Astrobee Robot Software: Modern Software System for Space

Lorenzo Flückiger[1], Kathryn Browne[1], Brian Coltin[1], Jesse Fusco[2],
Theodore Morse[1], Andrew Symington[1]

[1] *SGT Inc., NASA Ames Research Center, Moffett Field, CA, 94035*
[2] *NASA, NASA Ames Research Center, Moffett Field, CA, 94035*
Emails: lorenzo.fluckiger@nasa.gov; kathryn.browne@nasa.gov; brian.j.coltin@nasa.gov; theodore.f.morse@nasa.gov; andrew.c.symington@nasa.gov

ABSTRACT

Astrobee is a new free-flying robot designed to operate inside the International Space Station and perform surveying, monitoring, sensing and scientific tasks. Astrobee's capabilities include markerless vision-based navigation, autonomous docking and charging, perching on handrails to preserve energy, and carrying modular payloads. Its open-source flight software is based on the Robot Operating System, and runs on three interconnected smart phone class processors. We present an architectural overview of this software and discuss the lessons learned over the development life cycle. A key feature of this software is a programming interface that enables research teams to integrate their projects with Astrobee. We highlight several research projects that are already using this interface to develop and test novel ideas that may ultimately end up as on-orbit experiments aboard Astrobee.

1 INTRODUCTION

Astrobee is a new class of free-flying robot shown in 1 that is being developed by NASA to perform a range of functions inside the International Space Station (ISS). Astrobee's propulsion system uses impellers to actively compress air into a chamber, which it then carefully redirects through one of six nozzles. Each Astrobee has two propulsion modules to provide full holonomic control of the platform. Three Astrobees and a supporting dock will be launched to the ISS towards the end of 2018.

Astrobee packs a tremendous range of sensing, processing and networking capabilities into a one foot cube [1]. Driving these capabilities is open-source flight software running on three interconnected processors that interacts with seven micro-controllers, six cameras, two propulsion modules, and numerous other peripherals. This software enables Astrobee to navigate safely, perform autonomous docking and recharging, as well as perch on handrails to conserve energy. This paper focuses on this flight software, and describes the objectives and requirements driving its design, the implementation approach, and the lessons learned from its development.

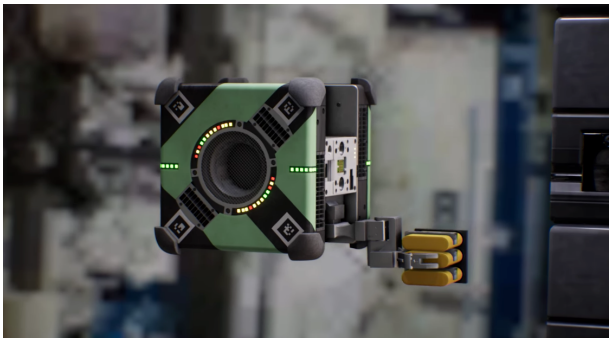

Figure 1. : A rendering of Astrobee in the International Space Station showing the perching arm deployed.

The Astrobee Robot Software[1] provides localization and mobility control, orchestrates the various software and hardware components, supports multiple modes of operation and accommodates external researchers' – referred to as *guest scientists* – software. The Robot Operating System (ROS) is used for all internal communication while the Data Distribution Service (DDS) handles external communication. The Astrobee Robot Software also provides a powerful Gazebo-based simulator and tools for building and deploying to embedded processors.

Astrobee flight software takes advantage of technologies seen in terrestrial and aerial mobile robots, but also addresses challenges unique to the space environment:

- The **lack of gravity** and **unconventional interior** of the ISS preclude localization techniques that exploit GPS, gravity, and prior maps from being used.
- Even though ISS can provide high bandwidth communications, the software must be resilient to frequent **Loss Of Signal (LOS) and high latency**.
- Although certified safe at the hardware level, flight software must maximize **software reliability** to minimize crew interventions.
- Astrobee robots are **not serviceable by an expert** on orbit, and thus the system needs to support complete remote software maintenance and introspection.

[1] Available online: `https://github.com/nasa/astrobee`

This paper is organized as follows. We first describe the Astrobee robotic platform, then the software architecture and components, and finally the lessons learned. The Astrobee Robot Software has been released under the Apache-2 Open Source license. This enabled early access by future guest scientists. We will describe how three existing projects are using the software in the lessons learned section.

1.1 Related Work

Flying robots on earth have to overcome the strong gravity and thus have quite different constraints than free-flyers on the ISS. Multi-rotors drones however tackle some similar problems to Astrobee in terms of packing computing performance in a lightweight package and localizing in an unstructured environment.

The JAXA Intball has demonstrated remote camera capabilities with success [2]. IntBall is much smaller than Astrobee, contains a miniaturised propulsion and control system and can be sent motion commands from the ground. CIMON from DLR [3] is about Astrobee size, with a focus on delivering AI capabilities to assist Astronauts. Both IntBall and CIMON are currently designed to be permanently crew tended and limited to operate in a single module. These robots also use visual markers to localize (Intball) or seed localization (CIMON).

NASA's Robonaut [4] is not a free-flyer, but another complex robot residing in ISS, which bears many similarities to Astrobee in terms of software. Robonaut uses ROS, is deployed on multiple computers and relies on a vision system to perform tasks like grasping handrails.

The SPHERES free-flyers robots[5] are one of the most used payloads on the ISS and have supported numerous micro-gravity research experiments over the last ten years. The SPHERES software is focused primarily on control problems and limited by the microcontroller used. Our group has previously developed SmartSPHERES [6], a SPHERES payload based on a smart phone, demonstrating what capabilities are possible with a more powerful computing platform.

2 THE ASTROBEE PLATFORM

2.1 Motivation

The Astrobee project is inspired by the AERCam [7] and PSA [8] space free-flyers. Astrobee is often considered a SPHERES replacement since it uses rechargeable batteries (no consumables that need to be up-massed), offers a modern computing platform, and does not require crew to perform its missions. Three typical scenarios envisioned for Astrobee show that a fundamentally new design is required:

Mobile Camera: Astrobee allows ISS Flight Controllers to monitor crew operation with an HD video stream.

Autonomous Survey: Astrobee carries a payload (i.e., an air quality sensor) and can perform a multi-hour survey of some environmental parameters in ISS.

Research Hosting: Guest scientists conduct experiments in microgravity (i.e., Human Robot Interaction experiment) with a software and/or hardware payload on Astrobee.

The software features required to support these scenarios include:

- Localization throughout the U.S. Orbital Segment (USOS) of the ISS without extra infrastructure.
- Precise motion planning and execution with safeguards against known and unknown obstacles.
- Control and monitoring from the ground with resilience to loss of communication.
- Support for multiple control modes, including remote teleoperation, autonomous plan execution and on-board control by guest science software.
- Dock (and undock) autonomously to recharge and for wired communications.
- Perch autonomously on an ISS handrail for power saving and pan/tilt camera.
- Manage guest science software, hardware payloads, and user interface components.
- Provide a control station application to command and monitor the Astrobee robot remotely. The control station is not addressed in this paper.

2.2 Hardware

The Astrobee project will deliver three free-flyer robots and one dock to the ISS. The dock is equipped with two berths on which the Astrobee robot can mate to obtain power and Ethernet connectivity. The dock is also equipped with an embedded computer that monitors the berths and Astrobee's status while in hibernate mode. Finally, the dock computer facilitates software updates on the robot.

Each Astrobee is equipped with three smartphone class ARM processors communicating through an Ethernet network as shown in Fig. 4. The "Low Level Processor" (LLP) runs the pose estimator, the motion control loop and communication with key hardware like the Inertial Measurement Unit (IMU) or the Power Manager. The "Mid Level Processor" (MLP) is responsible for computationally intensive computer vision and mapping algorithms. Thus the MLP is connected to two color imagers

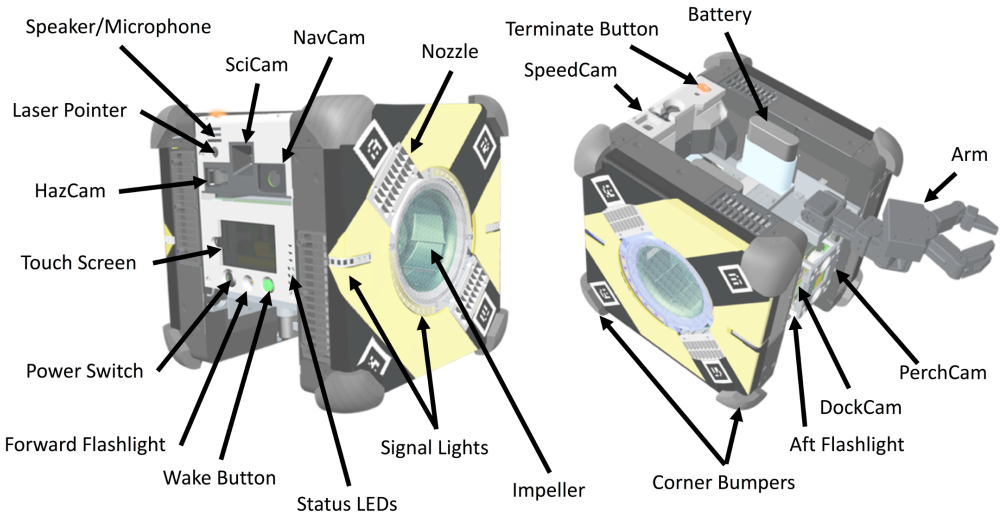

Figure 2. : Rendering of an Astrobee Flight Unit equipped with its perching arm, with nomenclature of its main external hardware components.

(NavCam and DockCam) and two depth imagers (HazCam and PerchCam). The MLP is also responsible for communication with the ground and the fault management system. The "High Level Processor" (HLP) is dedicated to **guest science** applications developed by external academics and commercial researchers. The HLP also manages the human-robot interaction devices like the SciCam, Touchscreen, Speaker and Microphone. The locations of the sensors and actuators are shown on Fig. 1.

The Astrobee robot moves using 12 nozzles placed on two propulsion modules that sandwich the core computing module. The nozzles push out air that has been pressurized in each module's plenum by an impeller. The design of using only two large impellers minimizes the sound level, but causes other control challenges like a high latency to acquire the desired pressure at startup.

Finally, the Astrobees are normally equipped with a perching arm allowing them to grasp ISS handrails and transform the free-flying robot into a remote pan-tilt camera.

3 SOFTWARE ARCHITECTURE AND COMPONENTS

The Astrobee Robot Software relies on a modern approach of software development: the system is composed of a set of distributed, loosely coupled and highly cohesive components. This is implemented in practice by ˜46 ROS nodelets[2] running on three CPUs. The dependencies between nodes are always kept to a minimum by defining clear responsibilities for each sub-systems. Fig. **??** shows the distribution of the main components on the Astrobee processors. The Astrobee Robot Software is written in C++ for its high-level constructs, ROS support, and high performance. The LLP and MLP processors run Linux (Ubuntu 16.04 LTS) because of its widespread use and the availability of software packages (particularly ROS) for this distribution. Astrobee does not use any Linux real-time kernel extension. The tightest control loop (GN&C) runs on the LLP at 62.5Hz with acceptable jitter. Motor control is done with dedicated microcontrollers. The code is developed on personal computers with the same version of the OS. Algorithms are tested on developer computers in the simulator, then the code is cross-compiled for the target ARM platform. The HLP processor, however, runs Android (Nougat 7.1) because it is the only OS supporting some key hardware for Astrobee (the HD camera, Video Encoder and Touchscreen). Android allows for the encapsulation of guest science software for the HLP as Android Packages (APKs), avoiding custom deployment and management methods.

3.1 Middleware

The interactions between the various Astrobee software components rely solely on the ROS communication framework. ROS messages are used for data distribution, ROS service calls for simple requests (i.e., turning a light on/off) and ROS actions for control of non atomic operations (i.e., a motion that has a certain duration). The use of ROS nodelets and judicious grouping of large data

[2] From the ROS manual: "nodelets are designed to provide a way to run multiple algorithms on a single machine, in a single process, without incurring copy costs when passing messages intraprocess". The Astrobee Robot Software nodelets are grouped into ˜14 ROS nodes.

producers and their consumers permit zero-copy message passing for high efficiency. For example, the cameras drivers and the vision algorithms transfer images without going through the network layer by running as part of the same node. The nodelet concept is so essential to the Astrobee Robot Software that we developed our own nodelet specialization that offers common functionalities. The Astrobee nodelet encapsulates a unified naming scheme, lifecycle control for the nodelet, a heartbeat mechanism, and fault management. These features, which are typically expected of reliable space software, are thus automatically available for the whole system.

Communication between one Astrobee and the outside world (control stations or other Astrobees) does not rely on ROS. Instead, RAPID [9] and the Data Distribution Service (DDS) are used. RAPID has been used previously to both control robots on Earth from space ([10]) and robots in space from Earth ([6]). A middleware with powerful Quality Of Service (QoS) capabilities combined with sufficient robot autonomy allows for reliable teleoperation over degraded networks [11]. Compared to ROS, DDS allows for a much finer control of the bandwidth usage and offers delay tolerance through manipulation of QoS per topic.

3.2 API

Astrobee offers multiple control modalities that can be mixed during a single session.

Teleoperation from a control station where commands are sent from either ground control or an ISS laptop.

Autonomous plans where the onboard Executive controls the execution of a sequence of actions.

Guest Scientists Applications where guest software on the HLP controls Astrobee behavior.

As described in the previous section, the communication between on-board components relies on ROS. The Astrobee Robot Software maximizes the re-use of ROS standard messages to leverage other tools offered in the ROS ecosystem. However, Astrobee has numerous non-conventional subsystems, hence a set of custom messages has been crafted. The combined standard and custom ROS message definitions define the internal Astrobee API.

Every command coming from outside the core software (control station or HLP guest science application) is filtered by the Executive as shown in Fig. 3. A JSON-based command dictionary describes all commands and their arguments. This command dictionary is processed by translators that create 1) set of DDS based commands (using the RAPID framework) for external control, and 2) a Java API for guest science control. This system provides a unified API to Astrobee users.

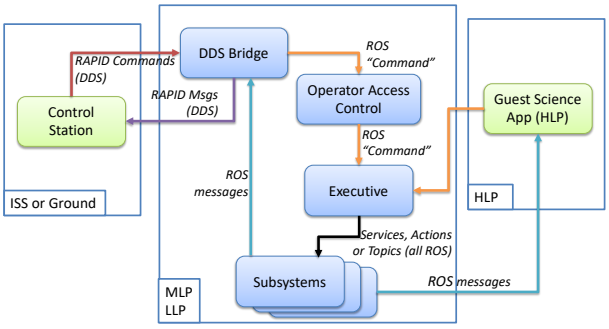

Figure 3. : Unified Command and Telemetry within Astrobee core system (LLP and MLP) and external applications (Control Station and/or HLP).

3.3 Infrastructure

Astrobee software and firmware will need to be updated while aboard the ISS without Astronaut support or external equipment. All the microcontrollers run a custom bootloader allowing for a safe firmware update from the host processor. The HLP processor running Android is updated via network from the MLP using standard Android tools (fastboot). The MLP and LLP processors have a more complex OS update mechanism using a rescue partition that is updated via wired network from the Dock. The Linux image creation for the ARM processors is based on a set of custom tools that guarantee consistency between the development OS (running on laptops) and the minimal OS running on the embedded processors. For reliability purpose, the Linux OSes use a read-only file-system. All the processors, but HLP, are using the exact same OS image, and the customization per robot/processor is injected with an overlay partition containing robot specific information (like IP addresses). Finally, the Astrobee free-flyers exhibit a complex network shown in Fig. 4 that requires elaborate routing and firewalls.

3.4 Simulator

An essential part of the Astrobee Robot Software is the simulator. The simulator serves both the Astrobee developers and guest scientists to test software before deploying it on the physical robot. The simulator simulates the robot propulsion system, perching arm, color and depth cameras, IMU, and environment. The Astrobee simulator is based on the Gazebo dynamic simulator with a number of custom plugins. For every software driver (code communicating with a hardware device), a Gazebo plugin has been created. For example, ROS messages commanding the propulsion system are consumed by a custom Gazebo plugin that convert commands into resulting forces applied to the robot. This plugin uses a high fidelity model of the blower/nozzles system. The resulting

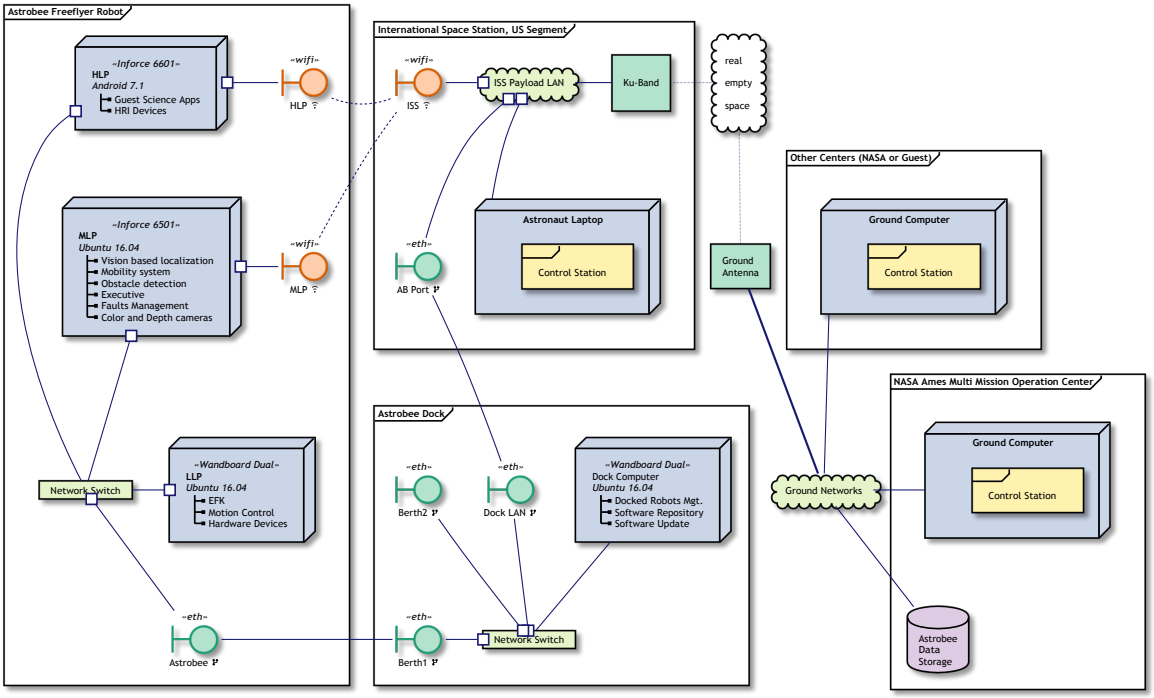

Figure 4. : Connectivity of the Astrobee robot with the Astrobee Dock and ISS network, including communication paths to the ground.

forces are used by Gazebo to compute the dynamic motion of the robot. An ISS CAD model provides a visual environment for the simulator. Camera models generate images that can be consumed by the robot nodes.

The simulator allows the control system to run at its target rate of 62.5Hz and simulates realistic localization measurements for the localization algorithms. The simulator can run either at wall clock time, or as fast as the simulation computer permits (10 times speedup on a desktop computer with good graphic card). Our architecture allows users to transparently run all components either in simulation or on the Astrobee, or as a mix of simulation with hardware processor(s) in the loop.

3.5 Software Components

Fig. 5 show the main software components distributed on the three processors.

3.5.1 Management

The **executive** filters incoming commands as a function of their sources and the current operating mode (teleoperation, plan execution or guest science). DDS commands issued from the ground are transformed into ROS commands by the **DDS bridge** (see Fig. 3). The DDS bridge also subscribes to all the ROS messages useful for real time monitoring, and transmits them to the ground at a controllable rate as DDS messages.

The Astrobee Robot Software uses a distributed fault management framework. The **system monitor** collect faults information from every node and responds according to a fault table. The distribution of faults keeps the responsibility for fault detection and analysis together within the subsystem concerned.

The Astrobee Robot Software provides a common framework to support and manage Guest Science applications developed by external users. The **guest science manager** works in concert with the executive to control the life-cycle of guest science applications. The **guest science library** seamlessly integrates the guest science manager and the guest science APKs. The Astrobee Java API library encapsulates the command dictionary and allows guest science APKs running on Android to harmoniously communicate with the Astrobee flight software.

3.5.2 Mobility

The **mobility** subsystem is responsible for the safe motion of the free-flyer. It executes desired trajectories and ensures that Astrobee is respecting a set of tolerances. Trajectories can be synthesized on the ground using the control station, or dynamically created onboard using trajectory **planners**. The system uses a plugin-architecture to switch between path planners. For example, the default

path planner generates trajectories respecting the same constraints (straight translations and facing forward) as the control station. A second planner creates smooth, optimal trajectories around obstacles [12] and can be selected at runtime without affecting the rest of the system.

The mobility subsystem is also responsible for obstacle detection. It creates an octree-based occupancy map from the Hazcam depth camera. It then validates trajectories in this map. The map is augmented with preconfigured keep-out zones that can be defined for each mission scenario.

3.5.3 Localization

Astrobee's **localization** relies on a pose estimator that integrates measurements from a variety of sources depending on the robot's localization mode:

1. The general purpose localization uses the forward facing camera NavCam with a wide 120 field of view. Visual features from a pre-built map of the ISS allow a position error lower than 5 cm in the nominal case.
2. When docking, Astrobee uses the back pointing camera DockCam with a 90 field of view. Artificial markers positioned on the dock allow a reduction in the localization error to a sub-centimeter level.
3. For perching on handrails, Astrobee uses the time of flight camera PerchCam. 3D features are extracted from the depth image and fed to the pose estimator.[13].
4. A visual odometry algorithm that keeps track of features across the most significant 16 frames contribute to the stability of the localization.

See [14] for further details about Astrobee's localization algorithm.

3.5.4 Control

Astrobee's motion control subsystem runs on the LLP. By limiting the number of processes running on this processor, the close loop control has a jitter lower than the accepted tolerance. The control subsystem is developed using Simulink. C code is auto-generated with the Simulink buses being mapped to input/output data structures. The auto-generated code is wrapped into ROS nodes that integrate seamlessly with the rest of the system. The parametrization of the control models uses the same configuration files that are used natively in through the rest of the Astrobee Robot Software. This method allows Astrobee to capture the expertise of control domain experts while integrating well into the ROS ecosystem.

The control subsystem includes three main components that are connected by ROS interfaces:

1. An Extended Kalman Filter (**EKF**) implements the pose estimator described above.
2. The Control (**CTL**) algorithms realize the closed loop control of Astrobee.
3. The Force Allocation Module (**FAM**) translates forces and torques produced by CTL into actual propulsion nozzle commands.

This decomposition allows advanced users to replace a single component (typically CTL) with their own algorithms, while still benefiting from the other components.

4 LESSONS LEARNED

4.1 Software infrastructure cost

Many aspects of the Astrobee Robot Software, like the localization methodologies or trajectories planning, are innovative in the field of robotics and thus involve significant research. Furthermore, the software infrastructure itself is key to a successful deployment and comes with its own innovations to accommodate the distinctive Astrobee platform. Because of design constraints and hardware availibility, Astrobee computing elements are not identical. Only the LLP and the Dock Computer are using the same "System On Module" (SOM[3]). The MLP and HLP are two other SOMs from a different vendor. Even though the Linux distributions used are the same, numerous Linux kernel customizations are required for each SOM. In addition, Astrobee runs both Linux and Android which forces the team to acquire expertise with two development environments and create different tools to maintain each system. This heterogeneous hardware platform taxes software development effort. Finally, the cell phone technologies that allow Astrobee to provide the desired capabilities in this form factor introduce additional problems. The pace of the cell phone industry is much faster than the pace of a project like Astrobee which spans over more than 3 years: products like SOMs or HD cameras become obsolete before the project is mature enough to commit acquisition in sufficient quantities. This forces software adaptations costing time, without mentioning likely hardware changes.

4.2 Platform optimization

The use of an embedded computing platform for computationally intensive algorithms necessitates extracting all the potential from the hardware. For example, despite Simulink generated code optimizations, the performance

[3] The SOM include CPU, GPU, memory and peripheral like I2C, USB and network adapters.

Function	Coder	Eigen	Improvement
of_residual_and_h	87.0 s	2.0 s	98% (43x)
delta_state_and_cov	22.5 s	3.5 s	84% (6x)
covariance_multiply	18 s	< 2 s	89% (~10x)

Table 1. : Performance of some time consuming functions using the Mathworks Simulink **Coder** compared to hand written code using the **Eigen** library. The times are totaled over the run of a recorded data set.

of some functions have been drastically improved by using Eigen [15] efficiently, as show in Table 1. This is possible thanks to the efforts that Eigen developers invested into optimizing the library with the accelerated NEON instruction set of the ARM processors.

The MLP SOM also offers a Graphical Processing Unit (GPU) that could be used to optimize further some Astrobee code. Image processing algorithms are typically well suited for this type of hardware acceleration. Unfortunately, drivers for this GPU are not available under Linux at this time. However, in the future, we will consider improving the obstacle detection and mapping algorithms using the GPU.

4.3 Open Source

Open-sourcing the Astrobee Robot Software was a requirement from project inception. Our group is a pioneer from within NASA in terms of making our software projects available to outside communities via Open Source licenses. The Astrobee Robot Software is classified as Class-C[4] , non-safety critical software and thus is subject to the NASA 7150.2B requirements [16]. While less flexible than academic institutions, the rigorous NASA Open-Source process reviews projects to release with respect to these requirements. Releasing the Astrobee Robot Software under an Open Source license allows all existing or potential users of the Astrobee platform to access the code without putting in place any licensing contracts with NASA.

Making Astrobee Robot Software open source brings another benefit by fostering synergies with other open-source projects. For example, rather than pure "users" of ROS and Gazebo, our project now contributes ideas and code to the community and software patches are more easily exchanged.

The Astrobee Robot Software could not have been developed by a small team without extensive use of open source software. Table 2 lists a subset of key software packages which saved numerous man-months for the project.

[4] NASA define class C as: "Mission Support Software or Aeronautic Vehicles, or Major Engineering/Research Facility Software"

ROS	Robot Operating System
Gazebo	C++ robot simulator
OpenCV	Open Source Computer Vision
Eigen	C++ template library for linear algebra
SURF	Speeded-Up Robust Features
BRISK	Binary Robust Invariant Scalable Keypoints
DBoW2	bag-of-words library for C++

Table 2. : Subset of Open Source packages used by the Astrobee Robot Software. The full list of dependencies is available at `https://github.com/nasa/astrobee`

4.4 External Users

Three selected use cases illustrate how existing Astrobee Robot Software users are interfacing with the system.

4.4.1 Pure Simulation

The MIT ZeroRobotics project [17] is transitioning from its use of the SPHERES free-flyer to the Astrobee free-flyer. In the first phase of the transition, ZeroRobotics will develop a game framework using the Astrobee simulator and the Astrobee Java API without an Android system.

4.4.2 Controller Algorithms

Collaboration with the Naval Postgraduate School in Monterey is leading to control algorithms to perform aerial maneuvers using a manipulator. In this scenario, the researchers replace the Simulink control with their own model. They also contributed an additional Gazebo plugin to handle the perching mode of Astrobee.

4.4.3 Hardware payload

The REALM team at NASA JSC is developing a radio frequency identification (RFID) system for autonomous logistic management [18]. REALM-2 will become the first hardware payload on Astrobee. For this scenario, the REALM team is developing a guest science application running on the HLP. To develop their system before the final Astrobee becomes available, the team acquired a HLP development kit which they can connect their hardware payload to. The HLP development kit can be connected to a computer running the Astrobee simulator. This setup offers high fidelity testing with hardware in the loop.

5 CONCLUSION

The Astrobee Robot Software manages a powerful but complex hardware platform. Adopting a dual middleware approach allows us to take advantage of ROS onboard while respecting space network constraints. We hope in the future to transition to a unified middleware with ROS-2 using DDS as a transport layer. The software infrastructure for the Astrobee project is key for space deployment and maintenance and represents a significant fraction of the effort. Embracing ROS provides both a reliable distributed system and access to tools like Gazebo that are part of the ROS ecosystem. We demonstrated that a properly crafted ROS based software system delivers a beneficial solution for an embedded robotic platform. The Astrobee robot prototype has been operated numerous hours without being hampered by the lack of real time operating system. The software components developed enable:

- remote and on-board commanding, execution and monitoring of the robot
- support for guest science applications with a unified API
- markerless localization and navigation in a ISS like environment
- flexible management of hardware resources with ROS enabled drivers
- faster than real time simulation with an abstraction of the hardware drivers

The Astrobee Robot Software has reached a level of maturity that has allowed its release by NASA as an open source project. This permitted an easy access to several guest scientists developing experiments for the Astrobee platform on ISS. Researchers can interact with the Astrobee Robot Software at various level depending on their need. As the Astrobee robots are commissioned aboard ISS late this year, we hope that our software will open opportunities for new exciting research.

REFERENCES

[1] Trey Smith et al. "Astrobee: A new platform for free-flying robotics on the international space station". In: *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS)*. 2016.

[2] JAXA. *First disclosure of images taken by the Kibo's internal drone "Int-Ball"*. URL: http://iss.jaxa.jp/en/kiboexp/news/170714_int_ball_en.html (visited on 07/14/2017).

[3] DLR. *CIMON - the intelligent astronaut assistant*. URL: http://www.airbus.com/newsroom/press-releases/en/2018/02/hello--i-am-cimon-.html (visited on 03/02/2018).

[4] Julia Badger et al. "ROS in Space: A Case Study on Robonaut 2". In: *Robot Operating System (ROS)*. Springer, 2016, pp. 343–373.

[5] Swati Mohan et al. "SPHERES flight operations testing and execution". In: *Acta Astronautica* 65.7-8 (2009), pp. 1121–1132.

[6] Mark Micire et al. "Smart SPHERES: a Telerobotic Free-Flyer for Intravehicular Activities in Space". In: *AIAA SPACE 2013 Conference and Exposition*. 2013, p. 5338.

[7] Trevor Williams and Sergei Tanygin. "On-orbit engineering tests of the AERCam Sprint robotic camera vehicle". In: *Spaceflight mechanics 1998* (1998), pp. 1001–1020.

[8] Gregory A Dorais and Yuri Gawdiak. "The personal satellite assistant: an internal spacecraft autonomous mobile monitor". In: *Aerospace Conference, 2003. Proceedings. 2003 IEEE*. Vol. 1. IEEE. 2003, pp. 1–348.

[9] Hans Utz et al. *The Robot Application Programming Interface Delegate Project*. URL: http://robotapi.sourceforge.net/index.html (visited on 05/01/2013).

[10] Maria Bualat et al. "Surface telerobotics: development and testing of a crew controlled planetary rover system". In: *AIAA Space 2013 Conference and Exposition*. 2013, p. 5475.

[11] Lorenzo Flückiger and Hans Utz. "Service Oriented Rbotic Architecture for space robotics: design, testing, and lessons learned". In: *Journal of Field Robotics* 31.1 (2014), pp. 176–191.

[12] Michael Watterson, Trey Smith, and Vijay Kumar. "Smooth trajectory generation on SE (3) for a free flying space robot". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE. 2016, pp. 5459–5466.

[13] Dong-Hyun Lee et al. "Handrail detection and pose estimation for a free-flying robot". In: *International Journal of Advanced Robotic Systems* 15.1 (2018), p. 1729881417753691.

[14] Brian Coltin et al. "Localization from visual landmarks on a free-flying robot". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE. 2016, pp. 4377–4382.

[15] *Eigen Overview*. URL: http://eigen.tuxfamily.org/index.php?title=Main_Page#Overview (visited on 03/20/2018).

[16] NASA. *NASA Software Engineering Requirements, NPR 7150.2B*. URL: `https://standards.nasa.gov/standard/nasadir/npr-71502` (visited on 11/19/2014).

[17] Sreeja Nag, Jacob G Katz, and Alvar Saenz-Otero. "Collaborative gaming and competition for CS-STEM education using SPHERES Zero Robotics". In: *Acta astronautica* 83 (2013), pp. 145–174.

[18] Patrick W Fink et al. "Autonomous Logistics Management Systems for Exploration Missions". In: *AIAA SPACE and Astronautics Forum and Exposition*. 2017, p. 5256.

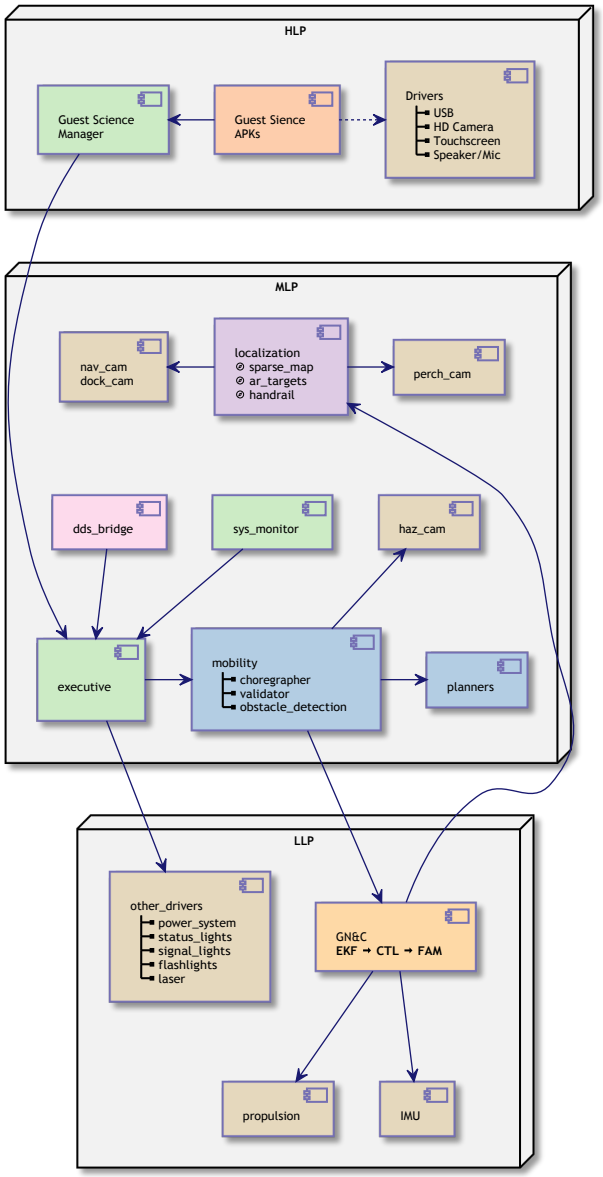

Figure 5. : The main software components running on Astrobee. The components on this diagram represent logical groupings normally composed of multiple ROS nodelets. The arrows indicate dependencies, not flow of information.