Long Term Autonomy in Office Environments

Wim Meeussen, Eitan Marder-Eppstein, Kevin Watts, Brian Gerkey
Willow Garage, Inc.

***Abstract*— We know that robust autonomy, even in constrained environments, is difficult and time-consuming to build. True autonomy remains the pinnacle of achievement for many in the robotics community, and is a worthy scientific goal. But if our primary concern is not autonomy for its own sake, but rather making robots do useful work, then we should consider what level of human involvement is acceptable for the task at hand. When is it OK for a robot to ask a person for help?**

In this paper, we explore options for improving the robustness of robotic systems by identifying failures and executing recovery behaviors, including asking for help from a human operator. As a motivating example, we undertook the task of allowing a robot to operate successfully in an office environment over long periods of time. The task required the robot to navigate the environment continuously and to recharge itself at a standard outlet when needed. During the longest running test, a 13-day uninterrupted run, the robot continuously navigates to randomly chosen points in the environment, covering a total of 138.9 kilometers. About every 1-2 hours the robot navigates to one of three designated recharging locations, and plugs itself into a standard outlet. Human help, via a web-based teleoperation interface, was required on average every 2-3 days

I. Introduction

A useful robot, like any useful tool, helps the user to do some valuable work. What sets robots apart from other tools is the possibility of *autonomy*: an autonomous robot can do work for the user without the user's direct involvement, other than perhaps initiating the activity. But how much autonomy is needed? When is it OK for a robot to ask for help? In this paper we explore these questions. We argue that robots should ask for help, and posit that in any long-running robot system, they will do so.

From decades of research and development, we know that robust autonomy, even in constrained environments, is difficult and time-consuming to build. True autonomy remains the pinnacle of achievement for many in the robotics community, and is a worthy scientific goal. But if our primary concern is not autonomy for its own sake, but rather making robots do useful work, then we should consider what level of human involvement is acceptable for the task at hand. Is it OK for a person to intervene once a day to help a robot that is otherwise doing important work 24/7? The answer will depend on a variety of factors, from characteristics of the task to determination of costs.

Such practical consideration is always warranted for commercial deployment of robots, where the motivating force for the use of robots in the first place is economic. To minimize cost, one should consider all options. But it is also warranted in research environments; how many hours have graduate students collectively spent tweaking obstacle avoidance systems to iron out the corner cases in navigation, when the research agenda was SLAM, or person-detection, or something else? What would have been lost had the student occasionally used a joystick to help an otherwise autonomous robot out of trouble?

Fig. 1. The PR2 robot continuously operated over a period of 13 days, covering a total of 138.9 km.

To be clear, we do not argue against autonomy, nor against its pursuit. But we do suggest that, in many circumstances, human intervention is an important technique, and one to be considered alongside more conventional engineering approaches to building robust systems.

In this paper, we explore options for improving the robustness of robotic systems by identifying failures and executing recovery behaviors, including asking for help from a human operator. As a motivating example, we undertook the task of allowing a robot to operate successfully in an office environment over long periods of time. The task required the robot to navigate the environment continuously and to recharge itself at a standard outlet when needed.

Although the combined navigation and recharging system never reached 100% autonomy, the robot ultimately ran for 13 days, 2 hours and logged 138.9 kilometers of travel over that period. To achieve this length of run, it was important to develop a system architecture that supported recovery modes and involved both autonomous and human-aided recovery strategies. This paper discusses that architecture, along with the challenges encountered in developing such a system.

II. Related work

There are many real world examples in a variety of fields where long running robots rely on some human input to achieve a high level of robustness. Already in more than 100 US hospitals mobile robots transport anything from meals

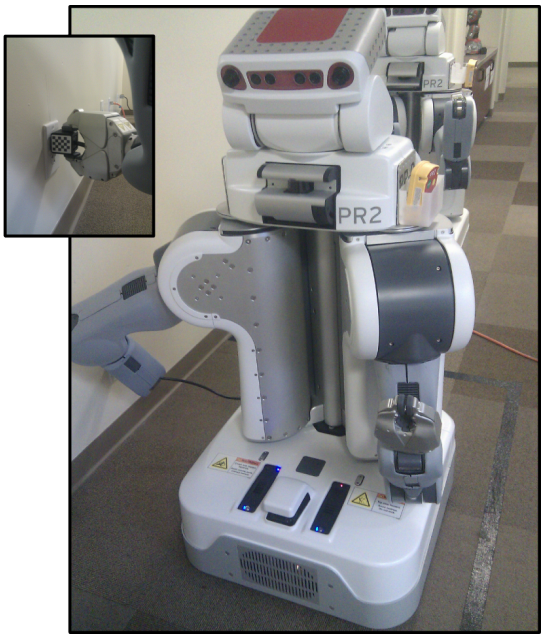

Fig. 2. The PR2 plugs itself into a standard outlet at a designated charging location.

to linens [1]. The robots are disk shaped mobile platforms that autonomously navigate their environment while avoiding obstacles. When a robot gets cornered it calls a help desk, where an operator manually steers around the obstacle or, when nothing else works, asks someone in the hospital to remove an obstacle.

Numerous robots have also been used as robotic tour guides. RHINO [2] was the first robotic tour guide, and was followed by MINERVA [3], Mobots [4], Robox [5], and Jinny [6] among others. These robots all navigated around museum environments with varying degrees of success, citing localization as their primary problem because museum environments are often crowded with people. These robots operated for long periods of time without any direct human help. However, their successful autonomy depended on humans making occasional modifications to the environment. The tour guide robots lacked the sensors to avoid certain types of dynamic obstacles. E.g. tables, chairs, and small objects that did not exist in the robot's a priori maps of the museum. To guarantee their traversed paths were free of such obstacles, the robots relied upon the museum staff to keep areas clear.

In some situations optimal robustness is achieved with a much higher level of human input. In space exploration for example robustness is absolutely critical, shifting the balance form autonomy towards human input. The NASA Mars Rovers [7] demonstrate an unprecedented level of robustness by combining both autonomy and human input. From the other side of the spectrum, devices such as cars, which traditionally are fully operated by humans, are shifting more towards autonomy. By taking away control of their drivers (ABS, traction control, adaptive cruise control [8], etc), cars can achieve a higher level of robustness. Even the long running web servers that power today's internet rely on human intervention to increase their level of robustness. A web server calls, texts or mails the system administrator in case something goes wrong.

III. Task Description

Our goal is provide a robust platform for robots to continuously perform tasks in an office environment. Candidate tasks include mail delivery, cleanup, and security monitoring. We identified two basic capabilities that are critical to enable long term execution of such tasks: (i) the ability for the robot to navigate to different locations in the building, and (ii) the ability for the robot to recharge its batteries. Once a robot possesses these two capabilities, it can keep itself alive over long periods of time by navigating to a recharging location whenever its batteries run low, and plugging itself into an outlet.

Our continuous operation platform uses a simple task queue to allow users to schedule the tasks the robot needs to perform between charging times. In the future, we expect our colleagues to regularly add tasks to the queue, treating the robot as persistently available infrastructure to support testing and data acquisition. For example, a vision researcher working on object recognition might add a task that gathers images of objects on tables. An HRI researcher who has developed a human-aware navigation algorithm might add a task to test the new approach. Our hypothesis is that if we can use the robot(s) as we do compute clusters, submitting jobs and receiving results with minimal maintenance or intervention, then we will collectively do more experimental work, and do it more thoroughly.

In the experiments reported here, our goal was to stress the two core components of this platform (navigation and recharging). So we continuously filled the task queue with random goals throughout our building. The test required the robot to navigate to these random goals until its battery level dropped below a certain threshold, at which point it should proceed to a charging station to plug in. The charging station is a standard US wall outlet, with a sign next to it to inform people that the area around the outlet should remain clear to allow the robot to reach the outlet (See Figure 2). The robot carries its own plug, which is magnetically attached to the robot base. After the batteries are charged beyond a another threshold, the robot should unplug and resume navigation to the random goals.

IV. Approach

To achieve robust recharging and navigation capabilities, we design a system that expects failures of individual components. When a failure is detected, one or more recovery behaviors are executed. For example, when navigating through tight spaces, the global (2 dof) path planner can propose a path that is infeasible for the local (3 dof) path follower. When the path follower repeatedly receives the same infeasible path, it will fail. This failure triggers a set of recovery behaviors, one of which runs a more expensive global path planner that reasons in 3 dof over the whole

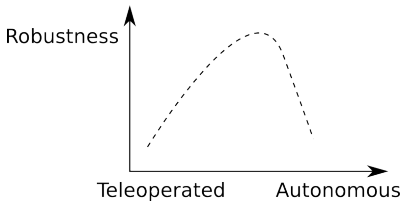

Fig. 3. Maximum robustness is not achieved by full autonomy or by full human control. The 'sweet spot' lies somewhere in between.

environment. Because this 3 dof global planner uses the same model as the local path follower, its plan is guaranteed to be feasible for the follower.

This approach results in a robust system, as long as there is a recovery behavior for each failure situation. However, in an uncontrolled environment that is shared with humans, it is not possible to foresee all possible situations that might occur. We therefore add a last-resort recovery behavior that calls a remote human operator for help.

We argue that maximum robustness is not achieved by a fully autonomous system. Nor is it achieved by giving a human full control of the system; humans make mistakes that an autonomous system would not make. Robustness is maximized by shared control between autonomy and human input.

In our initial experiments, the robot was operating fully autonomous and could not call for human help. In practice, this meant that we spent a lot of time keeping the environment free of "unreasonable" obstacles. In our building, people regularly leave behind obstacles, such as power cables, that are invisible to the robot's sensors, but big enough for the wheels to get stuck on. In the end it proved much more time efficient and effective to not try to control the environment, but to allow the robot to call for help when it got stuck on one of the "unreasonable" obstacles.

The call for help is handled via a remote web-based interface, eliminating the need for physical human presence. The remote interface allows a person to see through any of the robot's 7 onboard cameras, move the pan-tilt head to point cameras at specific targets, and command the mobile base (see Figure 4). The interface does not implement control of the arms, which made some types of failures unrecoverable (see Section VI-C). In our experiments, human help was required on average every 2-3 days. This low rate of interventions would allow a single person to manage dozens of robots from a remote location. Connecting to a service such as Mechanical Turk [9], it would be possible to offload and distribute the human intelligence required to keep the system running. For example, when the robot is unsure if the path ahead is obstacle-free, it could send out a set of images of its surroundings to Mechanical Turk, and have users around the world select obstacles in the images. Obviously, issues of security and privacy would need to be addressed in the implementation of such an approach.

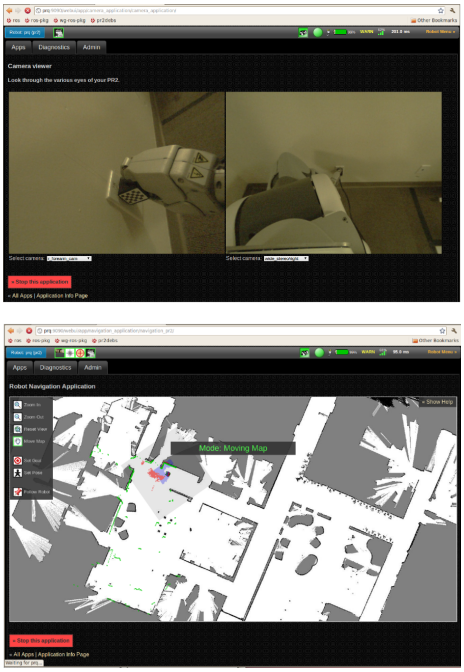

Fig. 4. The web interface allows a person to see through any of the 7 onboard cameras, move the pan-tilt head to point the cameras at specific targets, and command the mobile base.

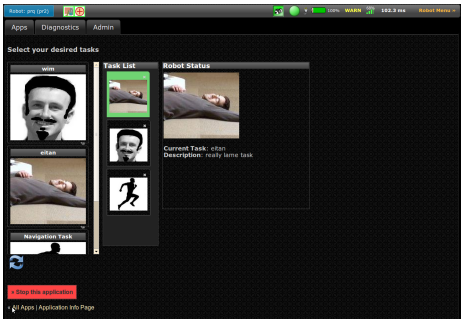

Fig. 5. The task manager web application.

V. System Architecture

There are three main components in the system: task manager, navigation, and recharging. Each component, along with its interaction with the overall system, is described in the sections below.

A. The task manager

The task manager is responsible for determining what tasks to run. It performs basic scheduling and manages robot resources at a high level, guaranteeing that only one task runs at a given time. Users of the task manager request tasks via a web interface that allows them to place tasks into a queue (see Figure 5). The task manager services this queue in order, though it may stop execution of a task at any time to initiate a recharging cycle when it detects that the robot's battery level falls below a threshold.

To facilitate interaction with the task manager, each component in the system wishing to be run as a task exposes

an interface allowing the task manager to start and stop the component. The task manager uses this interface to start and stop the tasks it wishes to run. However, in the case that a task fails to shut down cleanly for a user-specified timeout, the task manager reserves the right to kill a task's processes to force a shutdown.

For our long term autonomy application, the task manager's role is fairly simple. It cycles between navigation and recharging tasks, transitioning between the two based on the robot's battery state.

B. Navigation

The navigation component of the system wraps the ROS navigation stack configured for the PR2 robot [10], and exposes a task interface to allow interaction with the task manager. The ROS navigation stack consists of a 3D costmap, a 2 dof global planner, a 3 dof local planner, and a set of recovery behaviors. The costmap is responsible for aggregating obstacle information for use by the planners. It does so by placing sensor readings into an efficient 3D voxel grid, and then projecting down to 2D for more efficient planning. The 2 dof global planner takes this obstacle map and creates a coarse plan for the robot to follow through the building. However, this plan may not be feasible as the global planner doesn't take the kinematics or dynamics of the robot into account. To smooth and follow the plan, more accurate local planning is performed with a 3 dof planner that takes the shape and dynamics of the robot into account.

The ROS navigation stack is fairly robust, but does, on occasion, fail to create valid plans. In these cases, the navigation stack executes a set of user-specified recovery behaviors to attempt to rescue the robot from situations where it is stuck. Specifically, the navigation stack attempts recovery by clearing out dynamic obstacles seen far away from the robot's current position to clear space for the global planner, performing an in-place rotation allow to attempt to clear out space, and calling a full 3 dof global planner that is expensive to run, but capable of creating plans out of very tight spaces. If all these recovery behaviors fail repeatedly, and the robot's battery level drops below a certain threshold, a human is notified that the robot needs help and teleoperates the robot out of its stuck situation.

C. Recharging

The recharging component allows the PR2 to plug itself into a standard US electrical outlet. The outlet pose is acquired in two steps. First we estimate the wall normal from the stereo cameras in the head, using a projected pattern to add texture to the wall. Then, the remaining 4 degrees of freedom of the outlet are estimated from the monocular camera in the forearm. The forearm images are taken with the camera right in front of the outlet. On its base, the PR2 carries a plug that is modified with a checkerboard pattern to enable easy visual pose estimation. The full 6 dof pose of the plug is acquired from the forearm camera when the gripper is holding the plug.

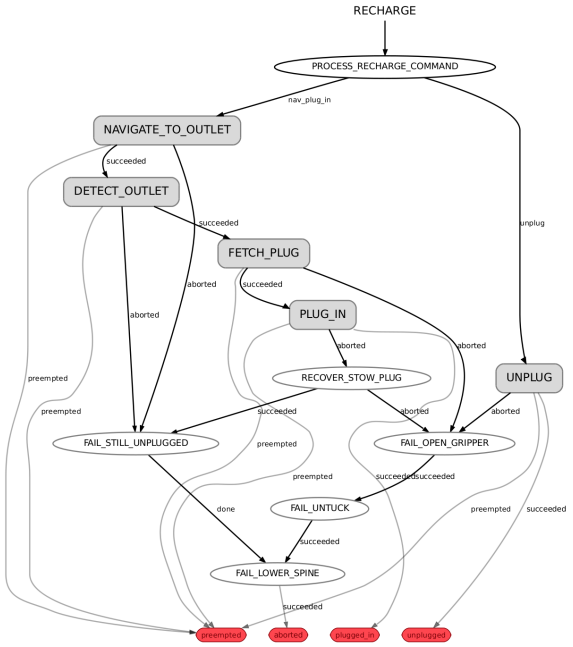

Fig. 6. A hierarchical state machine with built-in recovery behaviors controls the plugging in task.

Based on the estimated poses of the outlet and the plug, the insertion task is executed. Because the estimated poses have a fixed bias, every robot requires a specific x-y calibration offset in the plane of the wall. This calibration allows a robot to plug into any outlet in the building, using any plug. The autonomous plugging in capability is described in detail in [11].

VI. Lessons Learned

In this section we discuss the practical lessons we learned in building a robust robotic system. First we discuss the techniques used to increase the robustness of the system, at the level of the software engineering, the autonomy and the environment. Then we talk about the debugging tools we used and developed to uncover failure cases of the system. Finally the unrecoverable failures are discussed.

A. Increasing reliability

1) Software Engineering:

- **More processes and less state.** Individual computer processes will crash at some point in time. A crash is problematic if the process contains relevant state of the task the robot is executing. It is therefore beneficial to split up large, complex processes that carry state, into multiple smaller processes, where only some of the smaller processes carry state. This approach has two advantages: (i) the process that carries state becomes less complex and therefore less likely to crash, and (ii) the processes without state don't cause problems when they crash as they simply get restarted.
- **Store state externally.** If a statefull process crashes, it can recover successfully when it stored its state outside its own memory space. For example, our localization

process stores the current robot pose to an external parameter server every 2 seconds. If the localization process crashes, it can read its last valid state from the parameter server when it gets restarted, and resume its task.

- **Limited uptime.** The longer a process stays up and running, the higher the chance that something fails. Memory leaks, internal deadlocks, invalid state, and other hidden flaws are more likely to present themselves given more time. We therefore aim to only keep as few processes as possible running all the time. Our architecture based on an task manager that spawns and kills processes on demand makes it possible to not keep any of the navigation or plugging in related processes running for more than one navigate-recharge cycle.

2) Autonomy:

- **Redundancy.** Providing multiple algorithms for the same task makes it possible to overcome the weaknesses of each individual algorithm. For example, for navigation we provide both a fast 2 dof circle-based path planner and a slow 3 dof path planner that takes the correct robot footprint into account. When the 2 dof planner fails, we temporarily switch to the 3 dof planner instead.
- **Allow more time.** A trivial but effective strategy is to start the recharging task when the battery level is still relatively high. This allows the recharging task to fail and retry multiple times, and still plug in before the battery runs out.
- **Human in-the-loop.** Our 'solution' for autonomy is to replace some components with humans. This does not necessarily mean that either the autonomy or the human has control of the robot. A long running system can consist of some autonomous components and some human in-the-loop components. For example, one of the obstacle detectors could be a human who manually marks obstacles in the image, while the rest of the system is still autonomous.

3) Environment: Although we can't fully control the environment, it is still possible to build redundancy into the environment. For example, provide different paths through the building to reach a single location, or allow the robot to recharge at multiple locations. When the robot fails to plug in at one charging location due to obstacles or bad lighting, it can recover by choosing a different charging location.

B. Debugging tools

As the reliability of a system increases, the mean time between failures gets longer and longer. This poses a practical problem, as the time required to uncover the remaining failure cases in the system grows without bounds. Moreover it is often required to observe a single failure case multiple times to uncover the underlying problem. To keep increasing reliability it is therefore required to hit failure cases faster. This can be achieved by running the system more often or over a longer period of time.

1) More often: Some components of the system are only exercised every couple of hours or every few days. It therefore takes a long time to find failures associated with such components. Configuring the system to continuously run just these components helps accelerate the life-cycle by orders of magnitude. We used this technique for the plugging in task, which takes less than 2 minutes, while a full run-recharge cycle takes about 3 hours. Making the robot plug itself in continuously exercises the plugging in component almost 100 times faster, and quickly uncovers many of its problems. The same strategy was also applied to the Linux kernel. Every couple of days we experienced a kernel panic under some rare high-load conditions. By hammering the kernel continuously using artificial extreme load scenarios, we were able reproduce the panics in a matter of minutes. Moreover, the test setup allowed us to test multiple kernel versions and configurations in parallel.

2) Longer: An obvious strategy to find failures faster is to run the system all the time, even at night and during weekends. This approach however raised some unexpected problems: when the system fails while unattended, it is unable to recharge itself, and it slowly runs out of battery power. When the batteries run out, the onboard computers power down, and all system state that is relevant to find out what went wrong is lost.

Initially we tried to continuously record all relevant system state to a hard disk, but the amount of data recorded was huge, and the recording itself created new system failures because of load and disk space issues. The strategy that proved most successful only records the system state when an unrecoverable failure was detected. Because the system would not recover anyway, it is also possible to actively use the system actuators to gather more information about the state. For example, the robot arms would move around and record images of the robot's surroundings with the forearm-mounted cameras. These images show the state of the environment (obstacles, lighting conditions at night, etc.) when the robot dies. The images can also be added to training sets for vision-based detectors.

As detecting failures is part of the challenge, the system does not always correctly detect (and record state about) failures. Therefore an extremely low battery level triggers a human notification: the robot sends a text message to a set of pre-programmed cellphone numbers, giving us the opportunity to log into the robot and observe its state before the batteries run out.

C. Unrecoverable failures

There are a variety of failures that are unrecoverable, such as kernel panics, hardware failures, electronics failures, and battery problems. For most of these cases it seems unlikely to come up with a realistic recovery strategy. There are other failures however that are currently unrecoverable, but that have an easier path towards a recovery strategy. In general, this type of failure requires manipulation capabilities to recover. For example when the plug that the robot carries around on its base gets detached and starts dragging on

Fig. 7. An unrecoverable failure: the plug got detached from the base and drags on the floor behind the robot.

the floor, the robot currently can't recover (see Figure 7). Even human-in-the-loop recovery is not possible because the remote teleoperation interface does not allow an operator to move the robot arms. While autonomous recovery from this type of situation would be possible, it would be a challenging task to implement. A alternative approach to cope with this and similar failures is to improve remote teleoperation capabilities, to allow a human to use the robot arms to recover the plug and put it back on the base. Therefore the next step towards long term operation could very well be enabled by improved teleoperation.

VII. Conclusion

In order for many robotic systems to be useful, it is important for them to be reliable and robust. Traditionally, robustness is improved by attempting to improve the autonomy of a system. However, there are cases where it is extremely difficult, if not impossible, to achieve 100% task reliability, especially in dynamic environments. To achieve high levels of reliability we feel it is more important to focus on recovery from failure rather than accounting for every possible edge case a system might encounter. Furthermore, it is often possible to recover from failure cases autonomously, but there will likely be scenarios for which these recovery behaviors fail. In these cases, it makes sense to seek human help to recover from failure. As long as this help is required infrequently, the system is still extremely effective.

Shifting focus from task-level robustness to system-level robustness and relying on effective recovery behaviors allowed a PR2 robot to cover 138.9 kilometers over 13 days in an unmodified office environment. The individual tasks of navigation and recharging, while fairly robust, were both incapable of achieving such a milestone on their own. Runs typically lasted only 2 to 3 days. Allowing remote human intervention increased this to 13 days. Ultimately, the robot's run ended because it encountered a situation where the teleoperation interface for human-aided recovery was too limited. Specifically, there was no way for a remote operator to pick up a plug that had fallen off the robot's base.

From our experiences with a long-running system, we argue that achieving long term operation in the near term will require not only robust autonomy, but also effective remote teleoperation capabilities. Fully autonomous systems to complete complex tasks in unmodified environments are difficult to architect. Perhaps semi-supervised robotic systems, focusing on effective recovery from failure, will help to make robotic applications feasible that would otherwise be out of reach.

The Open Source ROS software used in this paper is available online at [12].

References

[1] http://www.aethon.com.

[2] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schimdt, "Map learning and high-speed navigation in RHINO," in *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonasso, and R. Murphy, Eds. Cambridge, MA: MIT/AAAI Press, 1997.

[3] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hhnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, "Minerva: A second-generation museum tour-guide robot," in *In Proceedings of IEEE International Conference on Robotics and Automation (ICRA99)*, 1999.

[4] I. R. Nourbakhsh, C. Kunz, and T. Willeke, "The mobot museum robot installations: A five year experiment," in *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2003, pp. 3636–3641.

[5] R. Siegwart, "Robox at Expo.02: A Large Scale Installation of Personal Robots," *Special issue on Socially Interactive Robots, Robotics and Autonomous Systems*, no. 42, pp. 203–222, 2003.

[6] G. Kim, W. Chung, K. rock Kim, and M. Kim, "The Autonomous Tour-Guide Robot Jinny," in *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, pp. 3450–3455.

[7] M. Maurette, "Mars rover autonomous navigation," *Auton. Robots*, vol. 14, no. 2-3, pp. 199–208, 2003.

[8] Z. Bareket, P. S. Fancher, H. Peng, K. Lee, and C. A. Assaf, "Methodology for assessing adaptive cruise control behavior," *IEEE Transactions on Intelligent Transportation Systems*, vol. 4, 2003.

[9] http://www.mturk.com.

[10] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *ICRA*, 2010, pp. 300–307.

[11] W. Meeussen, M. Wise, S. Glaser, S. Chitta, C. McGann, P. Mihelich, E. Marder-Eppstein, M. Muja, V. Eruhimov, and T. Foote, "Autonomous door opening and plugging in with a personal robot," in *ICRA*, 2010.

[12] https://kforge.ros.org/project/contops.