The Journal of the Astronautical Sciences (2020) 67:1575–1599
https://doi.org/10.1007/s40295-020-00216-1

ORIGINAL ARTICLE

Closed-Loop Software Architecture for Spacecraft Optical Navigation and Control Development

Thibaud Teil[1] · **Samuel Bateman**[2] · **Hanspeter Schaub**[1]

Published online: 9 June 2020

Abstract

A software architecture is discussed to develop, run, and test novel autonomous visual spacecraft navigation and control methods in a realistic simulation. This architecture harnesses two main components: a high-fidelity, faster-than-real-time, astrodynamics simulation framework; and a sister software package to dynamically visualize the simulation environment. Maneuvers such as fly-bys and orbit insertions occur over short periods of time and must occur autonomously. Yet, there are no open-source software packages that provide fully coupled spacecraft environments and Flight Software (FSW) enabling Optical Navigation (OpNav) mission scenarios. The presented tool consists of the *Basilisk** astrodynamics framework interfacing with a *Unity*-based visualization *Vizard* that provides a synthetic image stream of a camera sensor. This modular and extensible setup allows optical guidance, navigation and control (GNC) algorithms to be run in a closed-loop format purely in software. The optical measurements are generated in the visualization and passed to the simulation, allowing for real-time control and decision making. This *Vizard* software has the ability to import shape-models, planet maps, and move into an instrument point-of-view. Paired with open-source image processing libraries, these combined components provide all the necessary pieces to fully simulate autonomous, closed-loop, OpNav scenarios in a faster-than-real-time configuration. This allows for progress in the autonomy sector, as full-fledged FSW can be tested in a real flight environment. Furthermore, this enables more realistic and extensive testing of the software, which in turn increases reliability of the GNC methods as they are refined. This paper presents the *Basilisk* and *Vizard* interface architecture, its performance, and develops a example scenario. The image processing methods are displayed and the visualization scenes are validated for pointing purposes, which in turns allows to develop an autonomous pointing algorithm developed in this software environment.

https://bitbucket.org/avslab/basilisk

✉ Thibaud Teil
thte9300@colorado.edu

Extended author information available on the last page of the article.

Keywords Astrodynamics · Simulation · Optical navigation

Introduction

In recent years, autonomy has been established as a key technology enabler for future space exploration [1]. Reducing the frequency of ground-in-the-loop communication allows for less expensive mission support systems. Aside from lightening the load on ground-based tracking, autonomous guidance, navigation, and control can back-up ground-in-the-loop navigation if communication failures occur or when maneuver time and spacecraft distance make it impossible. Whether it be for robotic exploration of the solar system, manned spaceflight, or small satellite development, autonomy opens the door to new mission concepts [2–4].

One key enabler for autonomy is on-board optical navigation, as it provides measurements that can be gathered without contacting Earth. Furthermore, it provides direct information on what is often the subject of the mission's scientific objectives. This paper outlines a novel framework which seeks to combine navigation algorithms within a simulated spacecraft environment. These algorithms require a reliable and extensible test-bed to be developed and refined. This simulation test-bed must provide realistic spacecraft simulations, model the local space environment, and create three-dimensional visualizations of both the spacecraft and the environment. As the guidance and control development typically involves extensive Monte-Carlo sensitivity analysis, computational speed is of paramount importance.

High-fidelity dynamics simulations provide a vital test environment for spacecraft and robotics development. Existing tools such as DARTS [5] paired with DSENDS, Dshell, or ROAMS[1] provide high-fidelity dynamics and visualization capabilities [6]. These tools are are used in a closed software environment that are not generally extensible by researchers outside of the Jet Propulsion Laboratory. Furthermore, although DARTS provides closed-loop dynamics and control, it does not permit the use of visualization snapshots for image processing and OpNav. AGI-EOIR[2] is an STK-based visualization tool that uses physics based radiometric sensor and target image simulation. This software can provide highly accurate sensor images, but these are exported to file and not integrated into a closed-loop simulation. In the field of robotics, ROS [7, 8] and its sister software package *Gazebo*[3] are open source and provide hardware-in-the-loop capabilities. Yet, theses are tailored for ground-robotics applications and do not provide sufficient spacecraft models and features. An open and extensible software solution like ROS has not existed for the spacecraft community in the past.

OpNav simulations have focused either on the image processing component [9], on the estimation component [10, 11], or on using mission data [12]. These provide valuable insight on many facets of the problem; yet no common open-source software

[1] https://dshell.jpl.nasa.gov
[2] https://www.agi.com/EOIR
[3] https://gazebosim.org/

package exists that provides modularity and repeatability while bringing together contributions from many developers. Furthermore, current simulations do not couple spacecraft dynamics and control into OpNav measurements [13]. Camera models are linked to the image processing and filter performances [14], but this does not loop back to the spacecraft control algorithms.

One example scenario is an orbit insertion maneuver, which occurs in close proximity to the body of interest during a short time span (often too short for ground intervention) and is central to mission success. A spacecraft's on-board use of optical measurements can provide assurance of proper maneuver execution, notably if faults occur. Another example is the *New Horizons* Pluto fly-by. The mission extensively studied the likelihood of having Pluto in the image frame [15], whereas autonomous pointing could have provided more confidence by centering Pluto in the image frame [16]. Both these examples showcase the potential for more autonomy in the chain between OpNav, attitude control, and trajectory modifications.

Basilisk [18, 19] is a highly modular astrodynamics simulation framework that allows for the rapid simulation of complex spacecraft dynamics. Key features include solar radiation pressure [20–22], imbalanced reaction wheels [23], imbalanced control moment gyroscopes [24], flexible solar panels [25], fuel slosh [26, 27], depletable mass [28], as well as multiple body gravity and gravitational spherical harmonics. The sensor simulation and actuator components couple with the spacecraft dynamics through a publish-subscribe (pub-sub) messaging system. [19] A state engine allows for complex spacecraft dynamics to be setup without having to develop and code any dynamics differential equations. [29] An associated visualization is built using the *Unity* gaming engine and is called *Vizard*. [17]. Here the *Basilisk* simulation messages are streamed directly to the visualization to illustrate the spacecraft simulation and environment states. Figure 1 shows a Mars Orbit Insertion (MOI) performed in *Basilisk* and visualized inside the *Vizard* software. As ROS and *Gazebo* do for the

Fig. 1 Mars Orbit Insertion Scenario with the astrodynamics simulated inside *Basilisk* and visualized using *Vizard* [17]

robotics community, combined *Basilisk* and *Vizard* provide an open and extensible software architecture to both simulate and visualize spacecraft dynamics and control scenarios.

This papers explores a new software architecture where *Vizard* is not just used to visualize the *Basilisk* simulation states, but becomes itself a visual sensor module for *Basilisk*, thus allowing for closed-loop visual control simulations to be performed. This allows for visual guidance and control algorithms to be tested in a faster-than-realtime software platform that is also suitable for Monte-Carlo type sensitivity studies. The created visualization images are controlled and shared via a new two-way connection between *Basilisk* and *Vizard*. This is a challenge as it requires frame synchronization such that any type of camera resolution can be simulated while maintaining synchronization with the dynamics simulation. Furthermore, it is desirable to design a flexible communication setup between two software packages such that they can be run on a single or multiple computers.

This new *Basilisk*-*Vizard* software integration has the ability to support many scenarios at the cutting-edge of autonomy; these include optical deep space navigation, formation flying, close proximity and servicing applications, as well as visual navigation about small celestial bodies such as asteroids. Because *Basilisk* also allows for formation flying capabilities [30], formation flying dynamics have the potential to be paired into a OpNav framework for relative formation control. This allows for true-scale spacecraft models to be used for visual control, with features like self-occultation and realistic camera model. Furthermore, implemented star-maps can be used for realistic attitude determination and control, all within a closed loop software framework. For entry, decent, and landing (EDL) and asteroid missions' safety, these developments can add an important element of reliability by providing a testbed for autonomy and quantifying performance. Simultaneous Localization And Mapping (SLAM) and cross-correlation methods could also be implemented and tested in a realistic spacecraft environment. These algorithms are currently being developed for novel navigation purposes notably within NASA and ESA [31].

This paper also details the new software architecture that allows the *Vizard* software to become a highly configurable visual sensor module for *Basilisk*. First the numerical performance and computational cost of the communication overhead is explored. Next the visualization optical sensor module is validated for specific OpNav purposes. Finally a pointing scenario is developed in order to showcase the architecture's performance.

Software Interface Architecture

Overview of the Basilisk and Vizard Software

Basilisk is an open-source astrodynamics framework being developed by the University of Colorado Autonomous Vehicle Systems (AVS) lab and the Laboratory for Atmospheric and Space Physics (LASP). By implementing high-fidelity, faster-than-real-time dynamics, it allows to simulate spacecraft in realistic flight conditions. The inherent speed of the framework and its multithreaded Monte-Carlo capability

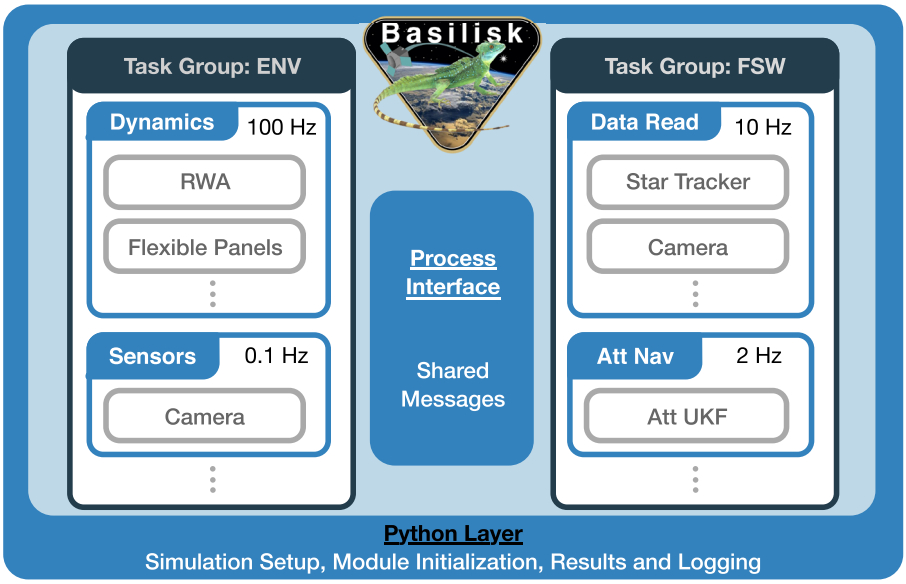


Fig. 2 Schematic Illustration of the *Basilisk* architecture [18, 19, 32]

provides high-end analysis tools. In this simulation, FSW and spacecraft models are placed into different Processes (or task Groups) to isolate their messages. By communicating through the pub-sub messaging system, blocks of code can be added and contribute to the simulation without necessary knowledge of other blocks, as seen in Fig. 2. This interface allows for closed loop control algorithms and simulations to be developed and tested in a highly modular manner where each component has its own unit and integrated tests.

Alongside this effort, *Vizard* [17] receives *Basilisk* state messages and dynamically displays these states. *Vizard* has the ability to import shape-models and planet maps, as well as display and render instrument point-of-view windows. Paired with open-source image processing libraries, such as *OpenCV*[4], these combined components provide the necessary software components to fully simulate autonomous, closed-loop OpNav or other visual sensing and control scenarios. This manuscript develops the software architecture that allows *Vizard* to be integrated into *Basilisk* as a visual sensor module. As a fully open-source project, *Basilisk-Vizard* allows for any user to contribute to the code base, and therefore centralizes progress in astrodynamics.

The modularity of *Basilisk* comes from the fact that modules publish and subscribe without requiring knowledge of other existing modules. Processes (or Task Groups) as pictured in Fig. 2 each own memory for their Tasks to communicate amongst themselves. These message containers can also interface in order to manage the separation

[4] https://opencv.org

of FSW and simulation models. This naturally welcomes another actor: the visualization. By creating a module with access to the required messages, the communication between the software nodes is established.

Faster-than-Realtime Interfacing

The software architecture of *Basilisk* allows *Vizard* to capture information from the spacecraft's environment and communicate it to *Basilisk* mid-run. *Vizard* then creates a three-dimensional visualization of the space environment including planets, moons, stars, other spacecraft, all from the perspective of the current spacecraft location and orientation using a specific body-relative camera frame perspective. After taking rendering this view the resulting image bitmap must be transferred back to *Basilisk* as an image message.

Implementation of the *Vizard* to *Basilisk* interface produces several key challenges. The first is making two heterogenous software entities written in different programming language of C/C++ and C# communicate. Next, the simulation must execute faster-than-real-time to be suitable for navigation and control sensitivity analysis. Finally, these heterogenous components must be integrated while maintaining synchronous operation of the modules through each integration time step. Two types of connections between *Basilisk* and *Vizard* are considered: the 'Direct Connection,' and a connection via *Black Lion* [32, 33].

Black Lion

First consider the case where *Vizard* and *Basilisk* are part of a larger distributed spacecraft simulation which uses the *Black Lion* architecture [32, 33] to communicate across simulation nodes. The benefit of this aproach is that the *Vizard*-based visual sensor *Basilisk* module can be integrated in very general distributed simulation environments, at the cost of additional central controller software. The *Black Lion* package developed in the AVS lab is middleware that ensures proper interfacing between nodes in a heterogeneous, possibly distributed spacecraft simulation. Essentially the message passing interface concept of *Basilisk* is expanded to function across a range of heterogenous simulation components such as a flight processor emulation or ground software system. For the scope of this paper the *Black Lion* nodes are *Basilisk* and the visualization as illustrated in Fig. 3.

In summary, *Black Lion* ensures :

- The transport of binary data via a transport layer (Transmission Control Protocol or TCP). Although User Datagram Protocol (UDP) provides a faster connection, the three-packet exchange provided by a TCP provides the high-reliability necessary for physical simulations. This ensures the camera image is received at the correct time by *Basilisk*.
- The marshaling (or translation) of binary data. Each node must know how to convert the received bytes into structures that can be managed internally.
- The synchronization of nodes to keep all the nodes in lock-step during the simulation run.

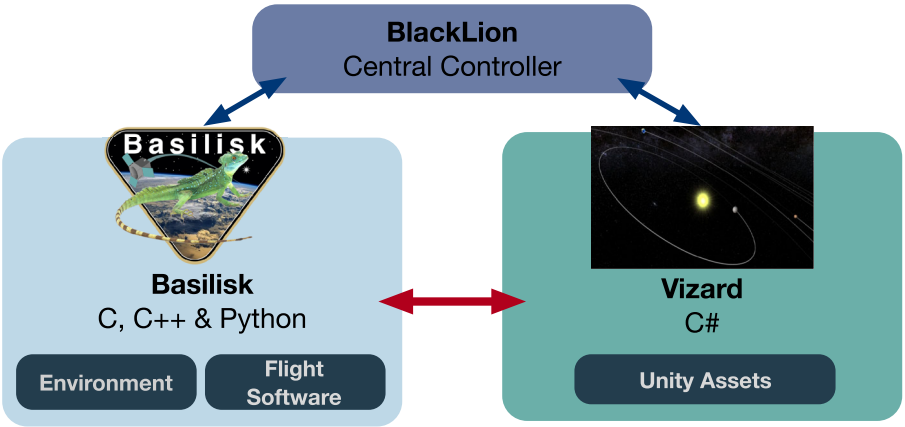


Fig. 3 Interactions between *Basilisk*, *Black Lion*, and *Vizard*

The central controller acts as a master in the synchronization of the nodes, and a broker in the data exchanges.

The last component is the marshaling of the data. Google Protobuffers[5] are used to provide a platform and language agnostic translation layer library to facilitate marshaling and unmarshaling of data between the two simulation applications. By creating these Protobuffer structures, both the C++ code in *Basilisk* and the C# code in *Unity* can read in and write out the necessary content. This method is currently in use at LASP for real-time *Basilisk*-based flat-sat testing while integrating the *Vizard* visualization. It notably allows running distributed simulations over a network. Users can distribute nodes across machines, use hardware in the loop, or run the *Vizard* on a computer with a high-end graphics card. This provides a wealth of optimization strategies with the slight added complexity of interfacing with middleware. Because *Black Lion* enforces synchronization across modules, the synthetic visual sensor images are guaranteed to remain in sync with the spacecraft dynamics simulation in *Basilisk*. This method is primarily aimed for more mature mission concepts. By using hardware in the loop, the faster-than-real-time aspect is lost, but more critical tests can be run.

Direct Communication

When performing fast analysis or making design choices, it is desirable to run *Basilisk* and *Vizard* on the same machine without having to synchronize with other spacecraft simulation components, such as ground software. In order to simplify the interface, a direct communication is implemented which allows for a two-way communication between *Basilisk* and *Vizard* without using *Black Lion* as a middle-ware interface layer. In the absence of a central controller, the `vizInterface` module

[5] https://developers.google.com/protocol-buffers

written in C++ takes on the synching responsibilities. Nevertheless the same methods and tools seen in the *Black Lion*-based implementation are used:

- The transport layer used is a TCP, implemented with *ZeroMQ*.[6]
- The translation layer uses the same Protobuffer structure.
- The synchronization is enforced in the simulation through a blocking communication interface: *Basilisk* waits for critical responses from *Vizard* through *ZeroMQ* before continuing the simulation.

The direct communication protocol utilizes a separate thread to spawn *Vizard* from the python layer to start a request-response pattern. In this direct communication scenario, there are two main modes that the interface can work in: a lock-step mode and a performance mode. The core difference between these is the frequency of communication between the two nodes.

1. Lock-step: In the lock-step mode, *Basilisk* sends updates at every time-step wether or not an image is requested. Lock-step provides a fluid visualization, and renders both the spacecraft camera and *Unity*'s main camera to screen allowing for user-feedback on the simulation setup and initialization. This also opens the possibility of controlling the simulation from the visualization, as it will always wait for a message verifying *Vizard* has received the simulation message. In lock-step mode, *Basilisk* always waits for a message from the *Vizard* saying it can move forward. This keeps the synchronicity as the message queues looks the same on each side and the visualization always has the latest message.
2. Performance: In performance mode, the visualization and the interface are greatly simplified. On the *Vizard* side, the spacecraft camera becomes the main camera. Furthermore, *Vizard* only places and updates simulation states if an image is requested and `vizInterface` only sends a simulation update when an image will be requested. This brings down the number of TCP pings to the camera image rate, instead of the simulation time step. A separate, OpNav-specific application is shipped for this purpose. Paired with *Unity*'s 'batchmode' command-line argument, the application can stay silent and run in the background effectively reducing the *Vizard* application to a simulation module.

Interface Implementations

Figure 3 shows the interaction between the major software nodes for both interface scenarios: via *Black Lion* (blue arrows) or directly (red arrows). Figure 4 outlines the details of the interfaces of the direct communication option. As stated previously, *Basilisk* modules write and subscribe to messages via the Process (or Task Group) message memory space without any knowledge of other existing modules. *Basilisk* contains a new C++ *Vizard* interface module which reads the required *Basilisk* messages, writes them as protobuffers, and gives them to *Black Lion* or *Vizard* directly.

[6] https://zeromq.org

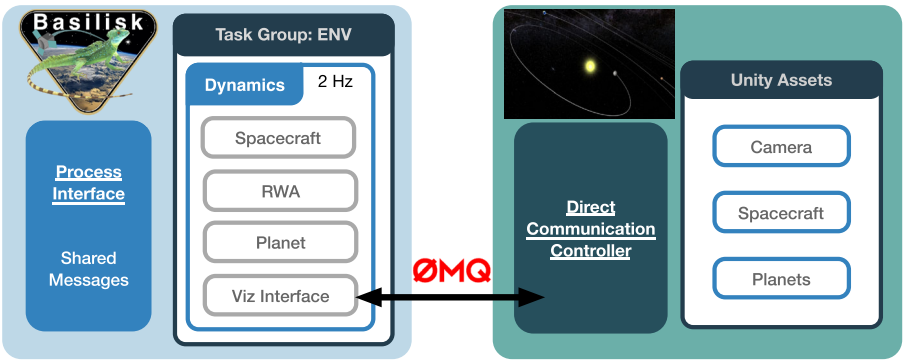


Fig. 4 Direct communication using the viz interface

The Visualization interface then unpacks the protobuffers in order for the game communication controller to use the data.

These design choices reflect the desire for a modular yet robust architecture. The use of Google Protobuffers allows for platform independent communication; *Unity* provides a user friendly and vast community for environment development; *Black Lion* is the middleware that connects and applications and synchronizes them across separate machines. Alternatively, the direct line is created between the visualization and *Basilisk* for the ability to debug, test, and analyze simulations to greater effect. These tools provide the building blocks for the framework being implemented.

Information Flow

In both communication protocols, the information flows back and forth between the two nodes. The main points of the information flow are detailed below and shown in Fig. 5. The module implemented showcases the general capabilities of the simulation, and represents only a fraction of the possible implementations. This architecture's greatest strength is its potential to support further complexity and development, thus making it very extensible. This section shows an example of the data flow that can be achieved with the architecture.

1. The `vizInterface` module in *Basilisk* checks for new information in the simulation. If any data has changed at a simulation time-step, the protobuffer message is updated. In the absence of change, the module will do nothing.
2. The protobuffer is passed to *Unity* (black arrow next to *ZMQ* logo) and packed in a dictionary. This allows for a simulation update on the game engine side and an image render if requested.
3. If an image is requested—this can be done through the presence of an image request message in the simulation, or if the simulation time is a multiple of the camera render rate—*Vizard* renders the texture viewed by its interal camera according to specifications. This is then sent back to Basilisk in a bit-map format while the simulation awaits the return message.

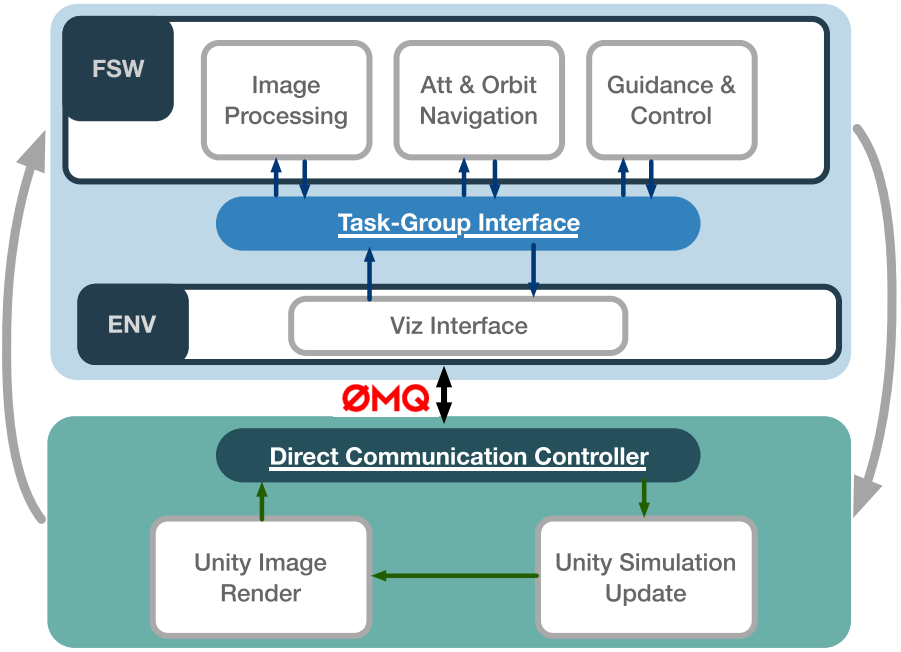
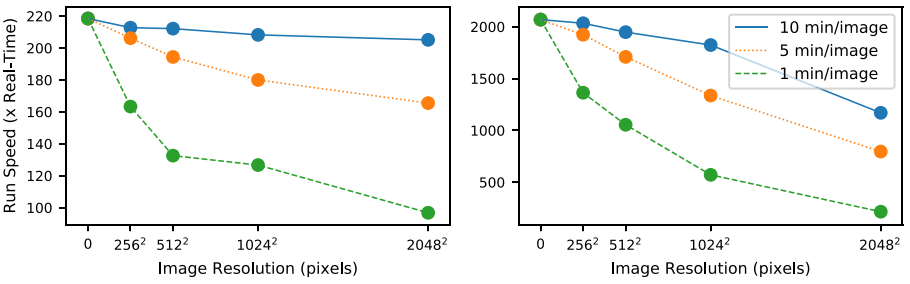


Fig. 5 Information flow between the visualization and the simulation

4. The received image is unpacked in the `vizInterface` module and repacked in a C structure for the rest of the *Basilisk* simulation to use (with relevant information such as time of capture and camera used). FSW modules are traditionally written in C, therefore the bitmap is recast to a void pointer in *Basilisk*. This prevents numerous copy operations of the image data and requires no dynamic allocation (which requires a `malloc` in C).
5. This image will be read by the image processing module, which will extract spacecraft relative position with Centroid and Apparent Diameter (CAD) algorithms. Figure 5 could also picture camera models for additional realism. This can include the CCD's sensitivity to certain colors, realistic jitter using true attitude variations, etc.
6. This value is next sent to an Orbit Determination filter or Attitude Guidance module along with the associated covariance as a measurement for position estimation. These FSW-specific exchanges are seen in Fig. 5 with the blue arrows which represent the pub-sub messaging system calls.
7. With an updated state estimate, the spacecraft can now control its attitude, position, and velocity.
8. These updated states are tracked by the `vizInterface` and sent back to *Vizard* for a new sim update in the visualization.

Unity can save images to an external file. This allows for a log of the images that were taken to be saved for debugging and validation. The `vizInterface` module



(a) Lock-Step Mode

(b) Performance Mode

Fig. 6 Performance of Both Closed-Loop Implementations

also saves all the protobuffers from a run to file, this allows for playback capability on every simulation run.

Closed-Loop Simulation and Performances

If the closed-loop simulations are to be used for Monte-Carlo analysis and Machine Learning, the software architecture presented must allow for faster-than-real-time speeds. As explained in the above subsection, there are two different modes that can be used, with different performance goals. Figure 6 plots simulated time divided by run times (averaging over 5 runs) for varying camera quality and render rates. The x-axis represents the image size and although the ticks read total number of pixels, the scale is linear (square-root of the tick labels) for legibility. The Fig. 6a shows the results of the lock-step mode, while Fig. 6b shows the performance-mode results. All tests are run on a MacBook Pro running macOS Version 10.13.6, a 3.5 GHz Intel Core i7 processor with 16GB of memory, and Intel Iris Plus Graphics 650 graphics card.

The simulation used in this section is an OpNav-point scenario developed in the last section of this manuscript. This 100min simulation has a 0.5s integration time step, and implements a spacecraft dynamics module alongside FSW algorithms for attitude control running at the same 2Hz. The spacecraft searches for Mars and points to it when able using an MRP-feedback control law [34]. The architecture provides speeds that allow for Monte-Carlo analysis and machine learning scenarios to be run in a reasonable amount of time. The first thing to notice is that performance-mode provides an order of magnitude speed-up relative to its 'Lock-Step'-mode predecessor. This difference shows the main slow-down incurred comes from *Basilisk* needing to wait for *Unity*'s updates. Furthermore, Fig. 6b shows that if the sim moves forward with minimal communication, the rendering of the image becomes the expensive operation. Images in Fig. 7 show the images that are received by the simulation for processing. This is done with a 60s camera render-rate in order to capture the motion of Mars. The planet is not initially visible from the camera's perspective, but the search algorithm brings the planet into the camera frame. In the second half of the run (bottom line), the planet is fully in view of the spacecraft. These images are from

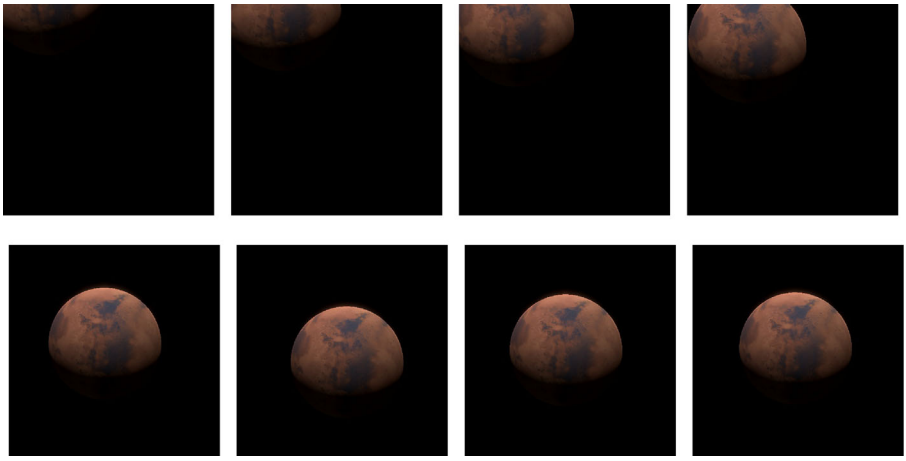


Fig. 7 Camera view as the spacecraft moves the simulation as received by *Basilisk*

the scenario described in the last section of this manuscript, and can import a wide variety of Mars surface maps.[7]

Optical Navigation Components

Optical navigation tracks planet and moon centroids and dimensions to determine the spacecraft location. Several optical navigation methods exist, such as star horizon [35], centroid and apparent diameter [36], star occultation [37], and landmark tracking [13]. Each of these have their specific application scenarios depending on the object they are required to track.

Centroid and apparent diameter measurements find the limb of a body and use the knowledge of its actual size and position. By extracting direction and distance, range and position information is extracted from planet images. The shape of the partially illuminated moon alone permits the estimation of the direction vector to the sun using just a star tracker [38]. With the knowledge of the body in sight, its ephemeris, and its size, determination of both the spacecraft's orbit and attitude can be achieved. These methods require minimal image processing power, are relatively fast to implement, and provide a wealth of extractable information from images. The use of *OpenCV* has helped accelerate module design. For these reasons they will be the baseline methods used in simulating autonomous OpNav.

However, other OpNav methods yield better navigation results. Measurements derived from landmark observations [13], point distribution methods [39], or crater detection [40] are some of the many feature tracking methods which provide promising results. The real-time component of this framework creates a realistic

[7] https://celestiamotherlode.net/catalog/mars.php

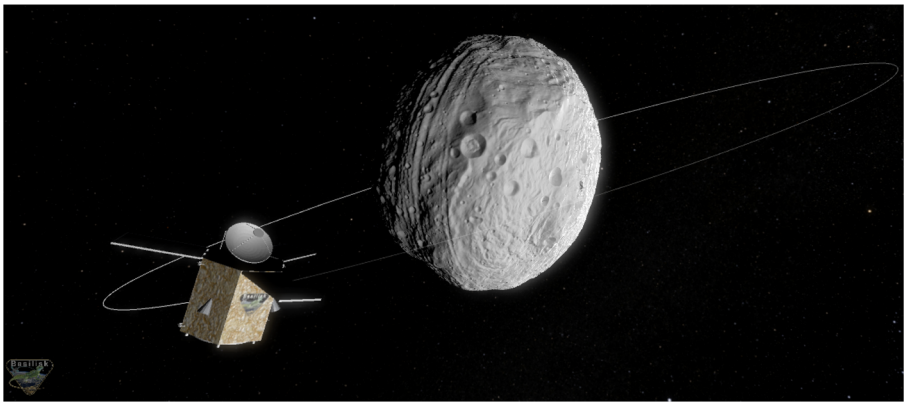


Fig. 8 Vesta shape model uploaded into the visualization

environment to quantify and run more computationally extensive algorithms. Future work will include higher-fidelity star-maps in order to do star-horizon detection [35], amongst other methods that have been described in the literature.

This software framework allows for rapid and high-fidelity testing, and can centralize progress from other fields within astrodynamics. In the aerospace field this has been seen with ORB-SLAM development [41, 42] and cross-correlation methods [31, 43]. These hold great promise for small body autonomous orbiting and have already proven to be useful on missions such as ESA's Rosetta and ongoing missions such as OSIRIS-REx. Although implementing such methods in *Basilisk* are currently advanced goals, this architecture allows for these additions. *Vizard* allows users to upload shape models for any celestial body as seen in Fig. 8 with Vesta.[8] This provides the opportunity to train and test shape model reconstruction methods by using fully coupled spacecraft attitude and orbital dynamics.

Centroid tracking and apparent diameter measurements are the baseline OpNav methods in this design. In parallel, developments for feature tracking will be added in along with more image processing capabilities.

Camera Models and Validation

In order to realistically model OpNav scenarios, the images generated by the visualization software must be on par with the method in use. *Unity* provides a large set of lighting libraries that can simulate self-shadowing and model lighting on imported shape-models. This allows for the generation of complex lighting scenes. By extension, it allows for the reading in of partially lit planets, showing crescent lighting. This lighting is seen in the visualization in Fig. 11a. In order to speed up the simulation as much as possible, *Unity* camera models are used to simulate a realistic camera

[8] https://nasa3d.arc.nasa.gov/detail/asteroid-vesta

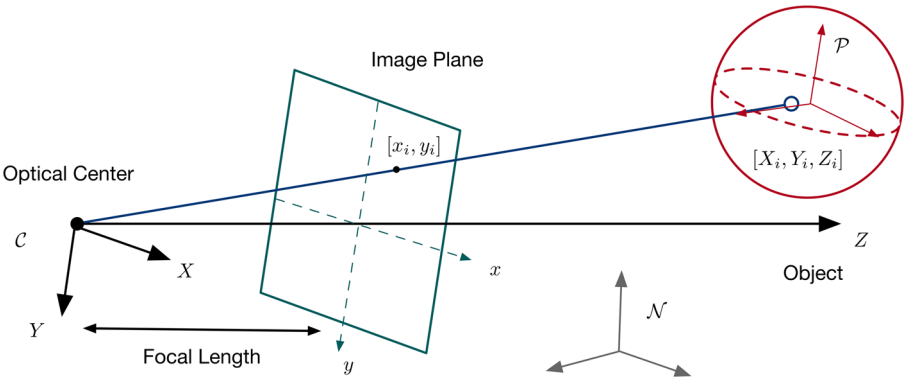


Fig. 9 Pinhole camera model

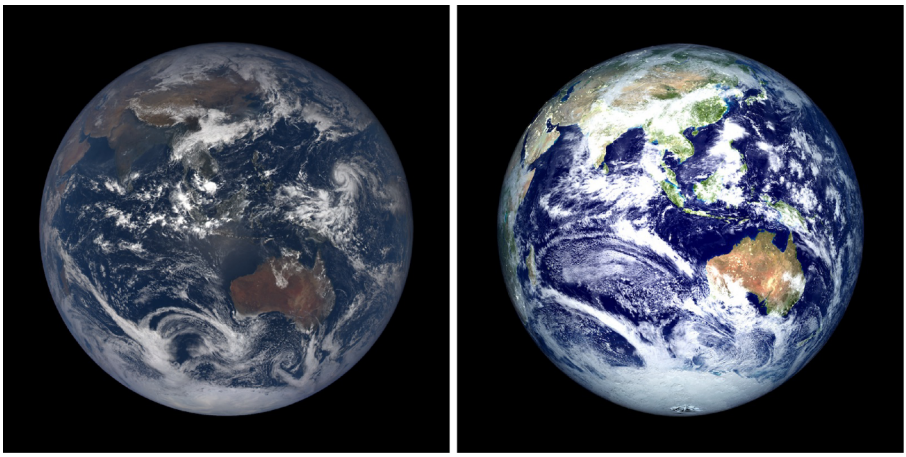
model which is a pinhole model seen in Fig. 9. A *Basilisk* camera model can be created as well in order to add more complicated errors. Lens-flaring or lens-distortion [44] can distort images and can be compensated for in post-processing. [45] Though not a method in *Unity 2018*, the modularity of the software package allows for such additions.

Figure 10 shows the visualization compared to true data taken by the Epic camera on DSCOVR. On the left is an image taken from Lagrange 1 on October 23rd at 4:35:25 UTC. The image is obtained from the Epic website[9] which also provides the camera specifications. These are provided in Table 1, and were used as such in the *Unity* camera model. Sensor size and field of view lock in the focal length, and that resolution and sensor size lock in the pixel size. Therefore all the needed information is provided regarding the camera.

Regarding the spacecraft position, the source provides distance between DSCOVR and Earth, DSCOVR-Sun distance, Earth-Sun distance, as well as the Sun-Earth-Craft angle. These values are provided in Table 2. It is important to note that they do not provide a unique possible position for the spacecraft. The spacecraft therefore lies on a circle off the Sun-Earth direction by 7.28 °. Since the exact position is not made public, the simulation placed the spacecraft exactly on the Earth-Sun direction, with the expectation of seeing some differences. The Earth and Sun were placed in the simulation using *Spice*,[10] which provides the Sun and Earth's ephemerides, as well as Earth's rotation in the inertial frame.

Figure 10 illustrates that the actual mission image and the synthetic *Vizard* image look very similar as the Earth has the same apparent location and size in the photo, and the continents are lined up correctly as well. Only a slight shift can be seen in the Earth's relative position: Australia seems to be more to the South-West on the real

[9] https://epic.gsfc.nasa.gov/?date=2018-10-23
[10] https://naif.jpl.nasa.gov/naif/



(a) DSCOVR image (b) Raw Vizard Image

Fig. 10 Comparing Vizard images to real data

image. This is certainly due to the 7.28° error in the camera's position. Besides this, the only significant differences are seen in the contrast and texture quality. It would seem that the colors seen by epic are more matte. This can be improved and modeled in the visualization as no color sensitivity is modeled, but for the purpose of centroid and apparent diameter, the results are sufficiently accurate.

A simple CAD algorithm provides a center point at the pixel coordinates (1023.71, 1023.03) and an apparent radius of 859 pixels for the real data. By running the same algorithm the simulation predicted a planet center at (1023.50, 1027.68), and a radius of 854 pixels. This represents a relative error of 0.46% pixels on the center's position and 0.58% error on the radius. These are relatively small errors given the uncertainty in the spacecraft ephemeris information.

Image Processing Methods

The biggest advantage of the software framework presented is its modularity. Certain state-of-the-art limb-fitting algorithms for pose-estimation are on-board capable [11] and can be implemented into *Basilisk* and used in the simulation. This allows for speed tests as well as better general performance understanding. In this manuscript,

Table 1 Epic camera paramters

Parameter	Field of View [°]	Resolution [pixels]	Sensor Size [mm]
Value	0.62	2048×2048	[30.72 , 30.72]

Table 2 DISCOR position

Parameter	Earth-S/C [km]	Sun-S/C [km]	Sun-Earth [km]	Sun-Earth-S/C angle [°]
Value	1,405,708	147,451,774	148,846,039	7.28

OpenCV[11] is chosen as a computer vision library. This saves development time by utilizing a robust software library with widely tested functionality.

The transformation used here is a Hough transform for circle finding, which exists in many derivative forms [46, 47]. Figure 11 displays the transformations that an image from the visualization is put through in order to extract apparent diameter and centroid information. The raw image is turned into grey-scale (Fig. 11b) before being fed into the HoughCircles function. This function then blurs and thresholds (Fig. 11c) the image within a call to the Canny edge detection transform (Fig 11d). This is then the image used in order to accumulate votes [48] on the existing circles in the image. Figure 11e shows overall good performance by the algorithm. Other examples using images of the Moon and Encaladus are shown in Fig. 12. It can be seen in some of the images in Fig. 12 that although the algorithm is generally quite robust, sometimes the radius of the planet is underestimated. This is seen notably in Fig. 12c. Image processing imperfections emphasize the necessity to output a measure of uncertainty with the Hough transform.

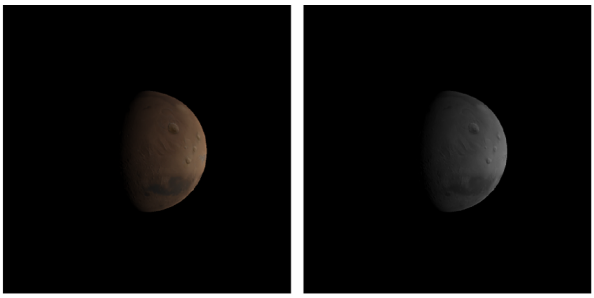
Attitude Guidance and Control Example

In this section, an example scenario is developed in order to illustrate the *Basilisk-Vizard* capabilities. A spacecraft is on orbit around Mars and seeks to align its camera bore-sight with the planet center. It takes images periodically for attitude guidance and control, and uses the Hough algorithm to extract the center and the apparent diameter of the planet being observed. The pixel data is pre-processed before being used to determine the planet direction. Initial conditions for the simulation are given in Tables 3 and 4, while simulation and flight software parameters are given in Tables 5 and 6. All modules listed are currently available on the *Basilisk* bitbucket repository[12] with additional documentation. The main assumption in these models is that the reference is static, in this case that the planet does not move in the camera frame. Although this is an erroneous assumption, it can be desirable to see how it holds, when it breaks, and what are the parameters (camera image rate, field-of-view, orbit elements) for each of these cases. This scenario puts this assumption to the test.

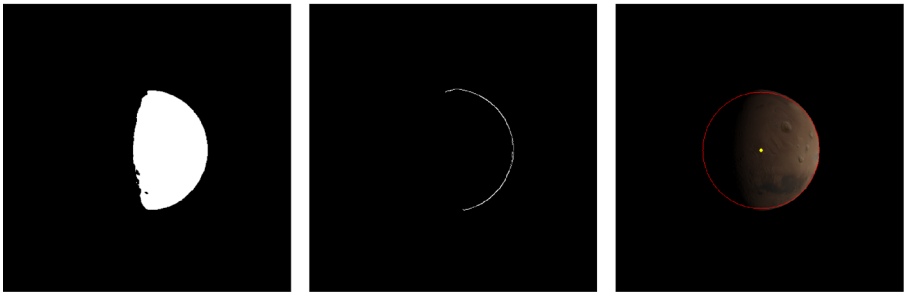
The Pixel and Line Transformation module performs the simple transformation from pixel data to spacecraft relative position. This scenario feeds raw measurements of the planet center (x_c, y_c) in pixels to the guidance module, with the knowledge of

[11] https://opencv.org

[12] https://bitbucket.org/avslab/basilisk



(a) Raw image (b) Gray scale



(c) Blurred and Thresholded (d) Canny edge detection (e) Hough circle

Fig. 11 Extracting center and apparent diameter from visualization image using *OpenCV*

(a) Slim Moon cresent (b) Thick Moon crescent (c) Enceladus

Fig. 12 Hough circle finding on several real images (Courtesy NASA/JPL-Caltech)

Table 3 Spacecraft Initial States

$\sigma_{\mathcal{BN}}$	$\omega_{\mathcal{BN}}$	Orbital elements $(a, e, i, \Omega, \omega, f)$
$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$	$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$	18000km, 0, 20°, 25°, 190°, 100°

Table 4 Camera parameters

$\sigma_{\mathcal{CB}}$	${}^{B}r_C$[m]	Resolution [pixels]	Sensor Size [mm]
$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$	$\begin{bmatrix} 0 & 0.2 & 0.2 \end{bmatrix}^T$	$\begin{bmatrix} 512 & 512 \end{bmatrix}^T$	$\begin{bmatrix} 10 & 10 \end{bmatrix}^T$

Table 5 Simulation parameters

Simulation Modules Instantiated	Necessary parameters at initialization
Spacecraft Hub	Inertia $[I] = \text{diag}(900, 800, 600)$ kg·m, mass $M = 750$kg
Gravity Effector/Eclipse	December 12th 2019 at 18:00:00.0 (Z), $\mu_{\text{mars}} = 4.28284 \cdot 10^{13}\text{km}^3/\text{s}^2$
Simple Navigation Star Tracker	Attitude error $\sigma_{\text{att}} = 1/3600°$, Rate error $\sigma_{\text{rate}} = 5 \cdot 10^{-5}°/\text{s}$
Reaction Wheel Effector	4 Honeywell HR16 Wheels*
Wheel orientations	Elevation Angles 40°, Azimuths angles 45°, 135°, 225°, 315°
	Positions in $\mathcal{B}$ [m] $\left[\ 0.8, 0.8, 1.79070\ \right]^T$ $\left[\ 0.8, -0.8, 1.79070\ \right]^T$ $\left[\ -0.8, -0.8, 1.79070\ \right]^T$ $\left[\ -0.8, 0.8, 1.79070\ \right]^T$

https://aerospace.honeywell.com

the camera. Notably the pixel size is given by $X = \frac{\text{SensorSize}_x}{\text{Resolution}_x}$ and $Y = \frac{\text{SensorSize}_y}{\text{Resolution}_y}$ in mm/pixel.

$$^{C}r_{BN} = -\left[\ \frac{X}{f} \cdot \left(\text{x}_c - \frac{\text{Resolution}_x}{2} + \frac{1}{2}\right)\ \frac{Y}{f} \cdot \left(\text{y}_c - \frac{\text{Resolution}_y}{2} + \frac{1}{2}\right)\ 1\ \right] \quad (1)$$

where $^{C}r_{BN}$ is the relative vector of the camera bore-sight with respect to the celestial center, where the left superscript represents the frame the vector is projected onto. f is the camera field of view, and the transformations on the measurements also re-center the pixels [35, 49, 50]. Since the measurements in this scenario are given raw to the guidance module and without consideration of covariance, this completes the measurement transformation.

OpNav Point Guidance

This simulation specifically uses the OpNav-Point module for guidance. The attitude guidance module has the goal of aligning a commanded camera-fixed spacecraft vector $\hat{h}_c$ with the measurement vector h. Here, $\hat{h}_c$ is the camera bore-sight, and so

Table 6 Flight software parameters

Flight software modules instantiated	Necessary parameters at initialization
Image Processing	param1 = 300, param2 = 20, minDist = 50
(arguments for HoughCirlcle method[1])	minRadius = 20, dp = 1, maxRadius = 409
OpNav Point	minAngle = 0.001°, timeOut = 100s
	$\omega_{\text{search}} = [0.06, 0.0, -0.06]°/\text{s}, {}^{C}h_c = [0., 0., 1]$m
Pixel Line Transform	Planet Target is Mars
MRP Feedback RW	K = 3.5 , P =30 (no integral feedback)
RW motor Torque	Control axes are $^{B}\left[\ b_1, b_2, b_3\ \right]$

https://docs.opencv.org/

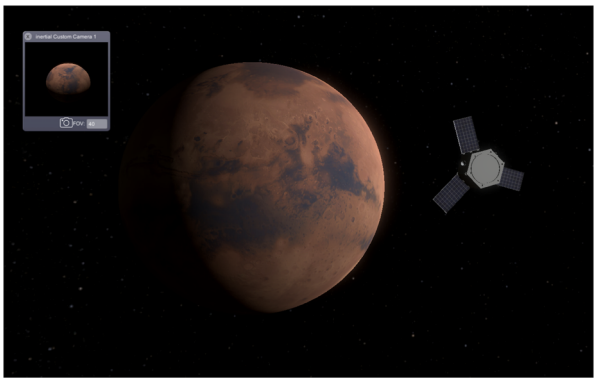


Fig. 13 OpNav Pointing Scenario

the attitude tracking errors seek to align the camera towards the target and achieve relative pointing.

In the following developments, all vectors are assumed to be taken with respect to a camera-fixed frame $\mathcal{C}$ if a frame is not specified. The attitude of the camera relative to the target reference frame $\mathcal{R}$ is written as a principal rotation from $\mathcal{R}$ to $\mathcal{C}$. The target $\mathcal{R}$ is defined simply by the vector $\hat{h}_c$ and body frame vectors. The body frame is defined as $B : \{\hat{\boldsymbol{b}}_1, \hat{\boldsymbol{b}}_2, \hat{\boldsymbol{b}}_3\}$.

At the start of the simulation, the camera does not have the planet in sight. In this situation, a search rate is requested: ${}^{B}\omega_{\text{search}}$. Once the planet is found, and measurements are provided, the requested rate is zeroed to keep the spacecraft at rest. The module also has the ability to request a rotation about the camera bore-sight for stability. This module is designed for simplicity and robustness. In order to be independent from a orbit determination solution, the inertial reference frame acceleration $\dot{\omega}_{R/N}$ is set to zero. This means the guidance will need to constantly adjust to a moving reference. Figure 13 pictures the spacecraft once Mars has been found and is being tracked.

Similarly to a sun-safe point guidance law, this module does not establish a unique target-pointing reference frame. Rather, it simply aligns $\hat{h}_c$ with h, which is an underdetermined 2 degree of freedom condition. If these two vectors are nearly collinear, numerical instabilities can occur, hence the `minAngle` variable set by the user.

The associated principal rotation vector $\hat{e}$ and angle Φ between $\hat{h}_c$ and h are

$$\hat{e} = \frac{h \times \hat{h}_c}{|h \times \hat{h}_c|} \qquad \Phi = \arccos\left(\frac{h \cdot \hat{h}_c}{|h|}\right) \tag{2}$$

If Φ is less then the module parameter `minAngle`, it is assumed that no valid planet heading vector is available and the attitude tracking error $\sigma_{C/R}$ is set to zero. For valid planet headings, this rotation from $\mathcal{R}$ to $\mathcal{C}$ is written as a set of MRPs through

$$\sigma_{C/R} = \tan\left(\frac{\Phi}{4}\right)\hat{e} \tag{3}$$

The set $\sigma_{C/R}$ is the attitude error of the output attitude guidance message. If the spacecraft is to be brought to rest, $\omega_{R/N} = 0$, then the tracking error angular velocity vector is computed using:

$$\omega_{B/R} = \omega_{B/N} - \omega_{R/N} \qquad \dot{\omega}_{R/N} = 0 \tag{4}$$

The attitude guidance message must specify the inertial reference frame acceleration vector. This is set to zero and is the assumption that needs to be justified as it can be poorly representative of reality.

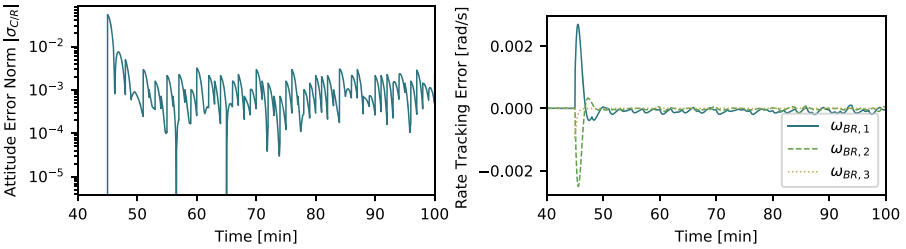
This concludes the module description, which is summarized in the following algorithm. The details of the implementation are currently available on the *Basilisk* open source package in the folder *fswAlgorithms/attGuidance/opNavPoint/*.

Algorithm 1 OpNavPoint.

OpNavMeas ← read(PixelLine)
firstPass ← True
timeOut ← False
if OpNavMeas is valid or (!firstPass and !timeOut) **then**
  if OpNavMeas is valid **then**
    ${}^{C}\hat{h}$ ← read(OpNavMeas)
    save(${}^{N}\hat{h}$)
    firstPass ← False
  else if !firstPass and !timeOut **then**
    ${}^{C}\hat{h} \leftarrow [\mathcal{CN}]^{N}\hat{h}$
  angleError ← arccos(${}^{C}\hat{h}_c \cdot {}^{C}\hat{h}$)
  if angleError ≤ minAngle **then**
    $\sigma_{\text{guid}} \leftarrow 0$
  else
    $\hat{e} \leftarrow \text{cross}({}^{C}\hat{h}, {}^{C}\hat{h}_c)$
    $\sigma_{\text{guid}} \leftarrow \tan(\frac{1}{4}\text{angleError})\hat{e}$
  $\omega_{\text{guid}} \leftarrow \omega_{\mathcal{BN}} - \omega_{\mathcal{RN}}$
else if Search **then**
  $\sigma_{\text{guid}} \leftarrow 0$
  $\omega_{\text{guid}} \leftarrow \omega_{\text{search}}$

OpNav Relative Pointing Results

Running this scenario using the *Basilisk-Vizard* interface shows interesting control results, which test the validity of the assumption stated previously. The spacecraft finds the planet after 40mins of searching with a slow search rate defined in Table 6. The assumption of holding the target frame static in the inertial frame holds well with dense measurements. Furthermore, with a fast run-speed, it is easy to test different setups and tailor the simulation to a specific goal or requirement. The control plots once the planet is in sight are pictured here: Figure 14 shows the attitude error norm and rate tracking error once the planet is found, for a constant 1minute gap between



(a) Attitude Error Norm

(b) Rate Tracking Error

Fig. 14 Attitude Control Results

consecutive images. Although the attitude error in Fig. 14a shows that the algorithm is constantly surprised by the new images, the error stays close to 10^{-3}. Figure 14b shows that the rates mirror this lag with the X and Y components oscillating to control the spacecraft onto the target. Finally, Fig. 15 shows the true pixels as crosses, alongside the pixels measurement by the HoughCircle transform as dots. It is important to note that this simple transform is performing well as its estimates are very close to expected values, which are computed with the true and noiseless spacecraft attitudes and positions. Furthermore, the measurements show the gap that is seen in Fig. 14a: each new measurement appears off-center. This is due again to a changing reference frame that the guidance module needs to constantly keep up with. Despite this, the planet stays in frame throughout the control and pixel offsets, as long as they are representative of the truth and do not hinder orbit determination. In a situation

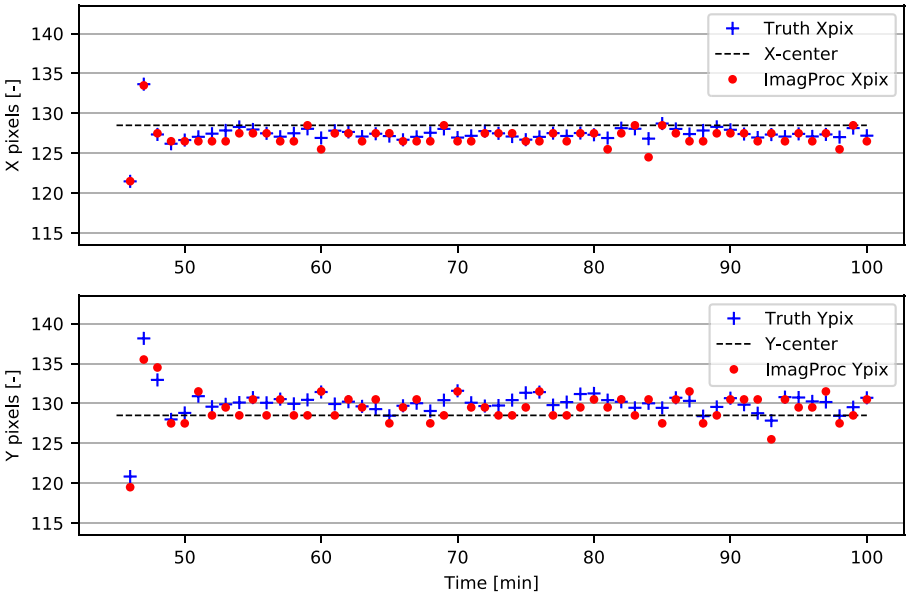


Fig. 15 OpNav Pointing Scenario Measured Pixels vs Expected Pixels

akin to the *New Horizons* Pluto fly-by, this simulation can provide a test environment for autonomous pointing algorithms.

Conclusions

This software architecture provides a testbed for a modern simulation framework to research visual navigation and control applications, including optical navigation and other novel navigation methods. Through the closed loop, coupled interaction between the simulation and the visualization, scenarios provide high-fidelity data at fast rates. Amongst other future endeavors, this architecture opens the door to Machine Learning techniques and Monte Carlo analysis. The open source nature of the project allows for continuous validation from the community, and contributions from developers around the world. Using a simulated camera, optical navigation methods, and the closed loop visualization-simulation interaction, these visual control spacecraft scenarios are tested in a relative pointing scenario. The simulation framework allows a user to make, test, and verify a hypothesis with ease, showcasing the ability for fast and robust analysis.

Acknowledgments The authors would like to acknowledge Mar Cols Margenet, Patrick Kenneally, Scott Piggott and Jennifer Wood for *Black Lion* and *Vizard* development.

Compliance with Ethical Standards

Conflict of interests On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Starek, J.A., Açıkmeşe, B., Nesnas, I.A., Pavone, M.: Spacecraft Autonomy Challenges for Next-Generation Space Missions, pp. 1–48. Springer, Berlin. https://doi.org/10.1007/978-3-662-47694-9_1 (2016)
2. Riedel, S.B.J.E.: Using Autonomous Navigation for interplanetary missions: The validation of Deep Space 1, Technical report
3. Kubitschek, D.G.: Impactor spacecraft targeting for the deep impact mission to comet Tempel 1, No. 03-615. Astronautical Society (2003)
4. Kubitschek, D.G.: Deep Impact Autonomous Navigation: the trials of targeting the unknown, 29th Annual AAS Guidance and Control Conference. Breckenridge, Colorado. Jet Propulsion Laboratory, National Aeronautics and Space Administration (2006)
5. Cameron, J.M.: Next generation simulation framework for robotic and human space missions, no. 5151 in AIAA SPACE conference and exposition. Jet propulsion laboratory, California institute of technology, Pasadena (2012)
6. Lim, C.S., Jain, A.: Dshell++: a component based, reusable space system simulation framework, Third IEEE international conference on space mission challenges for information technology jet propulsion laboratory california institute of technology (2009)
7. Quigley, M.: ROS: An open-source Robot Operating System, Technical report, Computer Science Department, Stanford University, Stanford (2007)
8. Alexander, B.: Robot Web Tools [ROS Topics]. IEEE Robot. Autom. Mag. 19, 20–23 (2012)
9. Li, S.: Image Processing Algorithms For Deep-Space Autonomous Optical Navigation. J. Navigat. **66**, 605–623 (2013)

10. Christian, J.: An On-Board Image Processing Algorithm for a Spacecraft Optical Navigation Sensor System, AIAA Space Conference and Exposition. AIAA, Anaheim (2010)
11. Christian, J.: Accurate Planetary Limb Localization for Image-Based Spacecraft Navigation. J. Spacecr. Rocket. 54(3), 708–730 (2017)
12. Dor, M., Tsiotras, P.: ORB-SLAM Applied to spacecraft Non-Cooperative rendezvous. American institute of aeronautics and astronautics 2018/05/21. https://doi.org/10.2514/6.2018-1963 (2018)
13. Liounis, A.: Autonomous navigation system performance in the Earth-Moon system, AIAA space conference and exposition. v, San diego (2013)
14. Christian, J.: Optical Navigation Using Planet's Centroid and Apparent Diameter in Image. J Guid. Control Dyn **38**, 2 (2015)
15. Schlei, Y.G.W.: New Horizons 2014MU69 Flyby Design and Operation, AAS/AIAA Space Flight Mechanics Meeting. Ka'anapali, HI, Paper No. AAS-19-334 (2019)
16. Harch, A., Carcich, B., Rogers, G., Williams, B., Williams, K., Owen, B., Bauman, J., Birath, E., Bowman, A., Carranza, E., Dischner, Z., Ennico, K., Finley, T., Hersman, C., Holdridge, M., Jackman, C., Kang, H., Olkin, C., Pelletier, F., Peterson, J., Redfern, J., Rose, D., Stanbridge, D., Stern, A., Vincent, M., Weaver, H., Whittenburg, K., Wolff, P., Young, L.: Accommodating Navigation Uncertainties in the Pluto Encounter Sequence Design, pp. 427–487. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-51941-8_21 (2017)
17. Wood, J., Margenet, M.C., Kenneally, P., Schaub, H., Piggott, S.: Flexible basilisk astrodynamics visualization software using the unity rendering engine, AAS guidance and control conference. Breckenridge, CO (2018)
18. Alcorn, J., Schaub, H.: Simulating Attitude Actuation Options Using the Basilisk Astrodynamics Software Architecture, 67Th International Astronautical Congress, Guadalajara (2016)
19. Kenneally, P.W., Piggott, S., Schaub, H.: Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework, 7Th International Conference on Astrodynamics Tools Adn Techniques (ICATT). DLR Oberpfaffenhofen, Germany. https://doi.org/10.2514/1.I010762 (2018)
20. Kenneally, P.W., Schaub, H.: High Geometric Fidelity Modeling of Solar Radiation Pressure Using Graphics Processing Unit, AAS/AIAA Spaceflight Mechanics Meeting, Napa Valley, California, pp. 2577–2587. Paper No. AAS-16-500 (2016)
21. Kenneally, P.W., Schaub, H.: Modeling Solar Radiation Pressure With Self-Shadowing Using Graphics Processing Unit, AAS Guidance, Navigation and Control Conference, Breckenridge. Paper AAS 17-127 (2017)
22. Kenneally, P.W., Schaub, H.: Parallel spacecraft solar radiation pressure modeling using Ray-Tracing on graphic processing unit, international astronautical congress, Adelaide, Australia. Paper No. IAC-17,C1,4,3,x40634 (2017)
23. Alcorn, J., Allard, C., Schaub, H.: Fully coupled reaction wheel static and dynamic imbalance for spacecraft jitter modeling. Control, AIAA J. Guid. Dyn. **41**(6), 1380–1388 (2018). https://doi.org/10.2514/1.G003277
24. Alcorn, J., Allard, C., Schaub, H.: Fully-Coupled Dynamical Jitter Modeling Of Variable-Speed Control Moment Gyroscopes, AAS/AIAA Astrodynamics Specialist Conference, Stevenson, WA. Paper No. AAS-17-730 (2017)
25. Allard, C., Schaub, H., Piggott, S.: General hinged solar panel dynamics approximating First-Order spacecraft flexing, AIAA journal of spacecraft and rockets, vol. 55, 1290–1298. https://doi.org/10.2514/1.A34125 (2018)
26. Cappuccio, P., Allard, C., Schaub, H.: Fully-Coupled Spherical Modular Pendulum Model To Simulate Spacecraft Propellant Slosh, AAS/AIAA Astrodynamics Specialist Conference. Snowbird, UT. Paper No. AAS-18-224 (2018)
27. Allard, C., Diaz-Ramos, M., Schaub, H.: Spacecraft Dynamics Integrating Hinged Solar Panels and Lumped-Mass Fuel Slosh Model. AIAA/AAS Astrodynamics Specialist Conference, Long Beach (2016)
28. Panicucci, P., Allard, C., Schaub, H.: Spacecraft Dynamics Employing a General Multi-tank and Multi-thruster Mass Depletion Formulation. Journal of Astronautical Sciences. (in press), https://doi.org/10.1007/s40295-018-0133-0 (2018)
29. Allard, C., Diaz-Ramos, M., Kenneally, P.W., Schaub, H., Piggott, S.: Modular Software Architecture for Fully-Coupled Spacecraft Simulations. Journal of Aerospace Information Systems. (in press), https://doi.org/10.2514/1.I010653 (2018)

30. Overeem, S.V., Schaub, H.: Small Satellite Formation Flying Application Using The Basilisk Astrodynamics Software Architecture. International Workshop on Satellite Constellations and Formation Flying, University of Strathclyde, Glasgow. IWSCFF 19-88 (2019)
31. Carson, J.M., Seubert, C., Amzajerdian, F., Bergh, C., Kourchians, A., Restrepo, C., Villalpando, C.Y., O'Neal, T., Robertson, E.A., Pierrottet, D.F., Hines, G.D., Garcia, R.: COBALT: Development Of a Platform to Flight Test Lander GN&c Technologies on Suborbital Rockets. American Institute of Aeronautics and Astronautics. https://doi.org/10.2514/6.2017-1496 (2017)
32. Margenet, M.C., Kenneally, P., Schaub, H.: Software simulator for heterogeneous spacecraft and mission components. AAS guidance and control conference, Breckenridge (2018)
33. Cols Margenet, M., Kenneally, P.W., Schaub, H., Piggott, S.: Simulation Of Heterogeneous Spacecraft And Mission Components Through The Black Lion Framework. John L. Junkins Dynamical Systems Symposium, College Station. No. 7 (2018)
34. Schaub, H., Junkins, J.L.: Analytical Mechanics of Space Systems, 4th ed. AIAA Education Series, Reston. https://doi.org/10.2514/4.105210 (2018)
35. Owen, W.: Methods of Optical Navigation (2011)
36. Christian, J.: Onboard Image-Processing Algorithm for a Spacecraft Optical Navigation Sensor System. J. Spacecr. Rocket. **49**, 2 (2012)
37. Psiaki, M.: Autonomous Lunar Orbit Determination using Star Occultation Measurements, Guidance, Navigation and Control Conference and Exhibit. AIAA, Hilton (2007)
38. Enright, J.: Moon-Tracking Modes for Star Trackers. J. Guid. Control Dyn. 22(1), 20doi:10
39. Tegmark, M.: An Icosahedron-based Method for Pixelizing the Celestial Sphere. The Astrophysical Journal Letters **470**, L81–L84 (1996)
40. Park, W., Jung, Y.: Robust Crater Triangle Matching Algorithm for Planetary Landing Navigation. Jounral of Guidance, Control, and Dynamics, Korea Advanced Institute of Science and Technology, Engineering Note https://doi.org/10.2514/1.G003400 (2018)
41. Mur-Artal, R., Montiel, J.M.M., Tardós, J.D.: ORB-SLAM: A Versatile and Accurate Monocular SLAM System. IEEE Trans. Robot. **31**, 1147–1163 (2015)
42. Mur-Artal, R., Tardós, J.D.: ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras, CoRR, arXiv:16.10.06475 (2016)
43. Martin, A.M.S., Bayard, D.S., Conway, D.T., Mandic, M., Bailey, E.S.: A Minimal State Augmentation Algorithm for Vision-Based Navigation without Using Mapped Landmarks GNC 2017: 10Th International ESA Conference on GNC Systems, vol. 10, Salzburg (2017)
44. Claus, D., Fitzgibbon, A.W.: A rational function lens distortion model for general cameras. 2005 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recogn. (CVPR'05) **1**, 213–219 (2005). https://doi.org/10.1109/CVPR.2005.43
45. Stein, G.P.: Lens distortion calibration using point correspondences. Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 602–608. https://doi.org/10.1109/CVPR.1997.609387 (1997)
46. Leavers, V.F.: Which Hough Transform?, pp. 250–264 vol. 58 No. 2, Department of Physics, King's College, Strand, London. WC2R 2LS (1993)
47. Petkovic, T., Loncaric, S.: An extension to hough transform based on grandient orientation proceedings of the croatian computer vision workshop (2015)
48. Duda, R.O., Hart, P.E.: Use of the hough tranformation to detect lines and curves in pictures. Graphics and Image Processing **15** (1972)
49. Battin, R.H.: An introduction to the mathematics and methods of astrodynamics, revised edition. American institute of aeronautics and astronautics. https://doi.org/10.2514/4.861543 (1999)
50. Owen, W.M.: Optical Navigation Program Mathematical Models Engineering Memorandum, vol. 314–513. Jet Propulsion Laboratory (1991)

Affiliations

Thibaud Teil[1] · Samuel Bateman[2] · Hanspeter Schaub[1]

[1] Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431, USA

[2] Computer Science and Applied Mathematics, University of Colorado Boulder, Boulder, CO USA