(Preprint) AAS 25-801

A MESSAGE-PASSING SIMULATION FRAMEWORK FOR GENERALLY ARTICULATED SPACECRAFT DYNAMICS

Juan Garcia-Bonilla[*] and Hanspeter Schaub[†]

This paper presents a model-based, message-passing simulation framework for spacecraft systems with generally articulated structures. Traditional astrodynamics simulations often assume rigid, single-body spacecraft, limiting their applicability to modern systems with robotic arms, deployable arrays, and other moving parts. The proposed framework decomposes kinematics, forces, and dynamics into compartmentalized models that exchange information via messages. This enables intuitive and rapid simulation setup without requiring expertise in complex dynamics algorithms. By encapsulating functionality within discrete models, the framework enhances usability, robustness, and reusability across missions. The resulting implementation integrates into the Basilisk simulation architecture with the MuJoCo dynamics engine, combining accurate orbital propagation with advanced multi-body dynamics. An example scenario with code excerpts and numerical results is presented to demonstrates the flexibility, ease of use, and rapid prototyping capability of this architecture.

INTRODUCTION

Simulation tools are critical in modern astrodynamics research and mission development. As spacecraft systems become more complex, simulation toolkits must become more accurate, extensible, and user-friendly to support all stages of these mission: from concept to operations. Rapid and easy to setup numerical simulation of complex spacecraft dynamics enables engineers to assess performance, validate control strategies, and reduce risk by predicting a spacecraft's behavior in its environment, without the need of expensive hardware testing.[1,2]

Numercial simulation is particularly vital for the design, testing, and verification of guidance, navigation, and control (GNC) algorithms. These algorithms must be robust to environmental disturbances, modeling uncertainties, and the coupling between spacecraft states. High-fidelity simulation environments enable early issue detection and faster design iteration.[3,4] During operations, simulation supports mission rehearsals and operator training, helping teams explore contingencies and make informed decisions.[5]

Post-launch, simulation supports telemetry analysis by enabling physically consistent reconstructions of vehicle behavior. It also plays a growing role in learning-based applications in space systems, such as in perception algorithms or control policies.[6] Because real-world spaceflight data

[*]Ph.D. Student, Department of Aerospace Engineering Sciences, University of Colorado Boulder, 431 UCB, Boulder, CO 80309, USA
[†]Ann and H. J. Smead Department of Aerospace Engineering Sciences, University of Colorado Boulder, 431 UCB, Boulder, CO 80309, USA

(a) "Haven-2" space station proposal[*].

(b) Dextre two-armed robot attached to the ISS[†].

(c) MASCOT-2 asteroid lander proposal[‡].

(d) Starliner docking with the ISS[§].

Figure 1: Examples of spacecraft configurations that benefit from multi-body dynamics modeling.

is often sparse or unavailable, high-fidelity simulators serve as critical sources of ground truth for training and validating these models.

Historically, spacecraft in astrodynamics have been modeled as single rigid bodies, with either six degrees of freedom or as point masses. While this simplification is effective for many analyses, it fails to capture the internal dynamics of real space vehicles. Spacecraft often include moving subsystems such as spinning reaction wheels, articulating solar arrays, deploying antennas, or robotic arms that manipulate payloads. These elements introduce dynamic coupling effects, exchanging momentum, altering inertial properties, and thus impacting operational control of the spacecraft.[7] Accurate simulation of these interactions requires a multi-body dynamics formulation, where each component is represented as a distinct body connected through joints allowing translation, rotation, or both.

The need for a general multi-body spacecraft dynamics simulation grows even more important in light of emerging space architectures. Concepts such as in-orbit assembly, modular space sta-

[*]"Starliner makes first docking with ISS on OFT-2 mission", `https://www.nasaspaceflight.com/2022/05/oft-2-docking/`, Accessed: 2025-07-19.

[†]"Landing on an asteroid", `https://www.esa.int/ESA_Multimedia/Images/2016/02/Landing_on_an_asteroid`, Accessed: 2025-07-19.

[‡]"NASA Image and Video Library: View of Dextre and CTC2", `https://images.nasa.gov/details/iss027e016182`, Accessed: 2025-07-19.

[§]"Vast Announces Haven-2, Its Proposed Space Station Designed To Succeed The International Space Station (ISS)", `https://www.vastspace.com/updates/vast-announces-haven-2-its-proposed-space-station-designed-to-succeed-the-international-space-station-iss`, Accessed: 2025-07-19.

tions, satellite servicing, and autonomous robotic operations require accurate modeling of systems composed of multiple interacting or articulated bodies (Figure 1).[8–10] Swarms, manipulators, and expandable platforms are all examples of multi-body systems. To ensure feasibility, safety, and effective control of these architectures, simulation tools must adapt to enable high fidelity modeling of these systems.

Although multi-body modeling is well-developed in terrestrial robotics, few multi-body dynamics tools are tailored for astrodynamics and spacecraft GNC. Robotics simulators like Gazebo, MuJoCo, and Simscape Multibody offer sophisticated joint modeling, contact dynamics, and efficient dynamics solvers.[11–13] However, these tools lack orbital mechanics, ephemerides, and space-relevant environmental models.

Conversely, space mission simulation platforms such as GMAT, STK, and FreeFlyer provide accurate orbital and attitude propagation,[14–16] but generally assume rigid-body dynamics and lack general support for articulated structures. Some tools, such as JPL's Dshell-DARTS, include multi-body spacecraft modeling capabilities,[1] but they are closed-source and not accessible to the broader community.

Basilisk is a modular, open-source simulation toolkit for astrodynamics and GNC.[2] It supports accurate orbital and attitude propagation with real-time performance. Its current multi-body capabilities are very fast to numerically evaluate and to setup in a simulation, but limited in the spacecraft configuration space that can be modeled due to its use of the Backsubstitution Method (BSM). BSM lacks support for branching kinematic trees, offers limited flexibility in combining joint types, and modeling coupled multi-vehicle scenarios remains challenging.[17–19] Critically, extending these capabilities requires manually deriving and implementing complex equations of motion, which is a time-consuming and analytically challenging process. The focus of BSM has been rapid numerical simulations of classic individual spacecraft and not the more complex space stations, landers, or on orbit assembly. The BSM benefits begin to vanish when complex spacecraft configurations, such as those including branching, are included.

This paper introduces a general-purpose simulation paradigm for multi-body spacecraft dynamics that leverages Basilisk's modular message-passing framework. The MuJoCo engine is integrated into Basilisk as an alternate dynamics engine for efficient, general-purpose multi-body dynamics, enabling Basilisk users to model articulated space systems without deriving equations manually. This architecture's effectiveness is demonstrated through an illustrative usecase featuring code, models, and numerical results.

MULTI-BODY DYNAMICS

The configuration of a multi-body system is described by its generalized coordinates, denoted $\boldsymbol{q}$. These coordinates represent the minimal set of parameters needed to fully define the system's state. For a point mass, this may include only its Cartesian position and velocity, such that $\boldsymbol{q} \in \mathbb{R}^6$. More complex systems, such as spacecraft with articulated arms, require additional parameters, including the pose (position and attitude) and twist (linear and angular velocity) of a base body, along with the angular positions and velocities of each joint. In general, $\boldsymbol{q} \in \mathbb{R}^n$, where n depends on the system's topology and degrees of freedom.

To simulate the system's behavior over time, we propagate the time derivative of the generalized coordinates:

$$\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = f(\boldsymbol{q}, \boldsymbol{\tau}(t, \boldsymbol{q})) \tag{1}$$

Here, $\boldsymbol{\tau}(t, \boldsymbol{q})$ represents the generalized forces acting on the system. These include externally applied forces and torques (e.g., from thrusters or actuators), internal joint torques (e.g., from motors), and environment-induced forces such as gravity, drag, or contact interactions.

Evaluating the dynamics function $f(\boldsymbol{q}, \boldsymbol{\tau})$ typically involves three key steps:

1. **Forward kinematics:** Compute the position, orientation, linear velocity, and angular velocity of all relevant frames based on the current configuration $\boldsymbol{q}$.
2. **Force and torque evaluation:** Determine all generalized forces $\boldsymbol{\tau}$ acting on the system, including environmental effects, control inputs, and internal joint forces.
3. **Forward dynamics:** Use the computed forces and the system's inertial properties to evaluate $\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}$.

Forward kinematics maps the generalized coordinates $\boldsymbol{q}$ to a set of physically meaningful quantities, such as the pose (position and orientation) and twist (linear and angular velocity) of frames of interest. This computation depends only on the system's topology and current configuration. It does not involve external inputs or dynamic effects.

Generalized forces, by contrast, arise from the physical environment or control actions. These forces must be explicitly modeled by the simulation engineer to reflect realistic mission conditions. Accurate force modeling is critical for analyzing the effects of environmental disturbances, actuator performance, and GNC algorithm behavior.

Forward dynamics then translates these generalized forces into time derivatives of the generalized coordinates $\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}$. Numerous open-source libraries efficiently perform this task.[11, 12]

Among the three simulation steps described above, forward kinematics and dynamics can be handled by well-established, system-agnostic algorithms. As such, the argument is made that a simulation framework should ideally abstract these computations away from the user. In contrast, defining and modeling the forces acting on the system is inherently application-specific and remains the primary responsibility of the simulation engineer. These force models encode environmental interactions and control inputs, making them essential for assessing system performance. A well-designed toolkit should therefore simplify force specification while automating low-level dynamics computation.

A MESSAGE-PASSING DYNAMICS SIMULATION PARADIGM

Basilisk is an open-source astrodynamics simulation framework built on a modular, message-passing architecture.[20] In this architecture, discrete models represent components such as sensors, actuators, or flight software, and they communicate by publishing and subscribing to structured messages. This design promotes code reuse, modularity, and ease of testing through encapsulation of behavior with clear and strict interfaces.

However, Basilisk's dynamics engine did not adopt this message-driven structure, but it implemented the BSM dynamics engine instead. In particular, force evaluation, kinematics, and dynamics computations remain tightly coupled, which can make maintenance and extension of these capabilities complicated. To address this, a new simulation paradigm is proposed that applies Basilisk's message-based design to a general multi-body system dynamics solution. While Basilisk serves as the motivating example, the principles introduced here are general and may be applied in other simulation environments.

This paradigm decomposes the computation of $\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}$ (Eq. 1) into three types of models that communicate via messages:

1. A **forward kinematics model** computes the pose and twist of selected frames on the spacecraft given the generalized coordinates $\boldsymbol{q}$.
2. A set of **dynamics models** compute generalized forces $\boldsymbol{\tau}$ acting on the system's bodies and joints using kinematic and environmental information.
3. A **forward dynamics model** combines the generalized forces and the system's inertial properties to compute the generalized coordinates' derivative $\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}$.

For example, a model interested in solar power estimation may subscribe to the pose of a solar array frame to compute its orientation with respect to the Sun. For another example, a control model may use the twist of a joint frame to compute a torque command. Each of these computations occurs within its own self-contained model, and receives its inputs and communicates its outputs through messages.

Both forward kinematics and forward dynamics are well-understood problems with efficient, general-purpose solutions. Numerous third-party libraries exist, particularly in the robotics community, that implement these algorithms robustly and efficiently. Because these computations are largely agnostic to the specific spacecraft or mission scenario, they should be treated as generic modules that are delegated to external solvers whenever possible.

Rather than re-implementing these algorithms within each simulation toolkit, a better approach is to integrate mature libraries that already handle the complexities of multi-body kinematics and dynamics. This not only improves simulation robustness and development speed but also grants access to advanced capabilities. For example, MuJoCo is one such open-source solver that supports articulated rigid bodies, joint constraints, collision detection, contact dynamics, and flexible body modeling.[12] While MuJoCo[*] is used in our implementation, the proposed simulation paradigm is solver-agnostic. Any engine capable of evaluating forward kinematics and dynamics from generalized coordinates and forces could be similarly integrated into the architecture.

Once kinematics and dynamics are handled by generic solvers, the main task of the simulation engineer becomes defining the forces acting on the system. In our paradigm, these forces are modeled using independent, reusable modules that operate on kinematic and other simulation inputs and produce force or torque outputs. Each model is responsible for computing a specific subset of the total forces, for example, the gravity force, thruster output, or aerodynamic drag.

This design has several advantages:

- **Reusability:** Models are written against generic input and output messages, making them applicable across different vehicles.
- **Clarity:** Each model implements a physically meaningful behavior, which improves transparency and debugging.

[*]MuJoCo, Advanced Physics Simulation, `https://mujoco.org/`, Accessed: 2025-07-19.

- **Composability:** Complex force environments are created by combining simple, well-tested modules.

To illustrate, consider a gravity model that takes as input the position of a body's center of mass along with the location of celestial bodies, and outputs a force vector representing the gravitational attraction. Similarly, a solar radiation pressure model for a flat panel might use the panel's surface normal, centroid position, and the relative direction of the Sun to compute the resulting force along the normal. In both cases, the models operate independently, reading kinematic data via input messages, performing a focused physical computation, and publishing their outputs as generalized forces to be consumed by the dynamics engine.

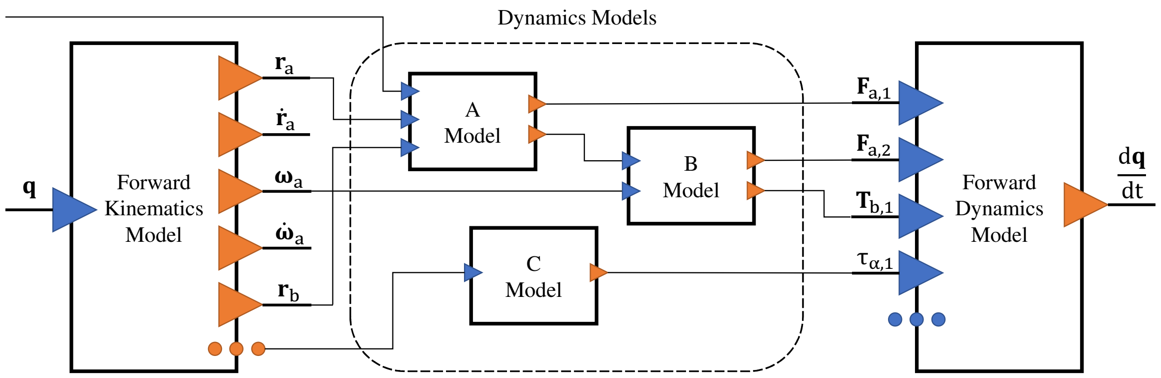


Figure 2: Schematic of the message-passing, model-based simulation architecture. Each model (black square) communicates via messages: inputs (blue triangles) and outputs (orange triangles). The forward kinematics model provides frame-level pose and twist; dynamics models compute forces and torques; the forward dynamics model aggregates these to compute the generalized coordinates' time derivative.

Figure 2 illustrates this simulation paradigm. Models are represented as black squares. Blue triangles denote input messages, and orange triangles denote output messages. Message links show publish-subscribe connections.

The forward kinematics model reads the system state $\boldsymbol{q}$ and outputs frame poses and twists (e.g., position $\boldsymbol{r}$, velocity $\dot{\boldsymbol{r}}$, attitude ($\boldsymbol{\omega}$), and angular velocity ($\dot{\boldsymbol{\omega}}$)). Multiple user-defined dynamics models (A, B, C) subscribe to this information and to each other to compute spatial or joint forces ($\boldsymbol{F}$, $\boldsymbol{T}$, τ). The forward dynamics model collects these generalized forces and produces the state derivative $\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}$, which is integrated over time.

This message-based flow encapsulates Eq. 1 in a modular and extensible structure. Simulation engineers focus on writing physically meaningful dynamics models, while the kinematics and dynamics computations are abstracted away.

EXTENSION FOR GENERALIZED DYNAMICAL STATES

The previous section described how multi-body dynamics can be computed from generalized coordinates $\boldsymbol{q}$ and their time derivatives. However, many astrodynamics simulations involve additional physical states whose evolution is governed by ordinary differential equations. These states may include fuel mass, battery charge, onboard filter estimates, or thermal conditions, quantities that evolve continuously over time and influence the system's dynamics or environment.

Let $\boldsymbol{x} \in \mathbb{R}^m$ be a vector of such continuous-time states. Their evolution is described by:

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = g(t, \boldsymbol{x}, \boldsymbol{q}) \tag{2}$$

where the rate of change may depend on time, the current state $\boldsymbol{x}$, and the system configuration $\boldsymbol{q}$.

These states may, in turn, influence the system's dynamics, either by modifying how external forces are computed or by changing the system's inertial properties. For example, fuel depletion changes the spacecraft's mass and center of mass, which affects the resulting accelerations under applied forces. To account for this, Eq. 1 to include dependence on both $\boldsymbol{x}$ and the time-varying inertial properties:

$$\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = f\left(\boldsymbol{q}, \boldsymbol{\tau}(t, \boldsymbol{q}, \boldsymbol{x}), \mathcal{M}(t, \boldsymbol{q}, \boldsymbol{x})\right) \tag{3}$$

where $\mathcal{M}$ represents the set of inertial properties, such as mass, center of mass, and inertia tensor, that may vary over time or depend on system state.

Because the mapping from $\boldsymbol{x}$ to $\mathcal{M}$ is highly application-specific, our simulation paradigm does not prescribe a fixed formulation. Instead, it supports arbitrary time- and state-dependent inertia models implemented as modular components. A fuel tank model, for instance, may define fuel mass as a continuous-time state, compute its rate of change from thrusters' thrust magnitude, and publish the resulting body mass and inertia to be consumed by the dynamics engine.

To support this behavior, the simulation architecture is extended in two key ways:

1. Models can declare and manage their own continuous-time states. These states are integrated along with $\boldsymbol{q}$ and made available to the model at each time step. Each model is responsible for computing and publishing the time derivative of its internal states, which is then used by the integrator to propagate the simulation forward.

2. The forward dynamics model receives inertial properties for each body (mass, center of mass, inertia tensor) via input messages, allowing them to change over time or depend on other model outputs.

Figure 3 illustrates how this extended message-based paradigm supports coupled state evolution and dynamic inertial properties. In this example, two models (E and F) register internal states x_0 and x_1, which are integrated over time. Each model is responsible for computing and publishing the time derivative of its state, which is then used by the integrator to advance the simulation. Model E publishes the center of mass $\boldsymbol{c}_A$ of a body based on its internal state, while also consuming outputs from the forward kinematics model and other models. This information is then passed to the forward dynamics engine, allowing it to compute motion using updated inertia parameters.

DYNAMIC MODEL REUSE

Many existing astrodynamics toolkits include a rich set of dynamics models for computing forces and torques. These often include point-mass, spherical harmonics, and polyhedral gravity models, atmospheric drag models, solar radiation pressure models, and others.[21–25] In most cases, such models take as input elements of the spacecraft state, such as their position or velocity, and return a force or torque vector that is applied to the spacecraft. These models are often well-tested, documented, and form a core part of existing simulation capability.

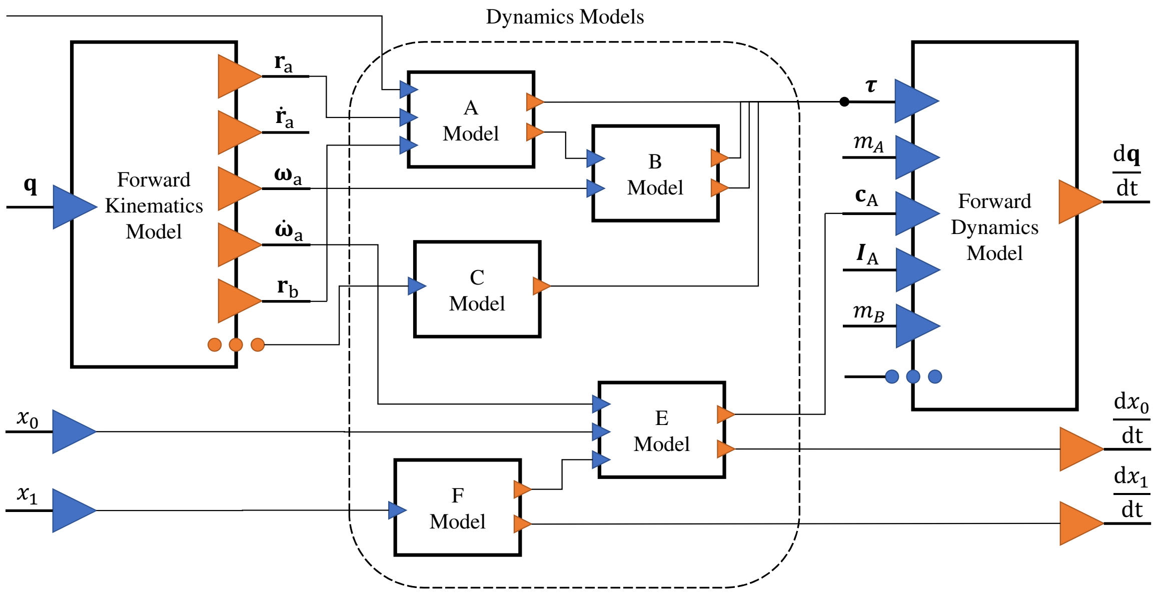


Figure 3: Extension of the model-based simulation paradigm to include continuous-time states and time-varying inertial properties. Models E and F each manage a continuous state (x_0, x_1), compute their time derivatives, and publish inertial properties to the forward dynamics model.

When transitioning to a message-passing, multi-body dynamics architecture, it is highly desirable to preserve this functionality. Fortunately, many existing models can be adapted to fit the new paradigm with minimal modification. The key idea is to isolate each model's computation from its access to a global simulation state. Rather than allowing direct access to internal spacecraft variables, inputs are provided through well-defined messages, and outputs are likewise published as messages. This encapsulation enables models to operate independently of the overall system topology.

For instance, in a traditional single-body simulation, models often assume access to the spacecraft hub's position and velocity as shared variables. These models then compute a net force and torque to be applied to the spacecraft's center of mass. In the proposed architecture, this same model can be adapted by subscribing to messages that contain the position and velocity of a specified body frame, and by publishing the resulting force and torque as output messages. These outputs can then be routed to the forward dynamics engine. This approach is functionally equivalent, but now supports arbitrary bodies, vehicles, and configurations.

Refactoring existing models into message-based dynamics modules not only preserves legacy functionality, but also extends its applicability. For example, Basilisk's gravity models, originally designed to act on a single hub body, can now compute gravity vectors for multiple bodies in a spacecraft. Because each body may have a different position relative to the planet, this naturally introduces first-order gravity gradient torques. Previously unavailable in Basilisk without manually specifying a torque model, such effects now arise naturally from spatial force distributions.

Generalizing legacy models also enables new usecases. Dynamics models can now be used outside the dynamics pipeline; for example, as part of a feedforward control system or estimation algorithm. Moreover, this modularization makes the models easier to understand, test, and verify, since they no longer rely on global state.

In Basilisk, evaluating force and torque contributions such as atmospheric drag, solar radiation pressure or thruster forces were traditionally implemented as *dynamic effectors*, which are objects that query the spacecraft state and apply forces or torques.[2,26] To enable their reuse in the new architecture, a special wrapper model is developed. This wrapper takes in kinematic messages (e.g., position, velocity, attitude) from arbitrary frames, formats them into the variables expected by the effector, invokes the effector's original computation, and then converts the result into output messages compatible with the new architecture. This enables significant reuse of existing dynamic effectors and enables a single code base to be maintained for both the BSM and newly implemented MuJoCo dynamics engines within Basilisk.

Figure 4 illustrates this approach. The wrapper creates a "mock environment" that mimics the state structure the effector expects. The effector operates unchanged as a black box, and its outputs are re-published as force and torque messages. This enables existing effectors to be reused within a message-passing multi-body simulation without rewriting their internals.

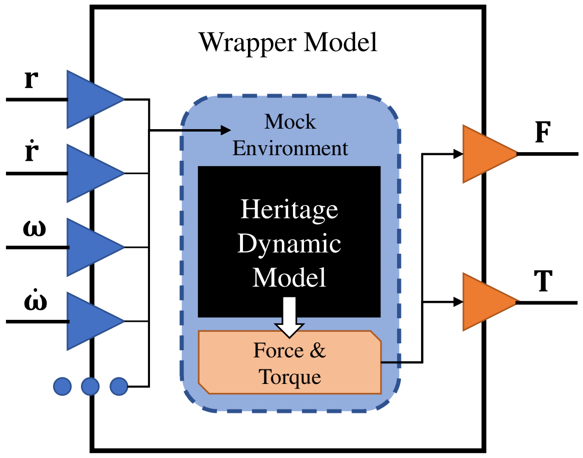
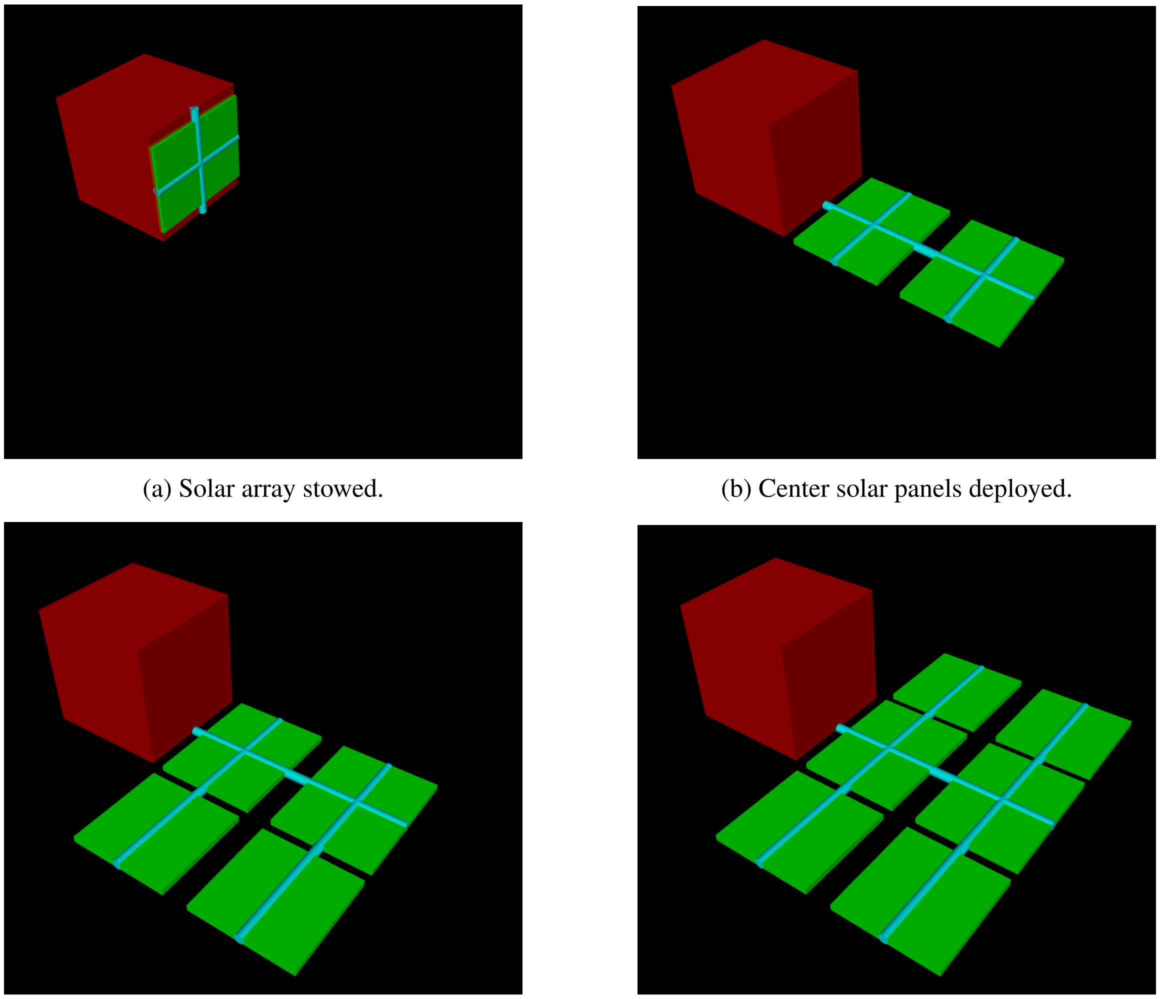


Figure 4: A wrapper model that adapts legacy single-body dynamics modules (e.g., Basilisk dynamic effectors) for use in a multi-body, message-passing simulation. The wrapper receives kinematic inputs via messages, populates internal variables expected by the original model, invokes the legacy force/torque computation, and publishes outputs in message format.

EXAMPLE SIMULATION SETUP AND RESULTS

This section demonstrates a complete simulation that uses the proposed model-based, message-passing paradigm, implemented in Basilisk with MuJoCo as the multi-body dynamics engine. The scenario involves a satellite deploying a complex articulated solar array. The simulation includes joint actuation using PID controllers, joint locking constraints, and staged deployment phases.

The spacecraft consists of a central body with six solar panels arranged in a 3×2 branching configuration. Each panel is hinged and deployed using an actuated joint. A PID controller is assigned to each joint, which tracks a smooth reference profile. The deployment is performed in three stages: the center panels first, followed by the right, then the left (see Figure 5). This simulation also features hinge limits and hinge locking, which is accomplished by leveraging the extensive MuJoCo constraint system.



(a) Solar array stowed.

(b) Center solar panels deployed.

(c) Right solar panels deployed.

(d) Complete solar array deployed.

Figure 5: Rendering of a simulated spacecraft multi-body system during solar array deployment. The six panels are deployed in a staged sequence, illustrating the model's ability to simulate complex, branching articulation.

Multi-body Configuration

Appendix A lists the full MuJoCo XML file used to define the spacecraft's multi-body structure. The XML defines each body using `<body>` elements with nested joints, geometries, and child bodies to create the kinematic tree[*]. Joints are declared with the `<joint>` (or `<freejoint>`) elements; these introduce degrees of freedom between a body and its parent body. A joint of `type="hinge"`, for instance, defines a scalar, rotational degree of freedom along a particular axis. Geometries, on the other hand, define the mass properties of a body, as well as its contact surfaces and its visualization properties.

The MuJoCo format supports both primitive and mesh geometries, and can either infer inertial properties from geometry or accept them explicitly. In this simulation, solar panels are modeled as low-density thin rectangular prisms with denser cylindrical support beams. Letting MuJoCo determine the appropriate inertial properties from this high-level structural description is faster and less-error prone than manual computation.

```python
from Basilisk.simulation import mujoco
from Basilisk.simulation import svIntegrators

scene = mujoco.MJScene.fromFile(XML_PATH)
integ = svIntegrators.svIntegratorRKF45(scene)
scene.setIntegrator(integ)

joints: dict[str, mujoco.MJScalarJoint] = {}
for panelID in ["10", "1p", "1n", "20", "2p", "2n"]:
    bodyName = f"panel_{panelID}"
    body: mujoco.MJBody = scene.getBody(bodyName)

    jointName = f"panel_{panelID}_deploy"
    joints[panelID] = body.getScalarJoint(jointName)
```

Listing 1: Loading a MuJoCo XML file in Basilisk, setting an integrator, and querying joints and bodies.

Listing 1 shows how one might load a MuJoCo XML file into a Basilisk `MJScene`. This object is a wrapper to fundamental MuJoCo data structures that also adds a series of methods to facilitate scripting and interaction with the rest of the Basilisk ecosystem. For example, through `setIntegrator`, one is able control what integrator to use from the many integrators available in Basilisk. This listing also shows how one might query the bodies and joints defined in the XML file and obtain their Basilisk representations (`MJBody` and `MJBody`).

Joint Control and Deployment Logic

Each solar panel is controlled using a PID controller connected to its joint actuator. Reference position and velocity trajectories are generated using interpolators that follow smooth deployment profiles. Listing 2 shows how each controller is initialized and connected to the joint it controls. Each controller needs four inputs: the measured joint position and velocity, and the desired joint position and velocity. Here, we assume perfect measurements and simply connect the joint state output messages of the forward kinematics model to the measurement inputs of the controller. To

[*]MuJoCo XML Reference, `https://mujoco.readthedocs.io/en/stable/XMLreference.html`, Accessed: 2025-07-19

obtain the desired position and velocity, we use a smooth profile that goes from the hinge's stowed position to the deployed position.

Appendix B contains the full implementation of `PIDController`, which illustrates how models read and write messages, as well as declare and handle continuous-time states.

```python
def addJointController(panelID: str, initialAngle: float, timeOffset: int):
    joint = joints[panelID]
    act = scene.addJointSingleActuator(f"panel_{panelID}_deploy", joint)

    pidController = PIDController(K_p = 0.1, K_d = 0.002, K_i = 0.0001)

    positionInterpolator, velocityInterpolator = generateProfiles(
        initialPoint=initialAngle, # rad
        finalPoint=0, # rad
        vMax=np.deg2rad(0.05),
        aMax=np.deg2rad(0.0001),
        timeOffset=timeOffset
    )

    pidController.desiredInMsg.subscribeTo(positionInterpolator.
    interpolatedOutMsg)
    pidController.desiredDotInMsg.subscribeTo(velocityInterpolator.
    interpolatedOutMsg)
    pidController.measuredInMsg.subscribeTo(joint.stateOutMsg)
    pidController.measuredDotInMsg.subscribeTo(joint.stateDotOutMsg)
    act.actuatorInMsg.subscribeTo(pidController.outputOutMsg)

    scene.AddModelToDynamicsTask(positionInterpolator, priority=50)
    scene.AddModelToDynamicsTask(velocityInterpolator, priority=49)
    scene.AddModelToDynamicsTask(pidController, priority=25)

addJointController("10", initialAngle=np.pi/2, timeOffset=0)
addJointController("20", initialAngle=np.pi,   timeOffset=0)
addJointController("1p", initialAngle=np.pi,   timeOffset=operationTime)
addJointController("2p", initialAngle=np.pi,   timeOffset=operationTime)
addJointController("1n", initialAngle=np.pi,   timeOffset=2*operationTime)
addJointController("2n", initialAngle=np.pi,   timeOffset=2*operationTime)
```

Listing 2: Adding a PID controller for a panel's hinge motor to follow a smooth deploy trajectory.

Figure 6 provides a schematic illustration of the models and message connections described in the previous paragraph. The position and velocity interpolator models take the simulation time t as their sole input and output the desired joint position θ^*_{1n} and velocity $\dot{\theta}^*_{1n}$, respectively. These desired values are fed into the PID controller model.

The controller also receives as input the measured joint position and velocity, θ_{1n} and $\dot{\theta}_{1n}$, which are published by the forward kinematics model. In addition, the controller accesses its own continuous-time state, the integral error e_{1n}.

Based on these inputs, the controller computes two outputs: a torque command τ_{1n}, which is sent to the forward dynamics model to be applied at the joint, and the time derivative of the integral error state, $\frac{de_{1n}}{dt}$, which is returned to the integrator for propagation.

Simulation Initialization and Execution

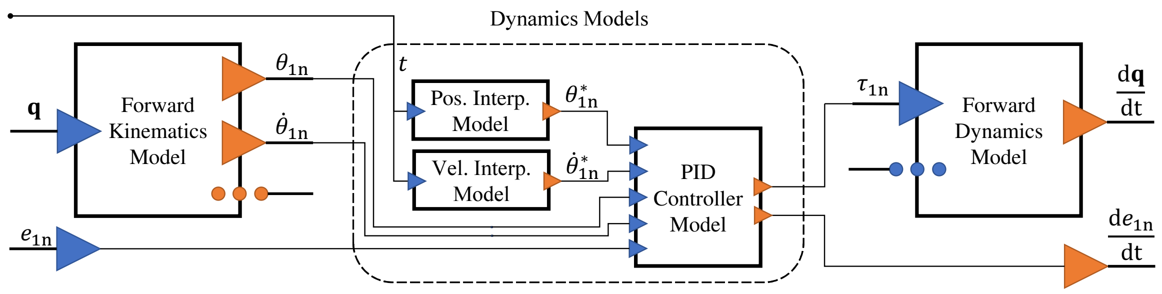


Figure 6: Message-based control architecture for a solar panel joint using PID control. Interpolator models provide desired position and velocity; the PID controller reads measured and desired values, computes control torque, and publishes it to the forward dynamics model. The integral error is tracked as a continuous-time state.

```python
sim.InitializeSimulation()

for panelID in ["10", "1p", "1n", "20", "2p", "2n"]:
    initialAngle = np.pi/2 if panelID == "10" else np.pi # rad
    joints[panelID].setPosition(initialAngle)
```

Listing 3: Initializing the state of the multi-body.

Listing 3 sets the initial joint angles to represent a fully stowed configuration. This listing illustrates how one can set the position (and also velocity) of any joint in the multi-body system. For hinge joints, the position simply represents the angular displacement with respect to some zero position. For free-floating bodies, it's possible to set their full pose and twist.

```python
for panelID in ["1p", "1n", "2p", "2n"]:
    lockJoint(panelID, angle=np.pi)

sim.ConfigureStopTime(operationTime)
sim.ExecuteSimulation()

lockJoint("10", angle=0)
lockJoint("20", angle=0)
unlockJoint("1p")
unlockJoint("2p")

sim.ConfigureStopTime(2*operationTime)
sim.ExecuteSimulation()

lockJoint("1p", angle=0)
lockJoint("2p", angle=0)
unlockJoint("1n")
unlockJoint("2n")

sim.ConfigureStopTime(3*operationTime)
sim.ExecuteSimulation()
```

Listing 4: Executing the solar array deploy scenario.

Listing 4 executes the deployment sequence in three stages. In each stage, only a subset of joints is actively controlled, while the remaining joints are temporarily "locked" using MuJoCo's joint constraint system. This ensures that only the intended panels move during each phase.

In the first stage, the center panels are deployed (Figure 5a to 5b); in the second, the right-side panels are deployed (Figure 5b to 5c); and in the final stage, the left-side panels complete the deployment (Figure 5c to 5d). This staggered activation is achieved by applying time offsets to the desired joint trajectories. Additionally, to simulate a mechanical locking mechanism when panels are either stowed or fully deployed, the MuJoCo constraint system is used to fix joint positions at specific angles.

```python
def lockJoint(panelID: str, angle: float):
    jointConstraintMsg = messaging.ScalarJointStateMsg()
    jointConstraintMsgPayload = messaging.ScalarJointStateMsgPayload()
    jointConstraintMsgPayload.state = angle # rad
    jointConstraintMsg.write(jointConstraintMsgPayload, 0, -1)

    joints[panelID].constrainedStateInMsg.subscribeTo(jointConstraintMsg)

def unlockJoint(panelID: str):
    # Disconnect any message constraining the joint value
    joints[panelID].constrainedStateInMsg.unsubscribe()
```

Listing 5: Locking and unlocking specific joints on the multi-body.

Listing 5 shows how joints are locked and unlocked using Basilisk's message interface. Each joint exposes a `constrainedStateInMsg`, which can be connected to a message specifying a desired fixed angle. This constraint is enforced by the forward dynamics model during simulation. Unlocking is achieved simply by unsubscribing from the constraint message, which restores the joint's freedom of motion.

Although not previously discussed in detail, this feature illustrates how the message-passing paradigm cleanly supports extensions to both the inputs and outputs of the forward kinematics and dynamics models. In this case, MuJoCo's constraint system is exposed as a first-class input message, demonstrating how simulation capabilities can be expanded within the proposed simulation paradigm.

Results

Figure 7 shows the joint angles of all six solar panels during the deployment sequence. Each panel's commanded trajectory (solid blue) is compared against its actual motion (dashed red). The PID controllers successfully track the target profiles, validating the controller implementation and the message-driven dynamics architecture.

This simulation demonstrates that a fully message-based, model-driven architecture can support complex, articulated spacecraft scenarios. Using MuJoCo's kinematics and dynamics engines within Basilisk enables realistic joint behavior, constraint enforcement, and dynamic evolution while isolating physical effects into clean, reusable models.

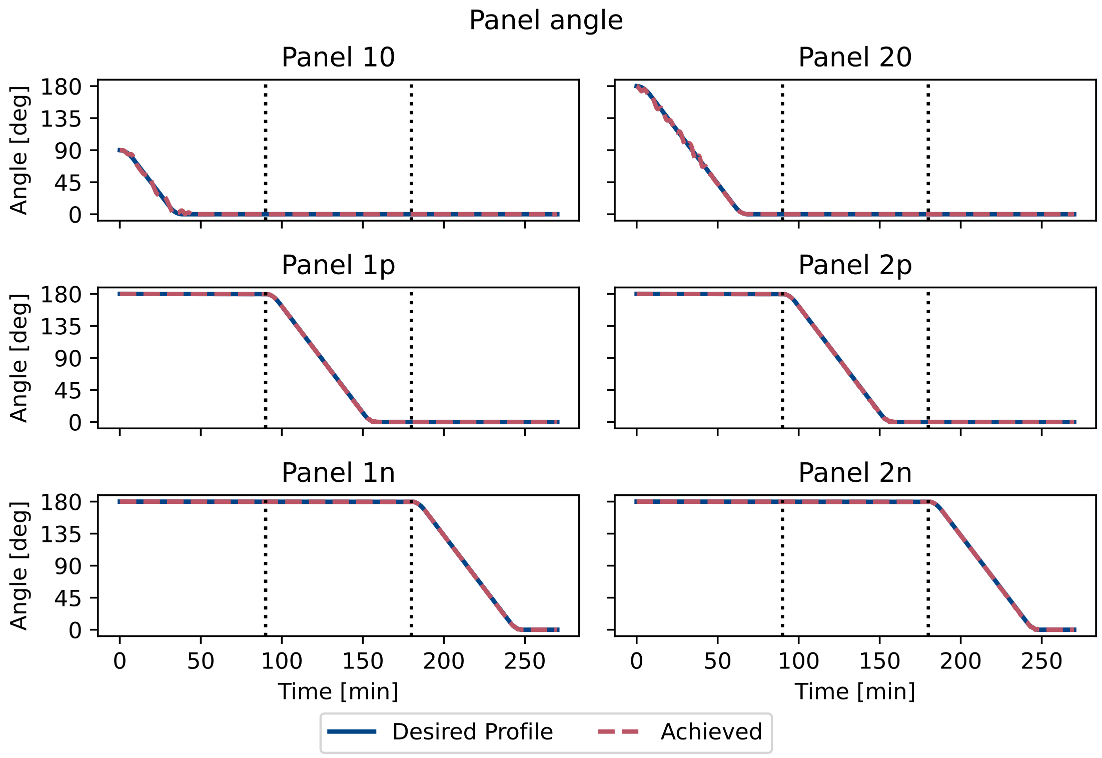


Figure 7: Time evolution of joint angles during staged solar array deployment. Each subplot shows the commanded (solid blue) and actual (dashed red) joint angle for one panel. Vertical dashed lines indicate deployment phases. The plots validate the closed-loop tracking performance of the PID controllers.

FUTURE WORK

This paper presents a fundamental shift into how dynamics are modeled and simulated in Basilisk. Future work will involve benchmarking and validating the new approach with respect to the existing back-substitution-based dynamics already present in Basilisk.[27] Further work with multi-body dynamics in Basilisk will involve exploting these new capabilities to model specific scenarios of interest, such as docking spacecraft, spacecraft landing systems, flexible solar panels, etc.

CONCLUSION

This paper introduced a model-based, message-passing simulation paradigm for spacecraft systems with generally articulated dynamics. The proposed paradigm decomposes the simulation of dynamics into: a generic forward kinematics model; user-defined models that compute generalized forces and torques; and a forward dynamics engine model. These components are linked through a message-passing architecture, enabling intuitive model composition, simplified testing, and clean separation of functionality. Critically, the complexity of rigid-body dynamics algorithms is abstracted away, allowing engineers to focus on modeling physical behavior, not on deriving equations of motion.

A complete example was presented involving staged solar array deployment, joint control using

PID models, and constraint enforcement, all implemented using Basilisk and MuJoCo. The results demonstrate that the architecture can express and execute complex, realistic scenarios using reusable models connected through standardized message interfaces.

Overall, this paradigm represents a significant shift in how spacecraft dynamics are generally modeled and simulated in astrodynamic tools. It brings together modern software engineering principles, such as modularity, encapsulation, and interface-based design, with high-fidelity astrodynmic simulation. It opens the door to reusable, extensible simulation workflows that scale with system complexity, and that bridge the gap between robotics-based multi-body dynamics and astrodynamics usecases.

ACKNOWLEDGMENT

The research was supported by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

Artificial intelligence (AI) tools were used in the preparation of this manuscript in accordance with AIAA guidelines. Specifically, AI was employed as a software development assistant to support code workflows relevant to the work. Additionally, AI was used to review and improve the clarity, grammar, and readability of the manuscript text.

REFERENCES

[1] J. Garcia-Bonilla, C. Leake, A. Elmquist, T. D. Hasseler, V. Steyert, A. Gaut, and A. Jain, "Dshell-DARTS: A Reusability-Focused Multi-Mission Aerospace and Robotics Simulation Toolkit," *2025 IEEE Aerospace Conference*, 2025, pp. 1–13, 10.1109/AERO63441.2025.11068690.

[2] P. W. Kenneally, S. Piggott, and H. Schaub, "Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework," *Journal of Aerospace Information Systems*, Vol. 17, No. 9, 2020, pp. 496–507, 10.2514/1.I010762.

[3] M. Cols-Margenet, H. Schaub, and S. Piggott, *Modular Attitude Guidance Development using the Basilisk Software Framework*, 10.2514/6.2016-5538.

[4] E. Mooij and M. Ellenbroek, "Multi-Functional Guidance, Navigation, and Control Simulation Environment," *Journal of Physics: Conference Series*, 08 2007, 10.2514/6.2007-6887.

[5] D. M. Geletko, M. D. Grubb, J. P. Lucas, J. R. Morris, M. Spolaor, M. D. Suder, S. C. Yokum, and S. A. Zemerick, "NASA Operational Simulator for Small Satellites (NOS3): the STF-1 CubeSat case study," 2019.

[6] M. Tipaldi, R. Iervolino, and P. R. Massenio, "Reinforcement learning in spacecraft control applications: Advances, prospects, and challenges," *Annual Reviews in Control*, Vol. 54, 2022, pp. 1–23, https://doi.org/10.1016/j.arcontrol.2022.07.004.

[7] A. K. Banerjee, "Contributions of Multibody Dynamics to Space Flight: A Brief Review," *Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 3, 2003, pp. 385–394, 10.2514/2.5069.

[8] A. Seddaoui, C. M. Saaj, and M. H. Nair, "Modeling a Controlled-Floating Space Robot for In-Space Services: A Beginner's Tutorial," *Frontiers in Robotics and AI*, Vol. 8, 12 2021, p. 725333, 10.3389/FROBT.2021.725333/FULL.

[9] Y. Qi and M. Shan, "Simulation on Flexible Multibody System with Topology Changes for In-space Assembly," *Journal of the Astronautical Sciences*, Vol. 71, Apr. 2024, p. 11, 10.1007/s40295-024-00431-0.

[10] J. Vaz Carneiro and H. Schaub, "Scalable architecture for rapid setup and execution of multi-satellite simulations," *Advances in Space Research*, Vol. 73, No. 11, 2024, pp. 5416–5425. Recent Advances in Satellite Constellations and Formation Flying, https://doi.org/10.1016/j.asr.2023.11.026.

[11] W. Qian, Z. Xia, J. Xiong, Y. Gan, Y. Guo, S. Weng, H. Deng, Y. Hu, and J. Zhang, "Manipulation task simulation using ROS and Gazebo," *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, 2014, pp. 2594–2598, 10.1109/ROBIO.2014.7090732.

[12] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033, 10.1109/IROS.2012.6386109.

[13] G. Boschetti and T. Sinico, "Designing Digital Twins of Robots Using Simscape Multibody," *Robotics*, Vol. 13, No. 4, 2024, 10.3390/robotics13040062.

[14] S. P. Hughes, R. H. Qureshi, S. D. Cooley, and J. J. Parker, *Verification and Validation of the General Mission Analysis Tool (GMAT)*. 2014, 10.2514/6.2014-4151.

[15] A. Wall, *Systems Tool Kit (STK)*, pp. 11–32. CRC Press, 1 2024, 10.1201/9781003321811-4/SYSTEMS-TOOL-KIT-STK-ALEXIS-WALL.

[16] G. Dell, M. Hametz, P. Noonan, L. Newman, D. Folta, and J. Bristow, *EOS AM-1 and EO-1 support using FreeFlyer and AutoCon*. 1998, 10.2514/6.1998-4196.

[17] J. a. Vaz Carneiro, C. Allard, and H. Schaub, "General Dynamics for Single- and Dual-Axis Rotating Rigid Spacecraft Components," *Journal of Spacecraft and Rockets*, Vol. 61, No. 4, 2024, pp. 1099–1113, 10.2514/1.A35865.

[18] A. M. Morell, J. V. Carneiro, L. Kiner, and H. Schaub, *Multi-Arm Post-Docking Spacecraft Dynamics Using Penalty Methods*, 10.2514/6.2024-1870.

[19] J. Alcorn, C. Allard, and H. Schaub, "Fully Coupled Reaction Wheel Static and Dynamic Imbalance for Spacecraft Jitter Modeling," *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 6, 2018, pp. 1380–1388, 10.2514/1.G003277.

[20] S. Carnahan, S. Piggott, and H. Schaub, "A New Messaging System For Basilisk," *AAS Guidance and Control Conference*, Breckenridge, CO, Jan. 30 – Feb. 5 2020. AAS 20-134.

[21] J. Schuhmacher, E. Blazquez, F. Gratl, D. Izzo, and P. Gómez, "Efficient Polyhedral Gravity Modeling in Modern C++ and Python," *Journal of Open Source Software*, Vol. 9, No. 98, 2024, p. 6384, 10.21105/joss.06384.

[22] J. R. Martin and H. Schaub, "The Physics-Informed Neural Network Gravity Model Generation III," *The Journal of the Astronautical Sciences*, Vol. 72, No. 10, 2025, 10.1007/s40295-025-00480-z.

[23] D. Vallado and D. Finkleman, "A Critical Assessment of Satellite Drag and Atmospheric Density Modeling," *Acta Astronautica*, Vol. 95, 02 2014, 10.1016/j.actaastro.2013.10.005.

[24] P. W. Kenneally, H. Schaub, and S. Tanygin, "OpenGL-OpenCL Solar Radiation Pressure Modeling with Time-Varying Spacecraft Geometries," *AIAA Journal of Aerospace Information Systems*, Vol. 18, May 2021, pp. 307–321, 10.2514/1.I010869.

[25] A. Elmquist, V. Steyert, S. Diehl, and A. Jain, "Accurate, GPU Accelerated Solar Radiation Pressure Modeling for Exo-Atmosphere Trajectory Simulation," *2025 IEEE Aerospace Conference*, 2025, pp. 1–10, 10.1109/AERO63441.2025.11068524.

[26] J. Vaz Carneiro and H. Schaub, "Spacecraft Dynamics With The Backsubstitution Method: Survey And Capabilities," *AAS Spaceflight Mechanics Meeting*, Kauai, Hawaii, Jan. 19–23 2025. Paper No. AAS 25-232.

[27] C. Allard, M. Diaz Ramos, and H. Schaub, "Computational Performance of Complex Spacecraft Simulations Using Back-Substitution," *Journal of Aerospace Information Systems*, Vol. 16, No. 10, 2019, pp. 427–436, 10.2514/1.I010713.

APPENDIX A: SOLAR ARRAY SCENARIO MULTI-BODY DEFINITION

The following is the content of the XML file used to define the multi-body system discussed in Section “Multi-body Configuration”.

```xml
<mujoco>
  <option gravity="0 0 0">
    <flag contact="disable"/>
  </option>

  <worldbody>
    <body name="hub">
      <freejoint/>
      <geom name="hub_box" type="box" size="1 1 1" density="2700" rgba="1 0 0 0.5"/>

      <body name="panel_10" pos="1 0 -1" xyaxes="0 -1 0 0 0 -1">
        <joint name="panel_10_deploy" type="hinge" axis="1 0 0" range="0 90"/>
        <geom name="panel_10_box" type="box"  pos="0 0 1" size="1 0.05 0.8" density="200" rgba="0 1 0 1"/>
        <geom name="panel_10_bar" type="cylinder" fromto="0 0 0 0 0 2" size=".075" density="1000" rgba="0 1 1 1"/>
        <geom name="panel_10_cbar" type="cylinder" fromto="1 0 1 -1 0 1" size=".075" density="1000" rgba="0 1 1 1" />

        <body name="panel_1p" pos="1 0 1" axisangle="0 1 0 90">
                    <joint name="panel_1p_deploy" type="hinge" axis="1 0 0" range="0 180"/>
                    <geom name="panel_1p_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8" density="200" rgba="0 1 0 1"/>
          <geom name="panel_1p_bar" type="cylinder" fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
                </body>

        <body name="panel_1n" pos="-1 0 1" axisangle="0 -1 0 90">
                    <joint name="panel_1n_deploy" type="hinge" axis="1 0 0" range="0 180"/>
                    <geom name="panel_1n_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8" density="200" rgba="0 1 0 1"/>
          <geom name="panel_1n_bar" type="cylinder"  fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
                </body>

                <body name="panel_20" pos="0 0 2">
                    <joint name="panel_20_deploy" axis="-1 0 0" range="0 180"/>
                    <geom name="panel_20_box" type="box"  pos="0 0 1" size="1 0.05 0.8" density="200" rgba="0 1 0 1"/>
          <geom name="panel_20_bar" type="cylinder"  fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
          <geom name="panel_20_cbar" type="cylinder" fromto="1 0 1 -1 0 1" size=".075" density="1000" rgba="0 1 1 1" />

          <body name="panel_2p" pos="1 0 1" axisangle="0 1 0 90">
            <joint name="panel_2p_deploy" type="hinge" axis="1 0 0" range="0 180"/>
            <geom name="panel_2p_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8
```

```
" density="200" rgba="0 1 0 1"/>
            <geom name="panel_2p_bar" type="cylinder"  fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
          </body>

          <body name="panel_2n" pos="-1 0 1" axisangle="0 -1 0 90">
            <joint name="panel_2n_deploy" type="hinge" axis="1 0 0" range="0 180"/>
            <geom name="panel_2n_box" type="box"  pos="0 0 1" size="0.8 0.05 0.8" density="200" rgba="0 1 0 1"/>
            <geom name="panel_2n_bar" type="cylinder"  fromto="0 0 0 0 0 1.8" size=".075" density="1000" rgba="0 1 1 1" />
          </body>
                </body>
      </body>

    </body>

  </worldbody>

</mujoco>
```

APPENDIX B: PIDCONTROLLER MODEL DEFINITION

```python
class PIDController(StatefulSysModel.StatefulSysModel):
    """
    A Proportional-Integral-Derivative (PID) Controller class for controlling joint states.

    This models an analog PID controller, which means that its output evolves in continuous
    time, not discrete time. Thus, it should be used within the dynamics task of ``MJScene``.

    Attributes:
        K_p (float): Proportional gain.
        K_d (float): Derivative gain.
        K_i (float): Integral gain.

        measuredInMsg (messaging.ScalarJointStateMsgReader): Reader for the measured joint state.
        desiredInMsg (messaging.ScalarJointStateMsgReader): Reader for the desired joint state.
        measuredDotInMsg (messaging.ScalarJointStateMsgReader): Reader for the measured joint state derivative.
        desiredDotInMsg (messaging.ScalarJointStateMsgReader): Reader for the desired joint state derivative.

        outputOutMsg (messaging.SingleActuatorMsg): Output message, contains the torque for the connected actuator.
    """

    def __init__(self, K_p: float = 0.1, K_d: float = 0.002, K_i: float = 0.0001):
        """Initialize"""
        super().__init__()
        self.K_p = K_p
        self.K_d = K_d
        self.K_i = K_i

        self.measuredInMsg = messaging.ScalarJointStateMsgReader()
        self.desiredInMsg = messaging.ScalarJointStateMsgReader()

        self.measuredDotInMsg = messaging.ScalarJointStateMsgReader()
        self.desiredDotInMsg = messaging.ScalarJointStateMsgReader()

        self.outputOutMsg = messaging.SingleActuatorMsg()

    def registerStates(self, registerer: StatefulSysModel.DynParamRegisterer):
        """Declared the need for a continuous-time state: the integral error"""
        self.integralErrorState = registerer.registerState(1, 1, "integralError")
        self.integralErrorState.setState([[0]]) # explicitely zero initialize

    def UpdateState(self, CurrentSimNanos: int):
        """Computes the control command from the measured and desired
        joint position and velocity."""
        # Compute the error in the state and its derivative
        stateError = self.desiredInMsg().state - self.measuredInMsg().state
        stateDotError = self.desiredDotInMsg().state - self.measuredDotInMsg().state
        stateIntegralError = self.integralErrorState.getState()[0][0]

        # Compute the control output
        control_output = self.K_p * stateError + self.K_d * stateDotError + self.K_i * stateIntegralError

        # Set the derivative of the integral error inner state
        self.integralErrorState.setDerivative([[stateError]])

        # Write the control output to the output message
        payload = messaging.SingleActuatorMsgPayload()
        payload.input = control_output
        self.outputOutMsg.write(payload, CurrentSimNanos, self.moduleID)
```

Listing 6: Definition of the `PIDController` model.