccurate in some region. We considered an implementation of the obstacle detection technique presented by Manduchi et al.[3] however we found that for 640 by 480 pixel images, our implementation was not fast enough to keep our robot responsive. To address this problem, we utilize the multiple cores on *Phobetor's* computer by modifying the algorithm to process different sections of the image in parallel. While there are implementations

of obstacle detection algorithms for graphics processing units[6] (GPU), we chose not to add a GPU to our to minimize our computer's power consumption and extend our robot's single-charge operation time.

Instead, we chose to parallelize our implementation of Manduchi's algorithm on our quad-core computer (Algorithm 2). In summary, the algorithm looks for pairs of points in the three-dimensional point cloud with large slope between them (e.g., oriented close to vertically). Such points are called compatible, and an obstacle is a set of points connected by compatibility. The set of compatible points S for a given point p lies in cones above and below p in 3D space, which are projected to two triangles in the 2D image [3, 2.1]. The cones are truncated based on platform-specific maximum traversable terrain height, and in our implementation only the upper cone is used to avoid repeating computations. By adjusting the slope threshold (the slope of the cones) for classifying pairs of points as compatible, we can tailor terrain classification (labeling traversable or non-traversable) to the capabilities of the hardware platform. Notably, no ground plane detection is required because compatibility is based on pairs of points, not points and the ground. In our opinion this is more robust than approaches based on ground plane segmentation.

For efficiency, when looking for compatible pairs, the algorithm only searches through the pixels in triangles above that pixel, whose sizes depend on the distance from the robot.[3] The number of pixels in each triangle is proportional to the number of pixels in the image so the algorithm has computational complexity $O(N^2)$, where N is the number of pixels in the image. However, the processing of blocks of the image can be parallelized to reduce actual computation time. We also limit the size of triangles to ensure consistent computation time.

First, we ensure each pair of points is checked for compatibility only once. To parallelize the computations, we divide the image into a series of overlapping tiles. There are two types of tiles, labelled A, and B, and this series is repeated across the image. As shown in Figure 4, A blocks do not overlap with each other horizontally, but they overlap with the A block above. B blocks overlap completely with the A blocks to the left and right. The purpose of the overlap is to ensure that all possible compatibilities are checked; the points near the borders of non-overlapping tiles could not be paired with their neighbors in a neighboring tile, so if tiles were disjoint these compatibilities would be ignored. Thus, tiles must be sized such that the largest triangle will fit in a single tile.

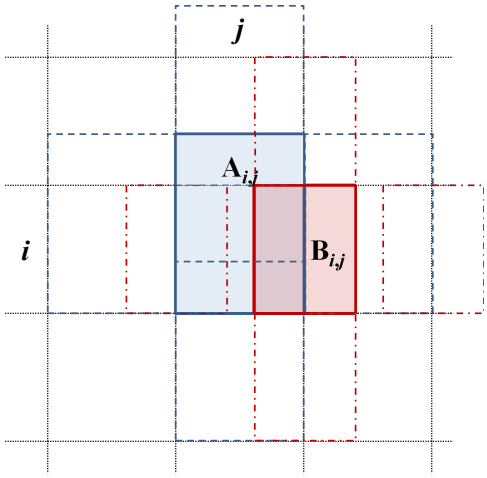


Figure 4. Overlapping tiles for parallel obstacle detection

Let $P(X_{i,j})$ be the set of points in a tile $X_{i,j}$, where $X_{i,j}$ is the X type tile in the i-th row, j-th column section. Define a function compatible_points(p, $X_{i,j}$) that returns the set of points $q \in P(X_{i,j})$ that are compatible with p. This function computes its output according to the criteria given by Manduchi.[3] Next, define

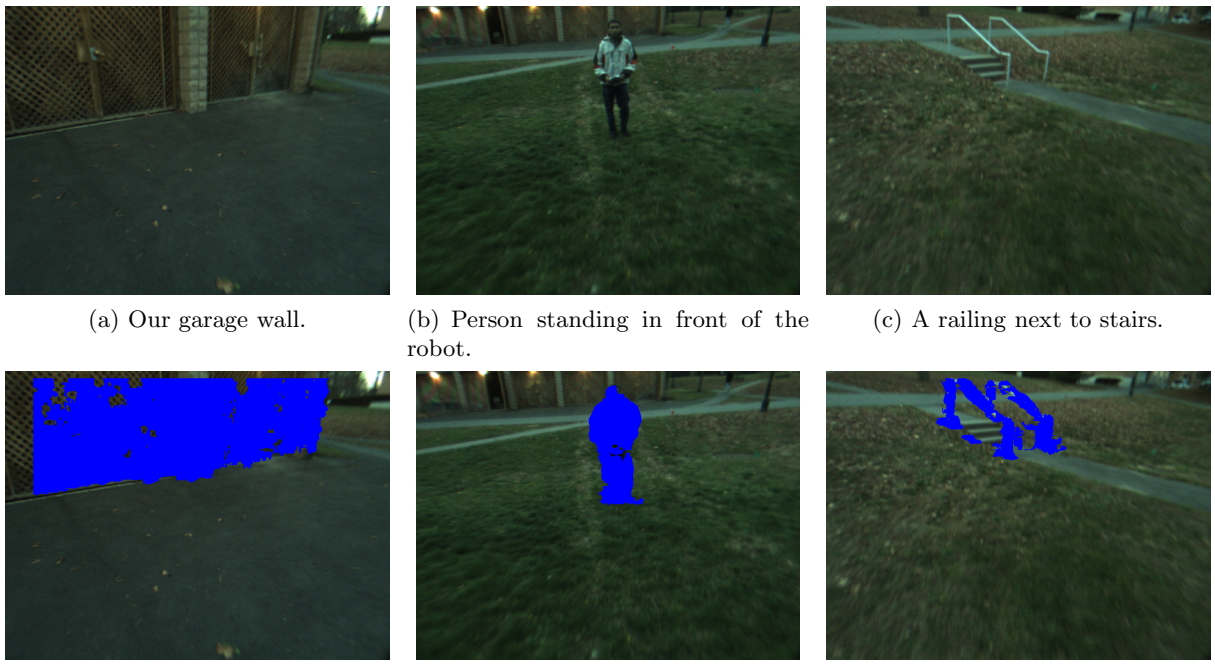
$$\text{compatible\_in\_set}(X_{i,j}) = \bigcup_{p \in P(X_{i,j})} \text{check\_compatibilities}(p, X_{i,j} \setminus p)$$

where \ is the set difference operator. Then, to evaluate all possible correspondences in the image, the algorithm computes

$$\bigcup_{i,j} (\text{compatible\_in\_set}(A_{i,j})) \cup \text{compatible\_in\_set}(B_{i,j}))$$

which computes the union over all tiles. Since compatible_in_set can be computed independently for each tile, parallelization is now straightforward. In this implementation, A-type tiles are 80 by 40 pixels so they fit in the L2 cache for lower access latency. We use OpenMP[7] for parallelization. When we timed our single-threaded implementation and the parallel one running on 640×480 test images collected from our camera, the parallel implementation took on average 150 milliseconds per frame when approximately half the frame contains obstacles, while the single-threaded implementation was 3x slower.

We also cluster the detected obstacle points in both image space and real space, discarding clusters with fewer than 20 pixels. This reduces noise in the detection.



(a) Our garage wall. (b) Person standing in front of the robot. (c) A railing next to stairs.

(d) Garage wall detected as an obstacle. (e) Person detected as an obstacle. (f) A railing detected as an obstacle.

Figure 5. Results of slope-based parallelized obstacle detection.

4.2 Lane Detection

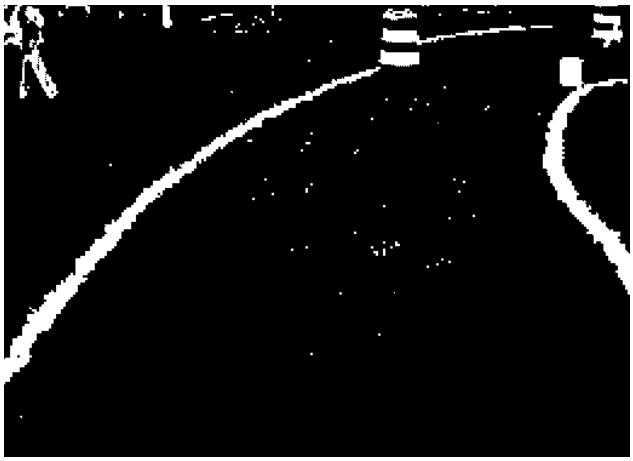
Our lane detection system was separated into two main steps. First, we processed the image to locate pixels that were part of lanes. Second, we fit a model that represented the lane. The system needs to handle images that have multiple lanes and operate in varied lighting conditions. We assumed that lanes were non-intersecting and always white (possibly sparsely or lightly painted) on grass or dirt.

To identify lane pixels in each image, we experimented with various techniques which included thresholding color values to isolate white pixels (Figure 6(c)) and looking for edges in the image using a Canny edge detector and a horizontal intensity bump filter (Figure 6(b)). However, each of these had many false positives when tuned for a sufficiently high detection rate because many obstacles had white coloration or lines that were detected as edges. We combined these two filters by only choosing pixels that satisfied both, and results on a sample image are shown in Figure 6(d).

We still expect false positive lane pixels in the filtered image, and we want to detect multiple lane lines in each image, so we use a robust RANSAC fitting technique as in Section 4.1.1. This technique was implemented

(a) Input image from stereo camera.

(b) Output of Canny edge detector and horizontal intensity bump filter.



(c) Output of white filter.

(d) Combination of edge detector and white filter.

Figure 6. Filtering steps of lane detection, which identify lane pixels.

and evaluated for each of two lane-boundary models: rotated parabolas (Section 4.2.1) and splines (Section 4.2.2). Both algorithms assume that lane boundaries tended to follow smooth curves, but spline-fitting handles more general curves than parabola models. We observed that parabolic lane models were sufficiently accurate for navigation purposes during testing and at previous IGVC competitions. However, this assumption can be challenged in other situations so to make our lane detection system robust we also present a lane modeling algorithm based on third-degree spline fitting in Section 4.2.2.

4.2.1 RANSAC for Rotated Parabola Fitting

Before fitting rotated parabolas to the filtered image, we transform each pixel in the image to the corresponding point on the ground and perform the fitting in those coordinates. Sample transformation results are shown in Figure 7(a). We use the OpenCV[8] image processing library to compute an appropriate homography offline, so the online transformation only involves matrix multiplication in homogeneous coordinates. The algorithm is presented as Algorithm 3.

Since lanes vary in orientation relatively to the robot, before using linear regression to fit a parabola to a set of points, we use principle component analysis to determine the rotation angle of the parabola. Thus, we find a

parabola $y' = f(x') = ax'^2 + bx' + c$, where the coordinates

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

are rotated from the axes in the robot's usual frame of reference. We use polynomial regression[9] for the fitting.

(a) Rectified image, simulating a top-down view. (b) Rectified image overlaid with fitted lane models.

Figure 7. Fitting of rotated parabolas to a filtered lane image.

After lane is found, we mask it out of the image so subsequent iterations can detect remaining lanes. With proper tuning of the inlier threshold, this ensures multiple models are not generated for the same physical lane. Results appear in Figure 7(b).

4.2.2 RANSAC for Spline Fitting

RANSAC has been previously applied to spline fitting for lane detection in Aly,[10] but we present a more general technique that does not require lanes to be approximately linear.

A naïve approach to using RANSAC for spline fitting chooses random k-subsets, $k > 3$, of the filtered lane points as hypothesized control points for a piecewise spline. In choosing a mathematical model for the piecewise spline, we require that

- the curve passes through each of its control points,
- the curve smoothly interpolates between control points, and
- the interpolation is computationally fast.

Catmull-Rom splines satisfy these constraints. Between control points $(x_{c,i}, y_{c,i})$ and $(x_{c,i+1}, y_{c,i+1})$, interpolated points are given by the following formula from Wang et. al.:[11]

$$\begin{bmatrix} x_i(t) & y_i(t) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{c,i-1} & y_{c,i-1} \\ x_{c,i} & y_{c,i} \\ x_{c,i+1} & y_{c,i+1} \\ x_{c,i+2} & y_{c,i+2} \end{bmatrix}, \quad t \in [0, 1]$$

A metric for the fitness of each spline is a function of the shortest distances from each point to the spline. There is no closed form expression for this distance, which must be calculated using optimization techniques.

The efficient optimization techniques discussed in Wang et. al.[12] assume that the spline segments have equal arc length, which is not guaranteed since control points are sampled randomly. We are exploring other techniques that relax this requirement.

Because the degrees of freedom increase significantly with multiple control points, hundreds of iterations are required for reasonable results and the system is too slow for real-time applications. Further research in spline fitting is ongoing, however. Future improvements include:

1. Regression to find control points from the k-subset, rather than using them as control points directly; this would allow segments of equal arc length, and permit other spline models such as B-splines. One such regression technique is described by Eberly.[13]
2. A more robust and efficient fitness metric than distance, such as the one used by Aly.[10]
3. Using probabilistic estimates of splines, such as the maximum likelihood estimate used by Wang et. al.,[11] to advise random sampling, as Aly does with linear estimates.[10]

4.3 Navigation

After lanes and obstacles are detected in the field-of-view, *Phobetor* updates a local map of the environment and then uses this map to plan an appropriate path to the next waypoint. The map generation, waypoint selection, and navigation algorithms are discussed below.

4.3.1 Occupancy Map

The environment of the robot is divided into a square grid with 10 cm dimensions. A map of a 200 meter by 200 meter environment can fit comfortably in machine memory, though for applications with larger working areas (such as an autonomous car) a modified representation might be required. We use an occupancy map[14] to represent the environment, maintaining a probabilistic estimate of whether each cell is occupied by an obstacle. For numerical reasons probabilities are stored in the log-odds representation.

To update this large map efficiently, we note that the robot's camera has a limited field-of-view and each update only changes our knowledge of the corresponding area on the map. Since we are using a stereo camera, each pixel of the image, or each detected obstacle point according to the algorithms presented above, provides an observation of a cell on the cost map. These observations can be "Obstacle", or "Free" for regions where the ground is visible, or the cell can be occluded in a particular image and provide no information. The observations are not independent in general because obstacles tend to consist of many pixels so obstacle pixels are likely located near other obstacle pixels, however we assume obstacle detection handles the situations described by Thrun et al. [14, Section 9.4.1]. With these limited range updates, the occupancy map can incorporate updates as quickly as the sensing algorithms described above can generate them.

We choose a simple inverse measurement model that assigns some log odds probability $l_{occ} > 0$ to cells corresponding to obstacles and another log odds probability $l_{free} < 0$ to cells seen as free. Occluded cells and those outside the field-of-view are not updated. With log odds probabilities, we add l_{occ} for occupied cells and add l_{free} for free cells for each processed frame. The probability for each cell is limited by minimum and maximum values to help cope with a dynamic environment. That is, in an unthresholded map, if the robot observes a cell is free for 100 frames so that the cell's value is $100l_{occ}$, if an obstacle then moves into that position ~ 100 more observations would be required before the robot's map and path planning could react.

We consider a cell occluded if there is an obstacle between that cell and the robot, and no "Free" observation for that cell. To efficiently compute an approximation, we discretize headings around the robot into N sections and create N bins $\{b_i\}$ to record the shortest distance to an obstacle in that direction, as illustrated by the dashed line-divisions in Figure 8. For each obstacle o_i the algorithm processes, it updates all bins o_i occupies if o_i is closer than previous obstacles $\{o_j : j < i\}$. After all obstacles are processed, for each cell c_i in the field of view not marked "Obstacle" or "Free," we look up the appropriate heading bin to see if c_i is closer or farther than the nearest obstacle, and classify c_i as "Free" or "Occluded," respectively. Thus, the entire map update has computation complexity linear in the number of obstacles and in the size of the field of view.

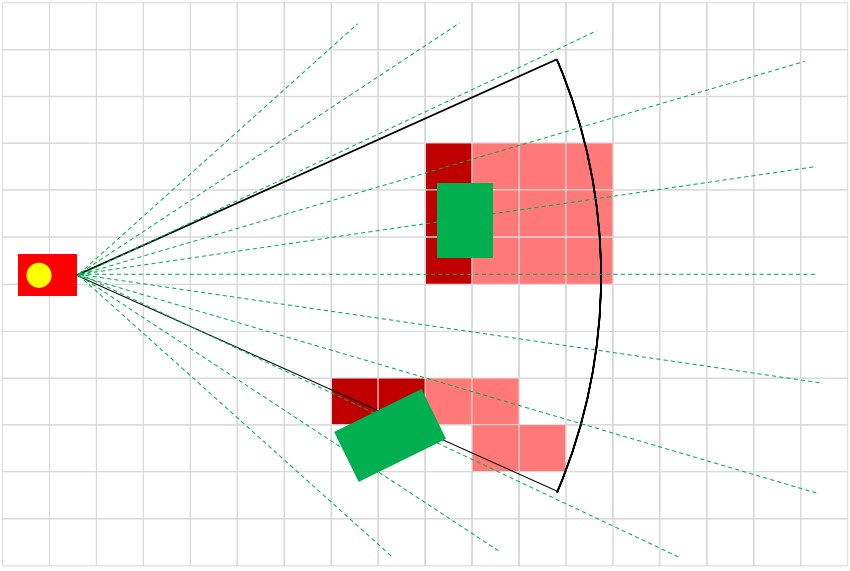


Figure 8. Local field of view map with heading bins. The green blocks are obstacles. Dark red cells are marked "Obstacle" in the map and pink cells are marked "Occluded."

Figure 8 shows an example result. Note that we use the centroid of each cell to determine inclusion in each heading bin or in the field of view. The remaining cells, including all cells external to the field of view, are unchanged. Also, we assume obstacle detection found points on the surfaces facing the robot (or the obstacles would not have cleanly defined edges).

4.3.2 Path Planning

Path planning is done via a search on a global map represented by a 2-D grid of 10 cm × 10 cm cells. Search is performed using the Anytime Dynamic A* algorithm[15] from the current position of the robot to a separately determined waypoint. This algorithm's advantage over the classical D* algorithm is its anytime property, whereby a suboptimal path is found extremely rapidly, and is iteratively improved to the optimal path as processing time allows. To do this, the heuristic is first greatly inflated to allow the algorithm to aggressively discover a suboptimal solution. The subsequent steps to improve the solution do not require the search to begin from scratch, but rather decrease the heuristic and update the search nodes whose costs were invalidated by the heuristic change. This gives us a simple and robust method to deal with large environments with many obstacles by stepping away from the hierarchical architecture of multi-layered search algorithms. The search is constrained to sixteen directions (Figure 9) to further improve path quality, and operates at 40 Hz in moderately cluttered environments. The graph connectivity may be easily doubled or halved to enable 32-directional or 8-directional search to trade-off between path smoothness and search speed.

5. LOCALIZATION AND ACTUATION

5.1 State Estimation

The state estimation system uses a square root central difference Kalman filter (SRCDKF)[16] to fuse sensor data from the compass, GPS, wheel encoders, and gyroscope and maintain an optimal state estimate. The SRCDKF is a sigma point filter that uses a deterministic set of points to represent a multivariate Gaussian distribution over states and measurements. The robot's state is defined as

$$\mathbf{x} = [x\ y\ \theta\ \delta\ \omega\ v_r\ v_l]^T,$$

where (x, y) are vehicle coordinates in a Cartesian local frame, $\theta \in [0, 2\pi)$ is heading, $\delta \in [0, 2\pi)$ is the bias between magnetic compass heading and true GPS heading, ω is the yaw rate of the vehicle, v_r and v_l are the right and left wheel ground speeds [2, 3.2].

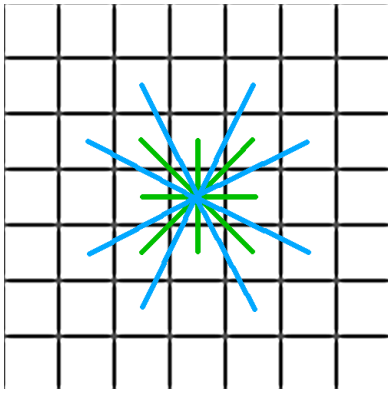


Figure 9. Anytime D* algorithm's 16-directional search generates paths that are continuous sequences of segments; segments may connect two cells in any of the 16 ways shown here.

5.2 Path Tracking and Low-level Control

Phobetor uses a crosstrack controller to execute path following after a path is planned (section 4.3.2) [2, 3.5]. Low-level control is implemented by tuned motor controllers; details for a similar system can be found in [2, 3.6].

6. CONCLUSION

We developed a flexible autonomous system with hardware and software innovations to support robust obstacle and lane detection and local navigation. Through the addition of RANSAC curve fitting to our vision system, we have significantly improved the quality of our lane detection results, and parallelization of our obstacle detection system has improved performance by 3x on the robot's computer. By implementing Anytime A* in our navigation stack, we further optimized our platform's path finding capability. Although the algorithms were developed and tuned for the IGVC, the same principles can be applied to other robotics systems in real-world applications, both civilian and military. In conclusion, we have created a viable robotics platform that applies novel, low-cost, lightweight, solutions to drive our real-time, autonomous platform.

APPENDIX A. ALGORITHMS

A.1 Ground-plane obstacle fitter

```
Data: Color image with depth map
Result: List of obstacle points
best_inlier_set = null; best_model = null; best_score = null;
for max_iterations times do
    // select random points from point cloud;
    p1 = random_cloud_point();
    p2 = random_cloud_point();
    p3 = random_cloud_point();
    // compute distances between them;
    v1 = p2 − p1;
    v2 = p3 − p2;
    // the vectors v1 and v2 define a plane;
    unit_normal = unit_vector( v1 × v2);
    // then we find the shortest distance from plane to the origin;
    distance_to_origin = distance_to_plane(unit_normal, [0 0 0]);
    inlier_set = {};
    foreach p in point_cloud do
        if distance_to_plane(p, model) < inlier_threshold then
            inlier_set.add(p);
        end
        model = least_squares_fit(inlier_points);
        score = score_model(model);
        if score > best_score then
            best_inlier_set = inlier_set;
            best_model = model;
            best_score = score;
        end
    end
end
return best_model, best_inlier_set;
```

Algorithm 1: Pseudocode for RANSAC ground-plane fitting.

A.2 Slope-based obstacle detector

```
Data: Color image with depth map
Result: List of obstacle points
type_A_tiles = create_A_tiles();
type_B_tiles = create_B_tiles();
all_obstacle_points = {};
// Executes each ‘‘iteration’’ of the loop in parallel
parallel
for tile in type_A_tiles do
    new_obstacle_points = check_compatibilities(tile);
end
// Executes each ‘‘iteration’’ of the loop in parallel
parallel
for tile in type_B_tiles do
    new_obstacle_points = check_compatibilities(tile);
end
// Discards isolated obstacle points which are likely false positives
clustered_obstacle_points = cluster(all_obstacle_points);
return clustered_obstacle_points;
```

Algorithm 2: Slope-based, parallelized obstacle detection.

A.3 RANSAC parabola lane fitter

```
Data: Color image with depth map
Result: List of rotated parabolas modeling the lanes in the image
detected_lanes = {};
has_more_lanes = True;
while has_more_lanes do
    best_inlier_set = null;
    best_model = null;
    best_score = null;
    for max_iterations times do
        // select random points from point cloud
        p_1 = random_ground_point();
        p_2 = random_ground_point();
        p_3 = random_ground_point();
        θ = pca_rotation_angle(p_1, p_2, p_3);
        r_1= rotate_point(p_1);
        r_2 = rotate_point(p_2);
        r_3 = rotate_point(p_3);
        model = determine_parabola( r_1, r_2, r_3); // not regression, exactly defined
        inlier_set = {};
        foreach p in ground_points do
            if squared_distance(p, model) <inlier_threshold then
                inlier_set.add(p);
            end
            model = least_squares_parabola(inlier_points);
            score = score_model(model);
            if score > best_score then
                best_inlier_set = inlier_set;
                best_model = model;
                best_score = score;
            end
        end
    end
    if inlier_set_too_small(best_inlier_set) then
        has_more_lanes = False;
    end
    detected_lanes.append(best_model);
end
return detected_lanes;
```

Algorithm 3: Pseudocode for RANSAC lane fitting with parabola model.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Clarence W. Rowley, Princeton University Department of Mechanical & Aerospace Engineering, for his continued support of this project. They would also like to express their gratitude to the Princeton School of Engineering and Applied Sciences, the Keller Center for Innovation in Engineering Education, and the Norman D. Kurtz '58 fund for their gracious donations of resources. Also, they thank Stephanie Landers of the Keller Center and Tara Zigler of the Princeton University Department of Operations Research & Finance Engineering for their tremendous logistical assistance.

REFERENCES

[1] Simmons, R. and James, D., *Inter-Process Communication: A Reference Manual* (August 2001).

[2] Abiola, S. O., Baldassano, C. A., Franken, G. H., Harris, R. J., Hendrick, B. A., Mayer, J. R., Partridge, B. A., Starr, E. W., Tait, A. N., Yu, D. D., and Zhu, T. H., "Argos: Princeton university's entry in the 2009 intelligent ground vehicle competition," *Intelligent Robots and Computer Vision XXVII: Algorithms and Techniques* **7539**(1), 75390N, SPIE (2010).

[3] Manduchi, R., Castano, A., Talukder, A., and Matthies, L., "Obstacle detection and terrain classification for autonomous off-road navigation," *Autonomous Robots* **18**, 81–102 (2004).

[4] Fischler, M. A. and Bolles, R. C., "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM* **24**, 381–395 (June 1981).

[5] Eberly, D., "Least squares fitting of data." `http://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf`.

[6] xu, Z.-y. and zhang, J., "Parallel computation for stereovision obstacle detection of autonomous vehicles using gpu," in [*Life System Modeling and Intelligent Computing*], Li, K., Fei, M., Jia, L., and Irwin, G., eds., *Lecture Notes in Computer Science* **6328**, 176–184, Springer Berlin / Heidelberg (2010). 10.1007/978-3-642-15621-2_21.

[7] "Openmp in visual c++." `http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx`.

[8] "Camera calibration and 3d reconstruction." `http://opencv.willowgarage.com/documentation/cpp/camera_calibration_and_3d_reconstruction.html#cv-findhomography`.

[9] "Numerical recipes in c." `http://www.nrbook.com/a/bookcpdf.php` (1992).

[10] Aly, M., "Real time detection of lane markers in urban streets," in [*2008 IEEE Intelligent Vehicles Symposium*], 7–12, IEEE (June 2008).

[11] Wang, Y., Shen, D., and Teoh, E., "Lane detection using Catmull-Rom spline," in [*IEEE International Conference on Intelligent Vehicles*], 51–57 (1998).

[12] Wang, H., Kearney, J., and Atkinson, K., "Robust and efficient computation of the closest point on a spline curve," in [*Proceedings of the 5th International Conference on Curves and Surfaces*], 397–406 (2002).

[13] Eberly, D., "Least-squares fitting of data with B-spline curves," (2008).

[14] Thrun, S., Burgard, W., and Fox, D., [*Probabilistic Robotics*], MIT Press, Cambridge, MA (2005).

[15] Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., and Thrun, S., "Anytime dynamic a*: An anytime, replanning algorithm," in [*In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS*], (2005).

[16] van der Merwe, R. and Wan, E. A., "Sigma-Point Kalman Filters For Integrated Navigation," in [*Proceedings of the 60th Annual Meeting of The Institute of Navigation (ION)*], 641–654 (2004).