Where Am I? A Simulated GPS Sensor for Outdoor Robotic Applications

Benjamin Balaguer and Stefano Carpin

School of Engineering
University of California Merced
5200 North Lake Road, Merced CA 95343, USA,
{bbalaguer,scarpin}@ucmerced.edu

Abstract. Advances in the field of robotic simulations in general and the complexity of virtual outdoor environments in particular have created a demand for accurate simulated open-air localization devices. In this paper, we answer this request by presenting the implementation of a simulated Global Positioning System receiver for the popular USARSim platform. The engineering tradeoff of speed versus accuracy is encountered throughout the design process and discussed comprehensively in the paper. Along the lines of a validation methodology we developed in recent years, the simulated sensor is implemented and extensively analyzed in a real/simulated scenario, where data logged from a real robot is evaluated against the data acquired in simulation.

1 Introduction

With the continuously growing focus on multi-robot cooperation and improvements in computer hardware, algorithmic techniques, and computer animation, robotic simulators are gaining momentum within the robotics community. Indeed, robotic simulators are now capable of simulating multiple blocks of an outdoor urban environment, comprised of a multitude of robots, victims, fires, collapsed structures, rivers, bridges, and more [1]. The typical assortment of sensors for robots operating in similar real world environments more and more often includes a Global Positioning System (GPS) receiver in order to ease the localization task. Henceforth, in order to create a faithful simulation environment, we designed a simulated GPS sensor that is, to the best of our knowledge, the first of its kind in comparable simulation systems. In fact, our goal is not to merely convert Cartesian coordinates into latitude and longitude components, but rather produce a realistic sensor that exhibits the same properties of current GPS receivers.

Even though the paper's aim is to provide a standard methodology for the construction of a simulated GPS sensor for outdoor robotic applications, we implemented our framework inside the Unified System for Automation and Robot

Simulation (USARSim)[1]. USARSim has become a popular real-time three dimensional simulator thanks to widespread validation effort from the community [2][3][4] and its utilization in the yearly RoboCup Rescue Simulation League [5][6]. USARSim is an open-source extension to the Unreal Tournament (UT) game engine that, consequently, only requires a modestly priced UT license. The game engine takes care of rendering scenes and computing physics while UnrealScript, an object-oriented programming language similar to C++, allows for the addition and modification of actors (e.g. robots, actuators, sensors) in the simulation. Since the sensor's implementation will exclusively be performed in UnrealScript, it is important to keep some of its drawbacks in mind; namely its slow computational speed and lack of floating point number precision.

This paper builds upon a validation methodology we developed for USARSim in the past and that has proved to be highly effective in order to close the loop between simulation and reality. In short, our approach consists in performing the same experiment in simulation and with the real world system, and to quantitatively compare the results. This effort may sometimes be costly, because it entails developing accurate models of the robotic systems at hand, but it has proved to be a formidable tool in order to assess which conclusions can be extrapolated from simulation to reality and which ones do not generalize. Part of the USARSim success draws from the abundance of these validation efforts, and this paper, besides illustrating the technical details of GPS simulation, can be read as a working example of the robot simulators validation process we advocate.

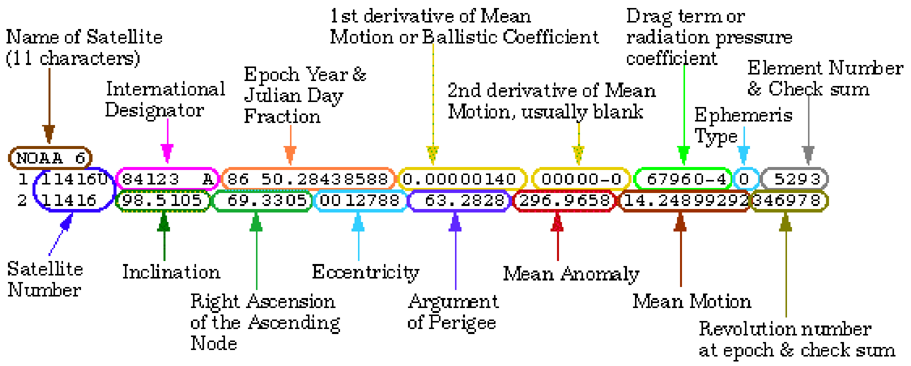
2 Methodology and Implementation

2.1 Satellite Tracking

Since a real GPS module receives signals from satellites, the first cornerstone to simulate a GPS receiver is to establish a relationship between the GPS sensor and the orbiting satellites. The most realistic method to establish this relationship is by tracking GPS satellites. The three governing North American aerospace institutions, namely the National Aeronautics and Space Administration (NASA), the North American Aerospace Defense Command (NORAD), and the Air Force Space Command (AFSPC), collectively promote the usage of the Simplified General Perturbations Satellite Orbit Model 4 (SGP4) for satellite tracking, the details of which are found in [7]. In fact, the SGP4 algorithm has gained a strong reputation among amateurs and professionals and quickly became the standard satellite tracking model, resulting in sustained research and constant improvements [8][9].

While the details of SGP4, which can be found in the referenced publications, are beyond the scope of this paper, we will provide a very brief and high-level

[1] This system was formerly known as Urban Search and Rescue Simulator. The name change reflects the much broader applicability it gained through the years.



Name of Satellite (11 characters)
International Designator
Epoch Year & Julian Day Fraction
1st derivative of Mean Motion or Ballistic Coefficient
2nd derivative of Mean Motion, usually blank
Drag term or radiation pressure coefficient
Ephemeris Type
Element Number & Check sum
NOAA 6
1 11416U 84123 A 86 50.28438588 0.00000140 00000-0 67960-4 0 5293
2 11416 98.5105 69.3305 0012788 63.2828 296.9658 14.24899292346978
Satellite Number
Inclination
Right Ascension of the Ascending Node
Eccentricity
Argument of Perigee
Mean Anomaly
Mean Motion
Revolution number at epoch & check sum

Fig. 1. Sample TLE File with Format Descriptions (modified from [10])

description of the orbital model. The algorithm takes as input a NASA/NORAD Two-Line Element (TLE) [10], a date, and a time and outputs the location of the satellite defined by the TLE at the given date and time. Real-time satellite tracking can consequently be achieved by continuously running SGP4 with the current date and time. The TLE format, a sample of which is given in Figure 1, provides the orbital information necessary to reconstruct the orbit of a satellite (see Figure 2) that can then be used to approximate the satellite's location. The SGP4 algorithm parses the TLE data and calculates the satellite's orbital state vectors, the result of which is usually expressed as latitude, longitude, and altitude components. Manifestly, the precision of SGP4 is strongly correlated to the accuracy of the TLE data. To that effect, the AFSPC publishes and maintains a database of TLE files available to approved users on the space track website [11].

Even though there already exists a plethora of satellite trackers implemented with SGP4, available both as online visualization tools [12][13] and comprehensive software suites [14][15][16], we constrained our search to user-friendly, reusable, open-source, and lightweight systems since we were, originally, looking to port the code to UnrealScript. After experimenting with a multitude of implementations and choosing GLSat [17], we realized that translating the code to UnrealScript was impossible due to poor floating point precision. Indeed, the UT engine is only capable of keeping track of seven decimal places, rendering live satellite tracking inaccurate. In addition, the amount of time spent calculating satellite positions took too much time away from other simulated components.

We circumvented the aforementioned problem by switching to an offline approach for satellite tracking. More specifically, we modified the GLSat program to take in the following parameters: a TLE file comprised of one or more satellite, a date, a time, and the amount of time during which satellite positions should be tracked. The program then computes all the satellite positions, starting from

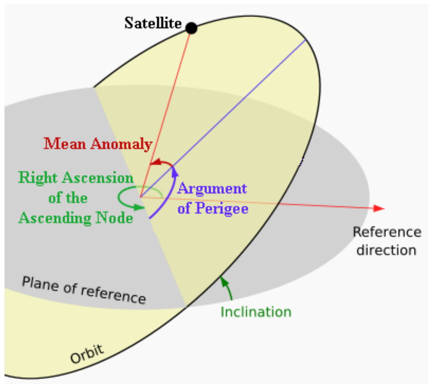


Fig. 2. Reconstructing a Satellite Orbit from a TLE-File

the given date and time, over the time interval with a pre-defined time step, and creates a configuration file that can be read by UnrealScript. The configuration file is then read-in and stored in appropriate data structures when the simulation starts, effectively yielding a lookup table of satellite positions, based on time. Even though this method requires greater user interaction and does not reflect real satellite positions for large time steps, it alleviates computational burden - a primordial aspect of robotic simulations. It is worthwhile noting that this approach allows flexible researchers to use any configuration file (for applications where it does not matter if satellite positions do not reflect the current date and time), while more stringent researchers can go through the process of generating their own configuration files (for applications requiring the proper satellite positions for a given time interval). The reader should observe that in order to render faithful simulation, the system we have developed does not perform a generic simulation of a GPS, but rather offers the possibility to locate the simulation in time, i.e. it extracts the position of the GPS satellites at a given point in time. In this way the correlation between simulation and real world systems becomes more binding.

2.2 Signal and Noise Model

With the simulation now capable of determining the satellites' location, we introduce signal and noise models. Real GPS receiver precision depends on the number of satellite signals received by the unit (i.e. the more satellites seen by the receiver, the better the accuracy of the location) as well as the geometric arrangement of the satellites (i.e. the dilution of precision) [18]. For the purpose of this paper and due to the additional computational burden of integrating dilution of precision calculations, we solely base our noise model on the number of satellite signals received by the GPS receiver. Consequently, the first step in our model is to determine the number of satellites observed by the simulated

GPS sensor; a two-step process. First, the angle of elevation between the current sensor location and each satellite position is calculated. Any satellite yielding an elevation angle less than five degrees is discarded. The process eliminates all the satellites that would require sending a signal through the earth's surface (i.e. negative elevations) as well as the satellites that are too low to consider (i.e. elevations between zero and five degrees). Eliminating satellites based on their elevation, through a set of straightforward equations, is of foremost importance to guarantee that the next elimination process is computationally friendly. Second, ray tracing is performed from the GPS receiver to each of the satellites' location to further eliminate some of the observed satellites. If the ray trace hits an environment entity (e.g. buildings, vehicles) on its way to a satellite then that satellite is discarded, otherwise it becomes one of the satellites seen by the GPS sensor. Ray tracing being a computational burden [19], the first elimination process assures that it will not be used needlessly. The proposed line-of-sight signal model provides a very crude approximation, but modeling realistic signal strength taking into account possible deflections would surely result in an intractable scenario.

As previously discussed, the amount of noise in a GPS sensor measurement needs to be proportional to the number of satellites available. We, once again, favor a computationally-friendly modular approach that allows researchers to effortlessly change the noise function as they see fit. Indeed, the currently implemented noise function, described below, can be swapped by another; thus allowing improvements or specific noise functions emulating different GPS receivers. Since experiments have shown GPS noise to have Gaussian distribution [18], we exploit the Box-Muller method [20] to generate Gaussian-distributed random numbers. More specifically, we use two configuration variables to dictate the maximum and minimum amount of localization error, in meters, when four and twelve satellites are available to the sensor, respectively. The reader shoud note that four is the minimum number of satellites required for a GPS fix, while twelve is habitually the maximum. The two configuration variables are especially important for users looking to effortlessly simulate different GPS receivers without having to change code or recompile the simulation environment. For example, both Wide Area Augmentation System (WAAS) capable and WAAS-incapable GPS receivers can be simulated by simply modifying the parameters.

Mathematically, the two configuration variables give the sensor two points on a curve with respect to the number of satellites observed. Our noise function linearly interpolates between those two points to create a function of error, in slope-intercept form, based on the number of satellites seen. First, the slope, m, of the linear function is calculated, using the two configuration variables maxNoise and minNoise as shown:

$$m = \left(\frac{\text{maxNoise} - \text{minNoise}}{4 - 12} \right). \tag{1}$$

Then, the y-intercept, b, of the linear function is derived using the slope:

$$b = \text{maxNoise} - (4 * m). \tag{2}$$

Finally, Equation 3 calculates a standard deviation, σ, for the Gaussian number generator by using the current number of satellite seen by the simulated GPS sensor. More specifically, the Gaussian random number generator is used with a mean of zero and a standard deviation of a third of the maximum noise, guaranteeing that, in 99.7% of the cases, the error produced will be within plus or minus of the maximum error.

$$\sigma = \left(\frac{m * \text{SatelliteSeen} + b}{3} \right) \tag{3}$$

Even though the proposed method might seem simplistic, it is efficient to compute and provides very good results, as will be described in the experimental section of the paper.

2.3 Implementation Details

The satellite tracker, along with the signal and noise models, encompasses the majority of the simulated GPS methodology but leaves a few open issues. One of the main dilemmas when using a GPS sensor in virtual environments is the mapping of a virtual location to a real one. Since many virtual worlds do not inherently possess latitude and longitude coordinates, we propose and have implemented three different methods to allocate a GPS coordinate to a virtual world, each with different levels of precedence. First, world developers can add a specially-created tag when building the virtual environment and modify its properties to reflect the desired latitude and longitude coordinates. The placement of the tag will define a reference GPS coordinate that can be used to determine the latitude and longitude of any point on the map. Second, configuration variables can be set inside the USARSim initialization file to provide the reference GPS coordinate of the (0,0) Cartesian coordinate in the virtual world. Third, the GPS sensor class can be modified to link the (0,0) Cartesian coordinate with a GPS coordinate. All of these methods solve the same problem and have been added for user-friendliness and backward compatibility with old virtual environments.

Once the amount of noise, in meters, has been established using the aforementioned techniques, latitudinal and longitudinal components have to be calculated and returned by the sensor - calculations that require a couple of assumptions. The first, aimed at lowering computationally intensive instructions, assumes a flat earth and, consequently, provides a straightforward translation between distance and degrees, using the surface distance per degree change conversion. In other words, under the flat-earth assumption, a one degree change corresponds to a specific change in distance, allowing conversions from meters to degrees. The second assumption involves the global coordinate frame of the virtual world. We assume that all X-axis motion is converted to latitude and that all Y-axis motion is converted to longitude. While, in most cases, the global X-axis points to the North, it is worthwhile noting that this is not always the case due to singularities that may occur. Indeed, as shown in Figure 3, the sensor handles singularities that occur at the earth's poles (at 90 degree North and 90 degree South) and on the longitude (at 180 degree West and 180 degree East). In other words, and as an example, driving along the X-axis at 89 degrees and 59.9 minutes will increase the latitude component of the GPS until 90 degree North is reached. At that point, the latitude component will decrease (meaning that the global X-Axis now points to the south). These singularities exist and are taken into account due to the flat nature of virtual worlds and the spherical shape of the earth.

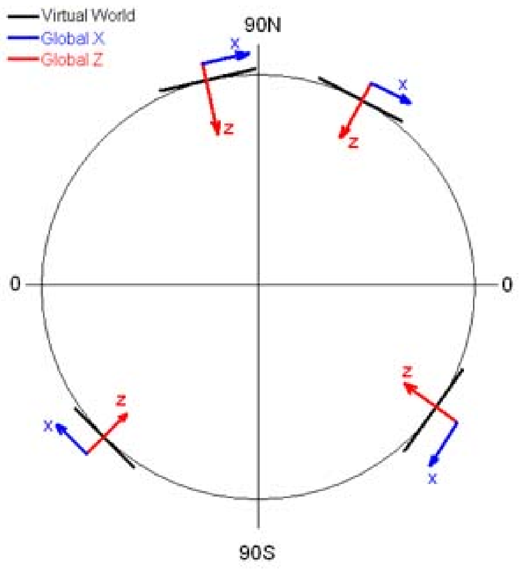


Fig. 3. Singularity Representation with Flat-Earth Assumption.

3 Experimental Results

We conducted a set of experiments aimed at validating the simulated GPS sensor by using a real/virtual testbed similar to [21]. More specifically, we teleoperate a real P3AT robot, equipped with a Holux M1000 GPS receiver, at various distances from buildings outside the University of California, Merced quad. During each run, lasting between two and ten minutes, data comprised of the latitude, longitude, and number of satellites observed by the receiver is logged to a file. To facilitate the correspondence between the real and simulated robot motions, we limited our experiments to straight paths, thus decreasing time-dependent mechanical differences between the robots. Once the runs were performed using the real robot, they were replicated in simulation, in real-time, with the extensively-utilized USARSim P3AT, equipped with the simulated GPS sensor. A simplified virtual representation of the UC Merced quad was built to that effect, only including major landmarks such as buildings and significant ground slopes. Some of the experimental results are shown in Figure 5, 6, and 7.

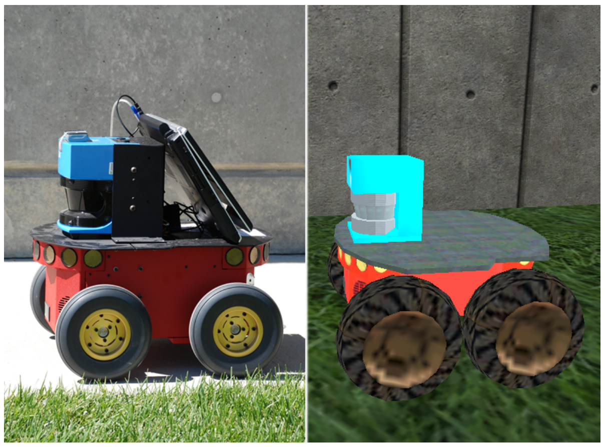


Fig. 4. Picture of the real P3AT (left) next to a screenshot of the simulated P3AT (right) in the experimental environment.

As can be seen from Figure 5, the latitude and longitude components reported by the simulated sensor are very close to those reported by the real receiver. The difference between the two paths stem from the noise model parameters used during the experiment. Indeed, the simulated run used a maximum noise parameter of three meters (the advertised accuracy of the Holux M1000 GPS receiver with WAAS enabled), but the real receiver actually produced errors, in our experiments, of up to eight meters. It is worthwhile mentioning that Run 3 is particularly off at the beginning of the experiment due to a cold start from the receiver; a typical GPS feature (i.e. they take time to initially localize) that was not modeled in our simulated GPS.

A plot of the error difference, in degrees, between the real and simulated GPS coordinates is given in Figure 6 for each of the three runs presented in this paper. Equation 4, exploited for each time step, gives insight into how the plots were created. The primary insightful result from Figure 6 is the fact that, for each of the three runs presented, the error between the simulated GPS sensor and the Holux M1000 GPS receiver did not surpass 0.00014 degrees, a testament to the accuracy of the simulated sensor. Furthermore, the error for Run 1 and

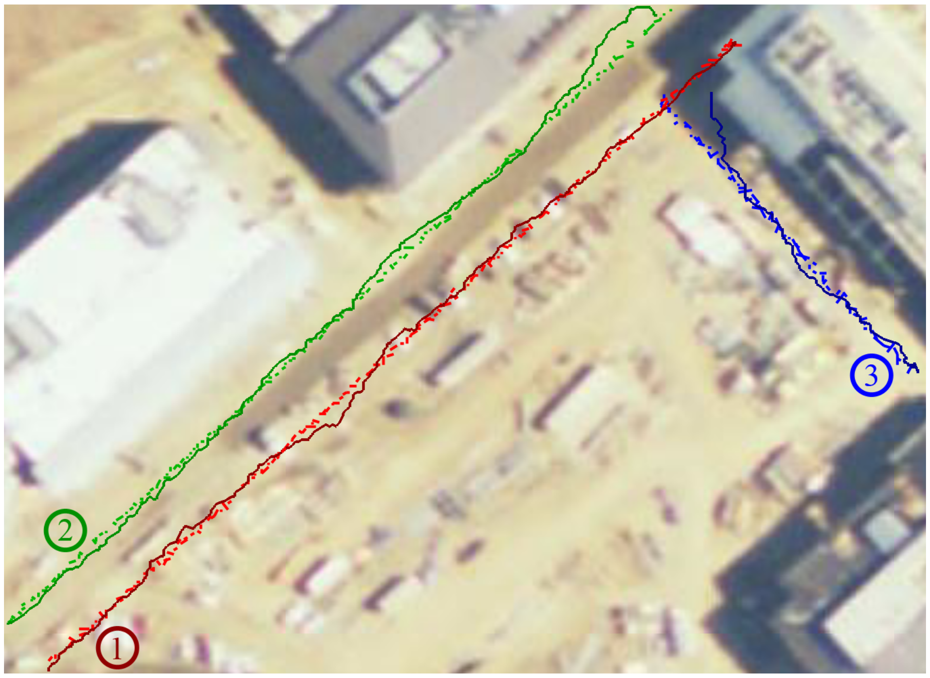


Fig. 5. Plot of GPS latitude and longitude coordinates in Google Earth for three different runs. Each run is labeled from 1 to 3 and comprised of two paths. The solid line indicates the path of the real robot whereas the dotted line shows the path of the simulated robot. Run 1 was performed from North-East to South-West, Run 2 was performed from South-West to North-East, and Run 3 was performed from North-West to South-East.

Run 2 varies the most at the beginning and towards the end of the runs. This behavior is explained by the inertial difference between the real robot and the simulated one. Indeed, the real robot requires a lot more time to reach a given speed than its simulated counterpart. The same behavior is observed at the end of Run 3. The beginning of Run 3 possesses an unusually high error due to the previously-discussed cold start.

$$\sqrt{\left(\mathrm{Real}_{\mathrm{Latitude}} - \mathrm{Sim}_{\mathrm{Latitude}}\right)^2 + \left(\mathrm{Real}_{\mathrm{Longitude}} - \mathrm{Sim}_{\mathrm{Longitude}}\right)^2} \quad (4)$$

Figure 7 shows, for each run, the number of satellites seen by the real and simulated GPS receivers. The significant aspect of the data is that the simulated plot follows, in terms of shape, the real plot. Two additional comments can be made. First, the simulated sensor sees, in most cases, more satellites than its real counterpart. Second, the simulated sensor is much more linear and experiences fewer changes in the number of satellites seen. Both of these facts can be

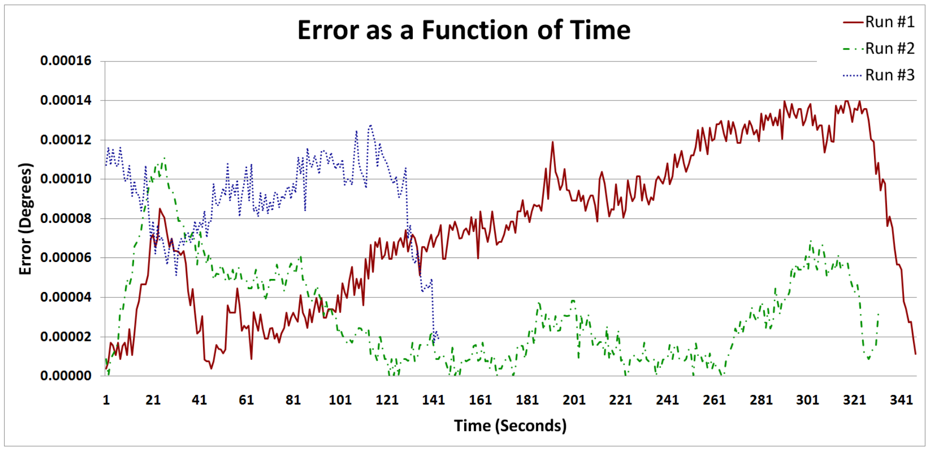


Fig. 6. Plot of the error, in degrees, between the real and simulated GPS coordinates for each run presented in Figure 5.

explained by the simplified virtual representation of the environment and the signal model. Since only major landmarks were included in the virtual world, small objects (e.g. trees, benches, stairs) interfered with the real receiver but not the simulated one. Additionally, the line-of-sight signal model does not take into account signal reflections, i.e. the reason for the oscillating number of satellites seen by the real receiver.

4 Conclusion and Future Work

In this paper, we presented a complete methodology supporting the creation of a simulated GPS sensor, supplemented by a USARSim implementation and experimental results comparing real and simulated data. The simulated results are close to the real receiver, especially when taking into account the assumptions made and the focus on computation friendliness over accuracy. In fact, the USARSim GPS sensor was selected and used in the 2008 RoboCup Rescue Simulation League in July 2008, where its robustness was successfully put to the test in a highly competitive scenario. Moreover, a DARPA Urban Challenge team has expressed the desire to use the sensor along with USARSim.

A few research opportunities stem from this work both in terms of improvements and extensions. Some improvements can be made within the noise and signal models, provided that they are not too demanding for the engine. More specifically, a great improvement would be to incorporate the dilution of precision as part of the noise model. In addition, getting real-time satellite tracking working, perhaps through the use of C++ dynamic library, would create a better all-in-one solution. Alternatively, porting the implementation to a different simulator, such as the new Microsoft Robotics Studio Simulator, could allow for more rigorous noise and signal models and the integration of real-time satellite tracking.

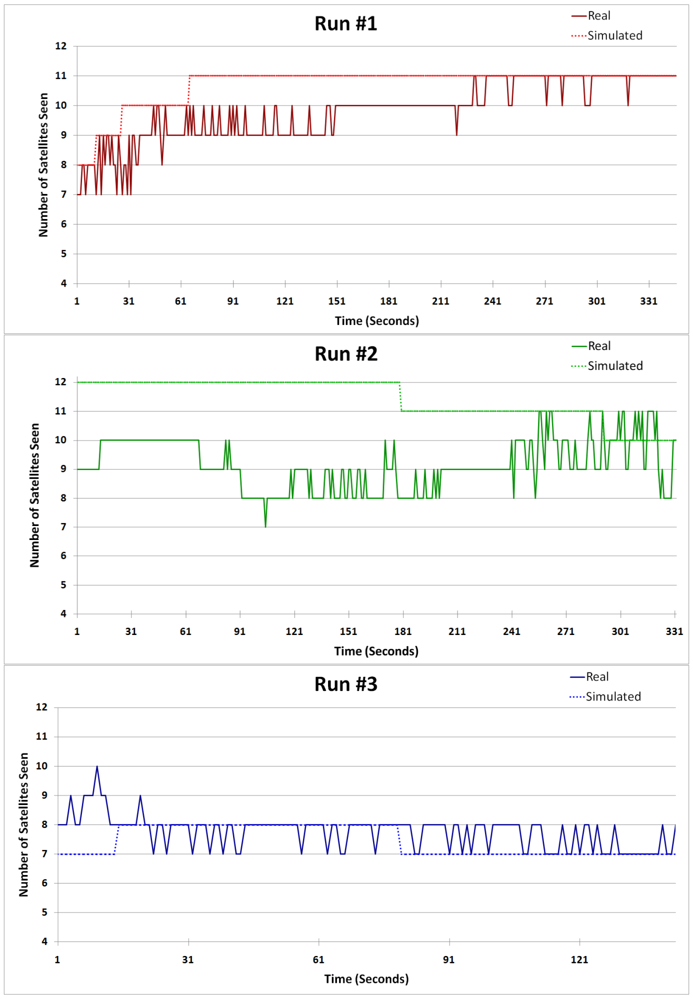


Fig. 7. Plot of the number of satellites seen as a function of time for each run. The runs correspond to the ones presented in Figure 5.

References

1. Balakirsky, S., Scrapper, C., Balaguer, B., Farinelli, A., Carpin, S.: Virtual Robots: progresses and outlook. SRMED (2007)
2. Carpin, S., Stoyanov, T., Nevatia, Y., Lewis, M., Wang, J.: Quantitative assessments of USARSim accuracy. Proceedings of PerMIS (2006)
3. Wang, J., Lewis, M., Hughes, S., Koes, M., Carpin, S.: Validating USARSim for use in HRI Research. Proceedings of the Human Factors and Ergonomics Society 49th Annual Meeting. (2005) 457–461
4. Pepper, C., Balakirsky, S., Scrapper, C.: Robot Simulation Physics Validation. Proceedings of PerMIS (2007)
5. Balakirsky, S., Lewis, M., Carpin, S.: The Virtual Robots Competition: vision and short term roadmap. SRMED (2006)
6. Carpin, S., Wang, J., Lewis, M., Birk, A., Jacoff, A.: High fidelity tools for rescue robotics: Results and perspectives. Robocup Symposium (2005)
7. Hoots, F., Roehrich, R.: SPACETRACK REPORT NO. 3 - Models for Propagation of NORAD Element Sets. Project Spacetrack Reports, Peterson (1988)
8. Vallado, D., Crawford, P., Hujsak, R., Kelso, T.: Revisiting Spacetrack Report #3. AIAA/AAS Astrodynamics Specialist Conference (2006)
9. Kelso, T.: Validation of SGP4 and IS-GPS-200D Against GPS Precision Ephemerides. AAS/AIAA Space Flight Mechanics Conference (2007)
10. Definition of Two-Line Element Set Coordinate System, http://spaceflight.nasa.gov/realdata/sightings/SSapplications/Post/JavaSSOP/SSOP_Help/tle_def.html
11. Space Track The Source for Space Surveillance Data, http://www.space-track.org/
12. NASA Science@NASA J-Track 3D, http://science.nasa.gov/realtime/jtrack/3d/JTrack3D.html
13. Real Time Satellite Tracking, http://www.n2yo.com/?k=20
14. Magliacane, J,: Portable PREDICT Plus! A Satellite Tracking, Pacsat Yakking, APRS Hacking, Linux Packing Mini Application Suite You Can Carry with You. CQ-VHF Magazine (2005)
15. Owen, M., Knickerbocker, C.: Nova for Windows, Real Time Tracking of an Unlimited Number of Satellites. Northern Lights Software Associates (2006)
16. Stoff, S.: Orbitron - Satellite Tracking System. Quick Starting Guide (2005)
17. Haupt, M.: Applicability of OSS to Space Thermal Engineering Open Source Software for Engineering Purposes. European Workshop on Thermal and ECLS Software (2003)
18. Kaplan, E., Hegarty, C.: Understanding GPS: Principles and Applications. Artech House Publishers, Norwood (2005)
19. Shirley, P., Morley, K.: Realistic Ray Tracing. AK Peters, Wellesley (2003)
20. Box, G., Muller, M.: A Note on the Generation of Random Normal Deviates. The Annals of Mathematical Statistics. **29** (1958) 610–611.
21. Balaguer, B., Carpin, S., Balakirsky, S.: Towards Quantitative Comparisons of Robot Algorithms: Experiences with SLAM in Simulation and Real World Systems. IROS 2007 Workshop (2007)