Interactive Visualizations for Testing Physics Engines in Robotics

Johan Fabry
PLEIAD and RyCh labs,
Computer Science Department (DCC),
University of Chile

Stephen Sinclair
Inria Chile

***Abstract*—Physics engines in robotics simulators should yield a simulation that is physically faithful to the real world. However, simple scenarios like dropping a ball on the floor already reveal that this is not so. There is hence a need to be able to test such engines in real world scenarios, to see where they are lacking. To help to quickly and efficiently develop unit tests for real-world behavior we developed a tool we call Live Tests for Robotics. In this tool paper we show how its interactive visualizations allow for the efficient construction of such unit tests.**

I. Introduction

Physics engines are an important element of the toolchain of robotic simulation software. Well-known examples are the SimSpark simulation used in the Robocup Simulation league[1] and the Open Dynamics Engine (ODE) physics engine for the DARPA Virtual Robotics Challenge [1]. Noted conspicuously in the DARPA challenge rules [2] is that physics engines should be physically faithful to real world behaviour. Yet empirical evidence shows that this is not always the case. For example, we found that using the current version of the Gazebo robotics simulator [3], if we drop a ball its behaviour is radically different depending on the physics engine used. The same setup will, with ODE, cause the ball to bounce when it hits the floor, while with Bullet[2] the ball does not bounce at all! Clearly, at least one of the two engine behaves incorrectly.

Hence, what is required are unit tests of real world situations to establish overall correct behaviours, and it should be possible to carry out these tests on several physics engines. To make such tests easy to write, we developed a methodology and toolset called Live Tests for Robotics (LT4R). In this tool paper we show how, in LT4R, a small set of interactive visualizations integrated in a live programming language enable the effective construction of unit tests of physics simulators for robotics.

II. Background and Related Work

The role of simulation in robotics has been acknowledged widely, notably in the form of the Virtual Robotics Challenge issued by DARPA in 2013, for which the Gazebo simulator using ODE was selected as the competition physics engine [1]. Tasks included controlling a walking robot, having it sit in a car and drive, and grasping and manipulating a fire hose. However, for 'in-simu' development to be transferable to real-world robotics, a simulation engine must of course be physically faithful to real world behaviour [2]. For example, in the Open Source Robotics Foundation's 2014 online survey on Gazebo, it was similarly found that "physics validation" was the highest-voted topic [4].

Despite many advances in the area, differences between various engines can lead to differing behaviours such as those we see in the bouncing ball example mentioned above. This being said, there is often an expected behaviour that one can validate by 'eye-balling' the results and an often-used method of testing is comparing the Root Mean Square error of an object along a motion path. However, one problem with the motion path error approach is that due to the integration process, small differences early on may lead to an accumulation of error, skewing the evaluation when one is more interested in whether the overall behaviour is correct—i.e., did the hand grasp the object or did the car steer towards the goal.

There are some examples of the approach of verifying the overall behavior in previous literature. For example, in their comparison of 5 simulation engines, Erez et al. proposed using short-time motion path error, in itself a unique and interesting idea, but ultimately for their grasping task, measured simply whether the object was successfully grasped for the duration of the simulation [5]. In another example, Gowal et al. evaluated a vehicle simulator based on a simple measure of how far the ending position was from the desired position, ignoring details of the full motion path [6]. Lastly, Castillo et al. suggested taking advantage of the aforementioned error accumulation by only looking at the final position of actors [7].

III. Live Tests for Robotics

As mentioned before, for transferability, a physics simulation must be realistic, yet in the current state of the art it is easy to find non-ideal behaviour, e.g., the simple bouncing ball. There is therefore a need for tests that can establish to what degree a given physics simulator is adequate to the simulation at hand. Moreover, testing infrastructure should provide a means to verify the overall behaviour of the simulation with regard to the real world.

In this section we present our solution: Live Tests for Robotics (LT4R). LT4R allows the rapid design of minimal unit tests using a state-based model of expected behaviour.

[1] http://wiki.robocup.org/wiki/Soccer_Simulation_League
[2] http://www.bulletphysics.org

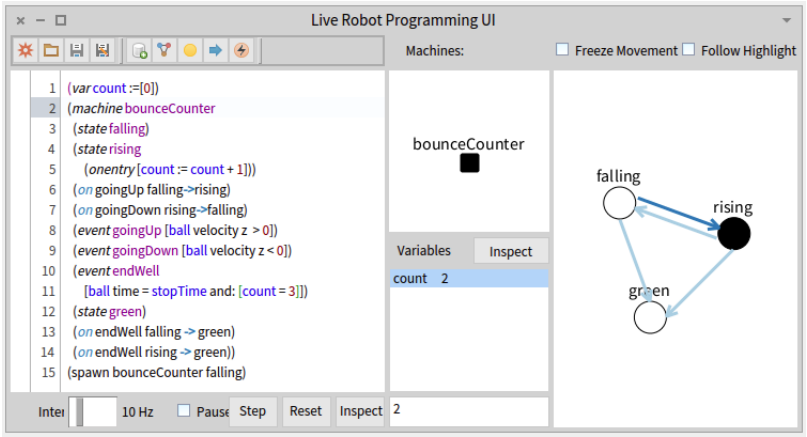


Fig. 1: The UI of LRP showing the program of Section IV-B1. On the left is the code editor, on the right the running state machine. The machine is in the rising state and the last transition taken is marked in dark blue. The middle top half shows the hierarchy of machines, the bottom half shows a list of variables and their values, which can be set using the bottom textfield.

A. The Design of LT4R

Firstly, in LT4R the physics engines are tested off-line: we use Gazebo non-interactively to run the simulation and produce a log of motion paths of all the simulated objects as a set of *trajectories*. The first advantage of this approach is that these trajectories can then be consumed by several of our tests, only requiring Gazebo to be run once for the execution of all these tests. Second, generation of such logs can be fully automated which then enables all tests to be automated.

As a second part of LT4R, we use state machines to encode the world state, i.e., "what is happening now." Such an encoding allows to define different aspects of the required behaviour as different state machines: one for each test. Furthermore, it is arguably straightforward to do since there is a natural mapping of the state of the world to a state in the machine.

Lastly, the goal of LT4R is to achieve maximum interactivity in building and modifying the state machines, which is why we use Live Robot Programming (LRP) using LRP, the state machine is directly and interactively constructed. This makes the development cycle minimal because a change to a machine can be immediately and interactively tried out on the trajectory under test. Moreover, LRP itself can be tailored to the task at hand (as will be discussed in more detail in Section III-B).

More in detail, LRP is a programming language for the specification of behaviours as nested state machines [8], [9]. The language follows the paradigm of live programming [10], allowing for the direct construction, visualization and manipulation of the program's run-time state. LRP is not tied to a specific robot API: it currently supports programming behaviours in ROS [11], the Lego Mindstorms EV3[3], and the Parrot AR.Drone[4]. Moreover, LRP is not fundamentally restricted to the development of robot behaviours: any type of behaviour that can be specified as a nested state machine can be programmed in LRP. The only requirement for interoperation of LRP with other systems is the availability of an API for Pharo Smalltalk, as LRP is implemented in Pharo. To access such an API, LRP embeds Smalltalk in itself as *action blocks*: blocks of Smalltalk code that can be triggers for events, variable initializers or code that runs according to the active state of the machine. Figure 1 shows the user interface and visualization of LRP. Note the visualization of the running machine, which is always in sync with code edits. A full introduction to LRP is outside of the scope of this paper. Instead we refer to the published literature [8], [9] as well as the LRP website: http://pleiad.cl/LRP for videos.

B. The LT4R Implementation

For LRP to interoperate with external systems, an API must be available for that system. In our case, we need to have some API that allows for the LRP state machines to reason over the recorded trajectories of a physics simulation. The LT4R implementation is exactly this, and we discuss it here.

The conceptual model of the API is a replay of the trajectories: Action blocks of LRP only have access to the physical properties of the objects at the current point in time, called a *snapshot*. The running of a test then consists of replaying the trajectory snapshot by snapshot, where at each step in time the state machine can react appropriately to that snapshot.

LT4R exposes the object trajectories as global variables, in effect reifying the Gazebo object as a global variable in LRP. This global variable has two methods: pose and velocity, respectively for the object's pose[5] and velocity (in the current snapshot). Each of these is a six-dimensional vector with methods x, y, z and rx, ry, rz that give the scalar values for the linear and angular components, respectively. Thus, for example, to get the linear z velocity of the bouncing ball example, if the object is called ball in Gazebo the expression in the action block would be: ball velocity z.

[3]https://education.lego.com/mindstorms

[4]http://developer.parrot.com/

[5]Pose is robotics terminology for position and rotation in 3D space.

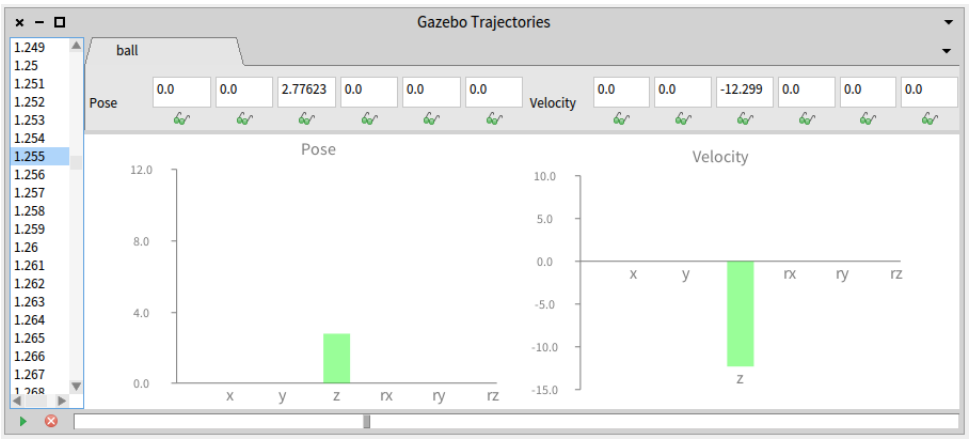


Fig. 2: The trajectory UI showing a snapshot of the bouncing ball trajectory. The left lists all snapshots in order, the right shows pose and velocity graphs for the current snapshot. Each component of these vectors can be plotted over time (by clicking on the green glasses). At the bottom are start and stop controls as well as a slider for trajectory playback.

All objects also have the time method, which returns the simulation time of the current snapshot, and there are two global variables: startTime and stopTime that return the time of the first and the last snapshot, respectively.

Crucially, LT4R provides a custom UI for the exploration of object trajectories and control of playback, shown in Figure 2. The current snapshot can then be manually picked from the list and playback controls allow for moving through simulation time. For the selected snapshot, for each object trajectory a tab shows pose and velocity vectors as bar charts, together with the exact values of their scalar components. Each component can be plotted over time for the entire simulation run, examples of which are shown in Figures 3, 4a and 4b. Notably, in these plots hovering over the line produces a tooltip-like popup that reveals the exact data of the point being hovered over by the mouse pointer.

Together with the machine visualization, this UI effectively allows for the interactive and live construction of the state machines that encode the test. This development experience is achieved by letting the user interactively experiment with the passing of time, e.g., by using the slider to scrub through all snapshots. He or she then sees the effects on the state machine, and can change this state machine on the fly when needed.

IV. Writing Unit Tests in LT4R

A. Sliding cylinder

As a first simple test we consider a cylinder that slides down an inclined plane. We found that in Gazebo when using ODE, the cylinder slides down the plane in a straight line, and remains standing on the inclined surface. In contrast, while using Bullet, the cylinder tips over and quickly starts tumbling end-over-end in the x, y and z axes. This tumbling can easily be seen in the visualization of the test data, for example in Figure 3b where the rotational velocity on the y axis is plotted over time. Contrast this with the log data of Gazebo for this experiment: 2030 twelve-dimensional datapoints for the cylinder (plus the same for the box). Given the amount of data, it is not possible to quickly ascertain what effect the tumbling has on pose and velocity without the visualization.

For the sake of the example, let us suppose that the ODE behaviour is accurate. Verifying correctness of behaviour can then be encoded in a simple test, by asserting that the rotational velocity of the cylinder remains 0. The code is as follows:

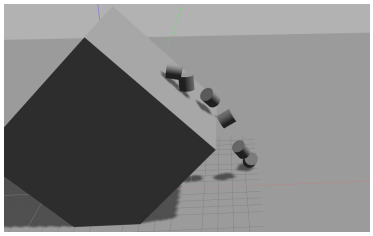
```
1  (machine slide
2    (state green)  (state red)
3    (on tumbling green -> red)
4    (event tumbling [
5          cylinder1 velocity ry  > 0.0001 ]))
6  (spawn slide green)
```

By convention, if a test ends in a state named green we consider the test passed, if it ends in any other state, we consider the test failed. If a test reaches a state named red during any moment of its execution it is an immediate fail. The code above thus defines both a red and a green state (in line 2) and declares (in line 3) that when the cylinder tumbles the machine goes to the red state. Line 6 starts the machine in the green state. Hence if at the end of the replay no tumbling occurred, the test passes, and if the cylinder starts tumbling the test immediately fails. Lines 4 and 5 declare what it means for the cylinder to tumble: the rotational velocity of y (e.g., as shown in Figure 3b) is bigger than (nearly) zero.

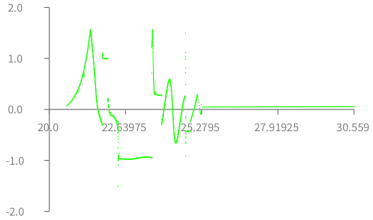
The use of a minimum of 0.001 is required here because without it the ODE tests would also fail—the y rotational velocity does not remain exactly at 0. This is revealed by the plot of the data of ODE, where hovering over the datapoints reveals their value to be slightly less than 0.0001.

B. Bouncing ball

Returning to the bouncing ball example, we now present two different tests that can be made on the same trajectory of a dropped ball. This is to illustrate the different features of our solution as well as to show the construction of different unit tests.



(a) Stop-motion of the Gazebo simulator showing the cylinder rotating around the y-axis as it tumbles down a plane.



(b) LT4R plot of the rotational velocity of the cylinder around the y-axis over time.

Fig. 3: The tumbling cylinder when using Bullet.

1) Three Bounces: Using Bullet, dropping the ball does not cause it to bounce, conversely in ODE it bounces three times, which is revealed by the visualization shown in Figure 4a. Note that the underlying data are 3604 twelve-dimensional datapoints, which again makes this real-world property of an object in the simulation difficult to ascertain without the appropriate visualization.

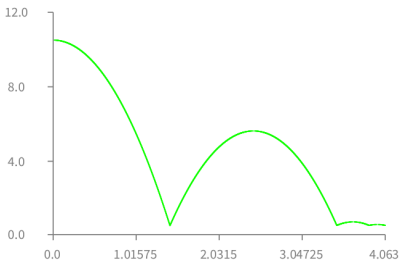
Given the physical properties of the ball, it should bounce a non-zero number of times. Also, under the same starting conditions this number should always be the same. Let us therefore encode this in a test, supposing that the ball should bounce exactly three time.

```
1  (var count :=[0])
2  (machine bounceCounter
3    (state falling)
4    (state rising
5        (onentry [count := count + 1]))
6    (on goingUp falling->rising)
7    (on goingDown rising->falling)
8    (event goingUp [ball velocity z  > 0])
9    (event goingDown [ball velocity z < 0])
10   (event endWell
11       [ball time = stopTime and: [count = 3]])
12   (state green)
13   (on endWell falling -> green)
14   (on endWell rising -> green))
15 (spawn bounceCounter falling)
```

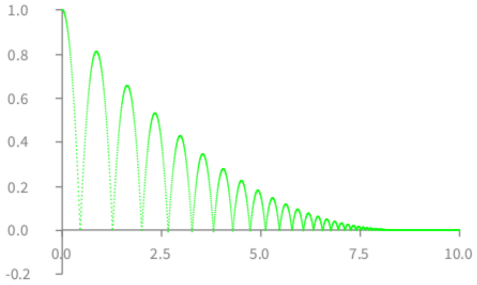
This machine has states for the ball falling (line 3), rising (lines 4 and 5) and transitions between the two states (lines 6 and 7), as well as the events (lines 8 and 9) that use the z velocity of the ball to trigger these transitions. The machine keeps a counter in a count variable (line 1), initialized to 0. To increase the variable on each bounce, line 5 contains an on-entry action to the rising state. (In LRP, states can define actions that are executed atomically when entering a state, when leaving a state, and when being in a state.)

Line 10 defines the event that causes the machine to go to green (line 12), which can happen irrespective of whether the ball is falling (line 13) or rising (line 14). The code of the event checks if the time of this snapshot is equal than the last time recorded (stopTime). If this is the case, we are at the last snapshot of the simulation and should therefore decide if the number of bounces is correct.

This test is constructed step by step, using different features of the UI. First the machine is built for rising and falling (lines 3-9 without line 5). By moving the slider of the trajectory UI (of Figure 2), the user then moves time forwards and back, observing the z velocity of the ball in the trajectory UI and confirming that the visualization of the machine shows it is in the corresponding state. Then the counting logic is added (lines 1 and 5), and again the user can verify that this logic is correct by moving in time and seeing the variable count increase in the variable view. The user can reset the variable to 0 in this view, and then perform a playback of the entire trajectory to see that the number of bounces is 3. Adding lines 10 to 14 then completes the test code, and the user can run the test interactively to verify correctness. Note that, once in the green state, the machine cannot transition to any other state. The user can however pick a state in the machine visualization and, through a context menu, force the machine into that state, such that he or she can keep experimenting.



(a) The z position of the ball over time, in ODE.



(b) The z position of the ball over time, in Siconos.

Fig. 4: Ball bouncing behavior depending on the physics engine, as plotted by LT4R

2) Bounce Height Descends Sensibly: In order to develop a more refined test for bouncing ball behaviour, we defer to an example from the Siconos non-smooth dynamical system simulator[6], developed by the BeBop group at INRIA, since we knew it to contain an accurate simulation of the bouncing ball as compared to the closed-form solution [12]. An image of an example trajectory from the Siconos simulation can be seen in Figure 4b.

Figure 4 allows to see at a glance the behaviour of ODE and the behavior of Siconos. Looking at the visualization of ODE the user can immediately see that there is an issue: the third bounce is unexpectedly low. Conversely, the visualization of Siconos shows a more realistic descent of bounce height for each sufficient bounce. Note that from the raw data it is nearly impossible to see this difference in behavior, again illustrating how a simple visualization is an efficient tool in this setting.

For a test of this behavior, we wish to verify that energy loss is consistent between bounces, and thus the ratio of the top of any bounce to the previous should remain the same within some comparison interval.

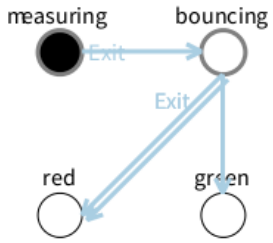


Fig. 5: The bounceLower machine when measuring. Note the thicker border on measuring and bouncing, indicating the presence of a nested machine.

To save space we will not reproduce the code of this machine here. All that is needed is a way to determine what the correct ratio is between each bounce, and then check it at each bounce. To do this, the bounceLower machine, shown in Figure 5 contains two nested machines, each a variant of the bounceCounter machine we have seen previously. The first machine measures the ratio as follows: it records the initial height of the ball in a variable top and the lowest height in a bottom variable when entering the rising state. The second time it enters the falling state it records ratio := (ball pose z - bottom) / (top - bottom). This machine then immediately ends and a transition is taken to to the measuring machine. In this machine, each time the falling state is entered, the ratio between the current height and the last highest point is checked for correctness. As a result the test will failing if bounce height descends in a non-sensible fashion.

This test is constructed interactively by the user, first making the bounceLower machine and nesting the two bouncing ball machines. The measuring machine is then refined, adding the code for measuring the rate, and this is tested interactively by moving the slider forward until the measured value of ratio becomes non-null. This can be seen in the variable view of the UI. The user can check if the computed ratio makes sense, i.e., the specified computation is correct. A transition guarded by a non-null test of ratio is then added, making the machine immediately transition to the bouncing machine. This machine can then be refined, first adding the testing logic and then moving the slider forward to where the ball starts falling. There, the testing logic is run and the user can verify if the behaviour is correct. Note that the user can also manually change the ratio value to force a different outcome of the test, and furthermore force the state machine in a specific state to keep on experimenting with the test implementation.

[6]http://siconos.gforge.inria.fr

V. Conclusion

In this paper we presented the interactive visualizations of LT4R, a tool we created for the writing of unit tests for physics engines in robotics simulators. We have discussed three different unit tests and talked about how the visualizations aid in the rapid construction of these tests. The goal for LT4R is to enable the construction of suites of such unit tests, which will aid in the development and fine-tuning of physics engines.

Acknowledgment

We would like to thank Sebastian Maass for identifying the tumbling/sliding cylinder scenario while testing Gazebo's physics engines at Inria Chile.

References

[1] J. M. Hsu and S. C. Peters, "Extending open dynamics engine for the DARPA virtual robotics challenge," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, Eds. Springer, 2014, vol. 8810, pp. 37–48.

[2] DARPA, "DARPA Robotics Challenge: Virtual robotics challenge rules," Mar. 2013, dISTAR Case 21064.

[3] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Conf. Intelligent Robots and Systems*, Sendai, Japan, Sep 2004, pp. 2149–2154.

[4] O. S. R. Foundation, "Gazebo blog: Gazebo survey results," http://gazebosim.org/blog/survey_2014, 2014, online. Accessed: 27-05-2016.

[5] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX," in *Conf. Robotics and Automation (ICRA)*. IEEE, 2015, pp. 4397–4404.

[6] S. Gowal, Y. Zhang, and A. Martinoli, "A realistic simulator for the design and evaluation of intelligent vehicles," in *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*. IEEE, 2010, pp. 1039–1044.

[7] P. Castillo-Pizarro, T. V. Arredondo, and M. Torres-Torriti, "Introductory survey to open-source mobile robot simulation software," in *Latin American Robotics Symposium and Intelligent Robotic Meeting (LARS)*. IEEE, 2010, pp. 150–155.

[8] J. Fabry and M. Campusano, "Live robot programming," in *Advances in Artificial Intelligence – IBERAMIA 2014*, ser. LNCS, A. Bazzan and K. Pichara, Eds., no. 8864. Springer-Verlag, 2014, pp. 445–456.

[9] M. Campusano and J. Fabry, "Live robot programming: The language, its implementation, and robot API independence," *Science of Computer Programming*, 2016, to appear.

[10] S. Tanimoto, "VIVA: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, 1990.

[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, no. 3.2, 2009, p. 5.

[12] B. Brogliato and V. Acary, "Numerical methods for nonsmooth dynamical systems," *Lecture Notes in Applied and Computational Mechanics*, vol. 35, 2008.