Kareem Omar
kareem.omar@uah.edu

Simultaneous Orbital and Attitude Propagation of Satellites in Low-Earth Orbit using CUDA for Aerodynamics Simulation

Introduction

This paper branches into three areas of aerospace education.

The first area is a collection of thorough *background* information on fundamental aerospace and physics topics relevant to this simulation, from gravity gradient torque to numerical integration. Sections in this category are prefixed with **Background**.

The second area is the *application* of the software infrastructure developed by the author, known as KPS, to modeling satellite behavior in a flexible and accessible manner suitable for university and other aerospace education. This path is less concerned with the inner workings of the simulation and more concerned with how it may be used to illustrate orbital and attitude dynamics to aerospace students and directly facilitate experimentation with satellite and mission design. Sections in this category are prefixed with **KPS Usage.**

The third area is the *design* of the software itself, encompassing CUDA and other high-performance computing techniques and the development of flexible simulation systems. The ability to construct high-performance simulations for aerospace problems places programmers in a valuable position in the aerospace industry, and as such, KPS has been designed to illustrate relevant software design concepts in a modular and clear manner for the interested programmer. This paper thus also focuses on the computational design of KPS for aerospace students who wish to add powerful computational tools to their arsenal, or for programmers who wish to apply their skills to the field of aerospace. Sections in this category are prefixed with **KPS Internals**.

An educational focus is maintained throughout. The author perceives significant value in providing a unified educational resource encompassing the broad range of topics involved in fusing satellite behavior and computer simulation, from quaternions to numerical integration to CUDA reduction techniques, particularly since specific code samples and use cases can be demonstrated in a concrete manner from their actual implementations in KPS.

Background – CUDA

CUDA is a parallel processing platform developed by NVIDIA[1]. CUDA tools compile code written in C++ for execution on the Graphics Processing Unit (GPU) rather than the Central Processing Unit (CPU), where code typically executes. GPUs have evolved to meet the needs of gaming graphics, which typically follow a Single Instruction, Multiple Data (SIMD) paradigm in

which little branching occurs and the same instruction must be rapidly (preferably simultaneously) executed on different data. See Figure 1. Note that CPUs do also have some SIMD capabilities, though they cannot rival GPUs for massively parallel compute-bound tasks.

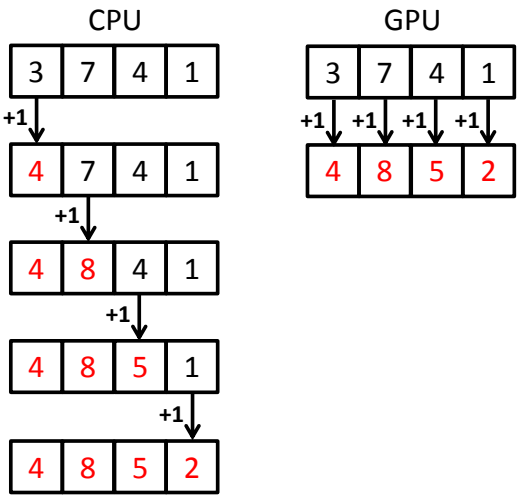

Figure 1. *Incrementing a 4-vector on CPU vs. GPU (simplified)*

As NVIDIA states, "The CPU… has often been called the brains of the PC. But increasingly, that brain is being enhanced by another part of the PC – the GPU…, which is its soul."[2]

With careful programming, certain massively parallel problems can be solved by GPUs much more quickly than by CPUs. In KPS, the aerodynamics simulation module is precisely such a problem. KPS provides aerodynamics simulation in either an analytical frontal area compute mode or by repeatedly spawning grids of particles to collide with the satellite, as will be discussed at length later.

Users who wish to run KPS in CUDA mode must have a CUDA-capable NVIDIA GPU. The full list of supported devices is available on NVIDIA's website[3]. Users wishing to compile KPS from scratch must also have the CUDA SDK, also available for free from NVIDIA[4].

The author understands that not all interested parties may have CUDA-capable devices, so *CUDA is not required to run KPS*. The software can run in a carefully optimized CPU-only mode.

Background – Coordinate Systems and Reference Frames

Uniquely determining a position or other quantity containing both magnitude and direction in 3-dimensional space requires a coordinate representation containing 3 **degrees of freedom (DoF)**, one for each dimension. Many such coordinate systems exist, each with advantages in particular applications.

Kareem Omar
kareem.omar@uah.edu

In KPS, **right-handed Cartesian coordinate systems** are used. This means coordinates are specified as a **3-vector** consisting of 3 values describing how far to travel from the **origin** in each of three mutually perpendicular directions x, y, and z. See Figure 2.

A system is **right-handed** if, when one curls the extended fingers of one's right hand from the x-axis around toward the y-axis, the thumb points in the direction of the z-axis. It is **left-handed** otherwise.

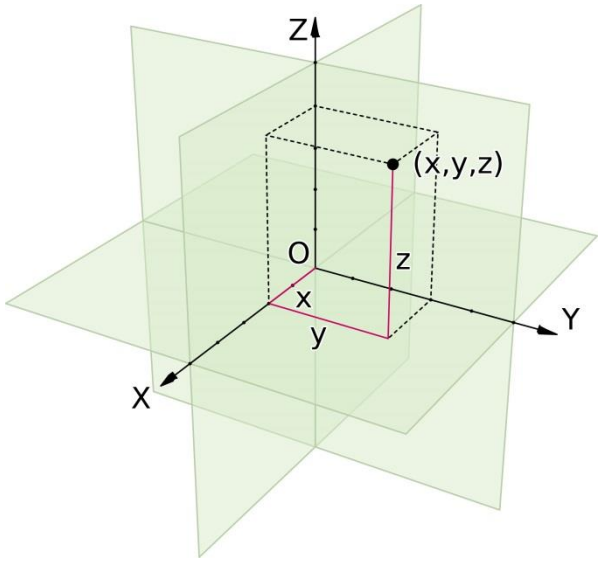

Figure 2. *A 3-D Cartesian coordinate system*

A **reference frame** is a specification of the location of the origin and the orientations of the axes of a coordinate system relative to another. The frame may translate and rotate over time relative to other frames.

Five such frames are discussed here:

1. Earth-Centered Inertial (ECI) Frame
2. Earth-Centered Earth-Fixed (ECEF) Frame
3. Local Vertical, Local Horizontal (LVLH) Frame
4. Orbital Frame
5. Body Frame

These frames are common in aerospace and astrodynamics and are thus explained fully for readers.

1. The ECI frame has its origin at the center of the Earth. Its z-axis points through the Earth's axis of rotation, or the celestial North pole. The x-axis points toward the Sun at vernal equinox. The y-axis completes the frame according to the right-hand rule. This frame does *not* follow Earth's rotation; although it travels with Earth around the Sun and although the entire Solar System is in motion, this frame can be considered an **inertial frame** for this application, meaning that Newton's Laws of Motion can be considered valid. It is the only inertial frame in this list. See Figure 3.
2. The ECEF frame is identical to the ECI frame at **epoch**, an arbitrarily chosen time. After this point, the two frames diverge only in rotation: the ECEF frame rotates along with

Kareem Omar
kareem.omar@uah.edu

Earth, matching its angular velocity, such that a point on Earth's surface remains at the same ECEF coordinates (but *not* at the same ECI coordinates).

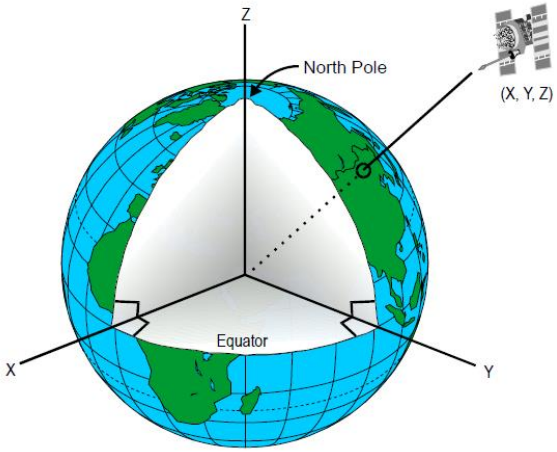

Figure 3. E*CI frame. The ECEF frame aligns with this frame at epoch but rotates about the z-axis along with Earth thereafter.*

3. There exist two common LVLH frames in aerospace: East, North, Up (ENU) and North, East, Down (NED, also known as Local Tangent Plane or LTP). These frames can be thought of as residing tangent to the surface of a sphere centered at Earth's center of mass. This could mean the surface of Earth itself, or perhaps a slightly larger sphere such that the origin of the coordinate system is an aircraft or spacecraft at some altitude above the surface. East-West refers to a direction tangent to Earth's parallels. North-South refers to a direction tangent to the local meridian. Up-Down refers to a direction away from or toward Earth's center, respectively. See Figure 4.

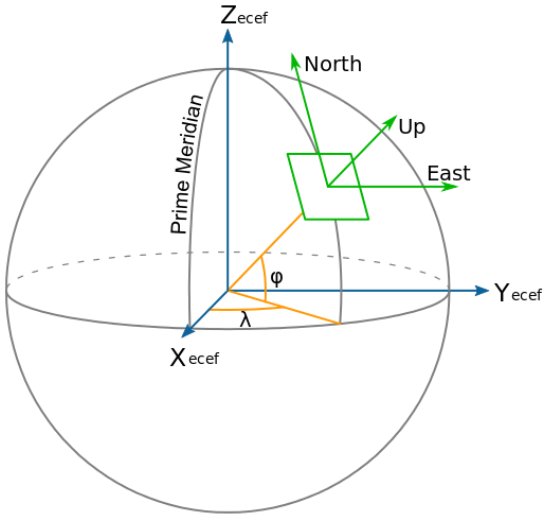

Figure 4. *LVLH frames. The ENU frame is shown here. The NED frame is identical except that the "Up" axis is replaced with a "Down" axis pointing in exactly the opposite sense, and the order of the coordinates is changed to maintain right-handedness.*

4. The Orbital frame has its origin at the center of mass of the spacecraft and travels with the spacecraft. The x-axis points **prograde**, i.e. in the direction of travel, or equivalently, in the direction of the velocity vector). The z-axis points **nadir**, i.e. straight "down" toward the center of Earth. The y-axis completes the frame according to the right-hand rule. Notice that this frame is only rigidly defined if the nadir is orthogonal to the velocity, which is not strictly true at all times for elliptical orbits, only for circular ones. If the nadir is not perfectly orthogonal, the x-axis should remain prograde while the z-axis points as nearly nadir as possible. See Figure 5.
5. The Body frame has its origin at the

center of mass of the spacecraft and travels with the spacecraft. Its axes are fixed to the spacecraft such that it rotates with the spacecraft; in other words, any point anywhere in or on the spacecraft is always at the same Body coordinates no matter the position, velocity, or attitude of the spacecraft. The axes are arbitrarily chosen for a particular satellite. They typically, but not always, are chosen such that they align with the principal moments of inertia of the satellite.

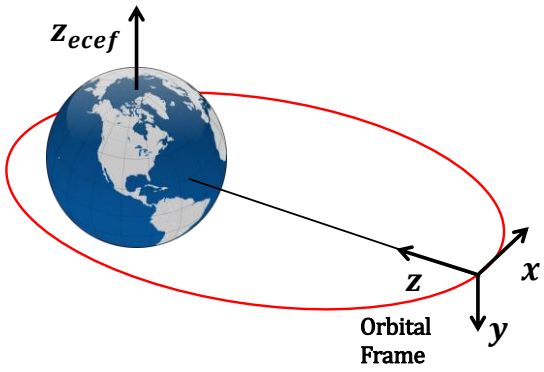

Figure 5. *Orbital frame*

Background – Orbital Dynamics

Consider a **two-body** problem consisting of two point masses in free space. Each exerts a gravitational force on the other according to Newton's Law of Universal Gravitation:

$$F = \frac{Gm_1m_2}{r^2} \tag{1}$$

In this equation, G is the universal gravitational constant. The two bodies, if initially with no velocity, will eventually collide due to this force no matter how far apart they begin. If, however, they have any linear velocity at all in a direction not in line with opposite body, the system also has **angular velocity**, which will be conserved. The bodies will pass around each other as they approach and will **orbit** a common **barycenter**.

If one body is vastly more massive than the other, some simplifying assumptions can be made. This is often the case with the Sun and the Earth, or the Earth and a satellite. Considering the latter case: the mass of a satellite is so much smaller than the mass of Earth that the barycenter is inside the Earth and quite near its center of mass. The Earth is perturbed so little by the satellite that the Earth may be considered stationary as the satellite orbits around it. Thus, only the effects on the satellite need be considered. Newton's Second Law states:

$$F = ma \tag{2}$$

Thus the acceleration due to Earth's gravity on the satellite will be:

$$a_2 = \frac{F}{m_2} = \frac{Gm_1}{r^2} \tag{3}$$

Kareem Omar
kareem.omar@uah.edu

The subscript 2 is now dropped on the acceleration, and m_1 is replaced by M, understood to mean the mass of the Earth:

$$a = \frac{GM}{r^2} \tag{4}$$

The constant GM occurs frequently and is given the name **standard gravitational parameter**. Earth's GM is sometimes represented as μ. It has a value of $3.986004418\ x\ 10^{14} \pm 8\ x\ 10^{5}\ \frac{m^3}{s^2}$. Dealing with this constant rather than separate values of G and M is not only for convenience; the combined value comes directly from astronomical observation and thus has an uncertainty significantly smaller than that of either individual constant.

The effect of this force on a body is to cause it to orbit in a path which follows a **conic section**. Captured orbits of the sort relevant to KPS are **elliptical**, with the orbited body (in this case Earth) at one focus of the ellipse. See Figure 6. The point of closest approach to the orbited body is known as **periapsis** in the general case, or **perigee** in the case of Earth orbit. The point of furthest distance from the orbited body is known as **apoapsis** in the general case, or **apogee** in the case of Earth orbit. A line connecting the perigee and apogee is known as the **line of apsides**. The distance from the center of the ellipse to the nearest point on the ellipse itself is known as the **semi-minor axis**, b. The distance from the center of the ellipse to the furthest point on the ellipse itself is known as the **semi-major axis**, a. The **eccentricity**, e, is a measure of how "squished" the ellipse is and can be calculated by:

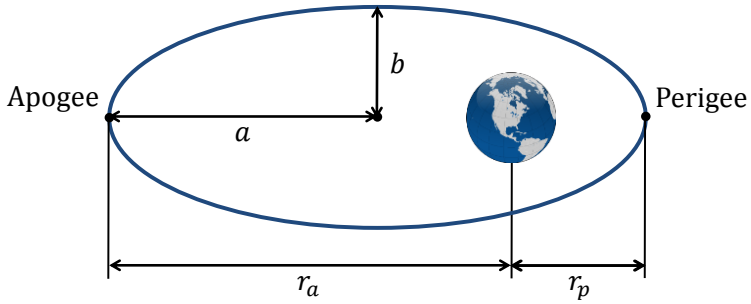

Figure 6. *Elliptical orbit.*

$$e = \sqrt{1 - \left(\frac{b}{a}\right)^2} \tag{5}$$

A circle has eccentricity zero, since $a = b = r$, and both foci are at the center.

Knowing the semi-major axis and eccentricity is a good start to uniquely defining a stable orbit – the general size and shape of the orbit have been described. Now its orientation relative to Earth must be defined.

Kareem Omar
kareem.omar@uah.edu

Consider the "tilt" of the ellipse: how does the plane of the orbit compare to the plane of the equatorial plane? What angle is produced between the two planes? This is known as the **inclination** of the orbit and denoted i. An inclination of zero is equatorial – the satellite is always above the equator. An inclination of 90°, or $\frac{\pi}{2}$ rad, is **polar** – the satellite travels over the north and south poles on every orbit as the Earth rotates underneath.

For an inclination other than 0°, the equatorial and orbital planes intersect at a line connecting the two **nodes**, or locations along the satellite's path where it crosses the equatorial plane. This line is known, rather uncreatively, as the **line of nodes**. The node the satellite crosses on the way "up", from south to north, is known as the **ascending node (AN)**. Conversely, the node it crosses on the way "down" from north to south is the **descending node (DN)**.

One can describe how "twisted" the orbit is relative to the Earth by specifying the angle between the ECI x-axis (recall that this axis points toward the sun at vernal equinox) and the ascending node. This parameter is known as the **Right Ascension of the Ascending Node (RAAN)** or the **Longitude of the Ascending Node (LAN)** and is denoted Ω.

One can describe the "rotation" of the ellipse by specifying where perigee occurs relative to the ascending node. The angle produced between the ascending node and perigee is known as the **argument of perigee** and is denoted ω.

The orbit has now been completely specified! All that remains is to record where the satellite actually *is* in its orbit at a given time. One can describe the satellite's progress along its orbit by specifying the angle between perigee and the satellite. This is known as the **true anomaly** and is denoted ν.

These elements taken together are known as the **classical** or **Keplerian orbital elements**[5]. They are six, summarized in Table 1 and Figure 7:

Kareem Omar
kareem.omar@uah.edu

Element	Symbol	Units	Literal Value	Meaning
Semi-major Axis	a	Length	Center of ellipse to furthest point	Size/Energy of orbit
Eccentricity	e	(Dimensionless)	$\sqrt{1-\left(\frac{b}{a}\right)^2}$	How "squished" the ellipse is
Inclination	i	Angular	Angle between equatorial plane and orbital plane	Tilt of ellipse relative to equatorial plane
RAAN or LAN	Ω	Angular	Angle between vernal equinox and ascending node	"Twist" of ellipse
Argument of Perigee	ω	Angular	Angle between ascending node and perigee	"Rotation" of ellipse
True Anomaly	ν	Angular	Angle between perigee and satellite	Progress of satellite along its orbit

Table 1. *Keplerian orbital elements*

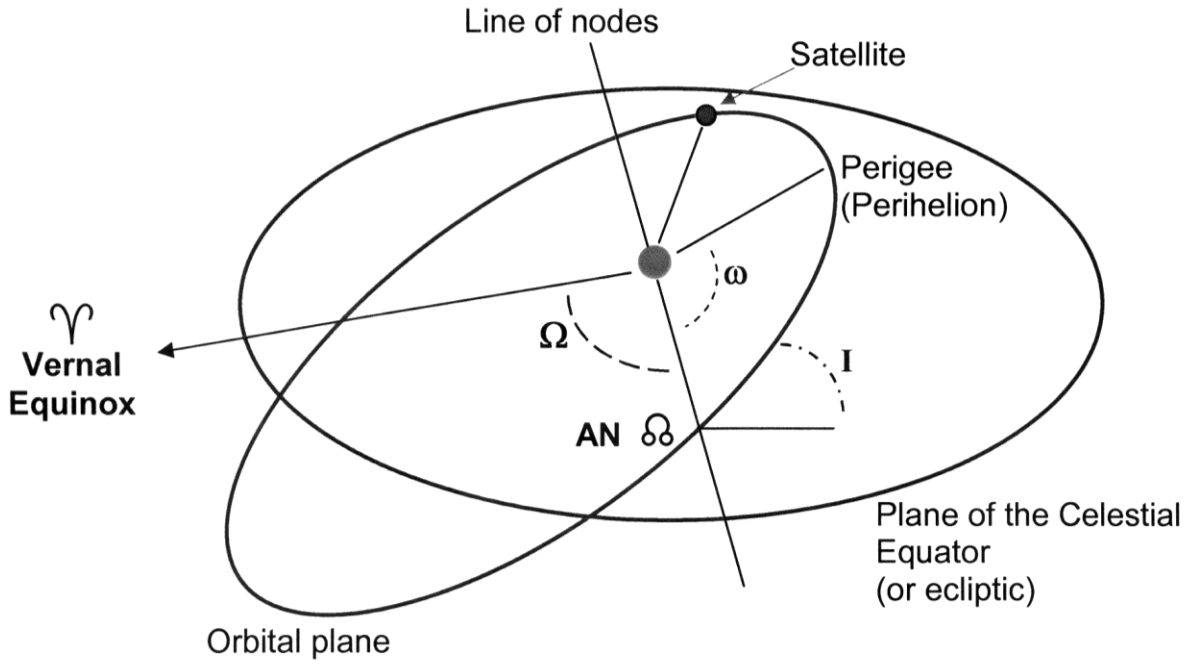

Figure 7. *Keplerian orbital elements*

Kareem Omar
kareem.omar@uah.edu

There also exist variations on the characterization of the satellite's current progress along its orbit. One is known as the **eccentric anomaly** and is denoted E. It can be thought of as what the anomaly *would be* if the orbit were actually circular. It is useful in deriving perturbations for circular orbits and then generalizing them to ellipses. The precise angular value is the angle between the perigee and a point on a circle with radius a, directly "above" (in the direction perpendicular to the line of apsides) the satellite's actual position. See Figure 8.

Another is known as the **mean anomaly** and is denoted M. It, too, is in a sense a measure of what the anomaly would be if the orbit were circular. The mean anomaly is the angular distance the satellite would have traveled if it were in a circular orbit *with the same orbital period* as the real elliptical orbit. It is notable because **Two-Line Element Sets**, or **TLEs** provided by NORAD and others for satellite tracking provide mean, not true, anomaly.

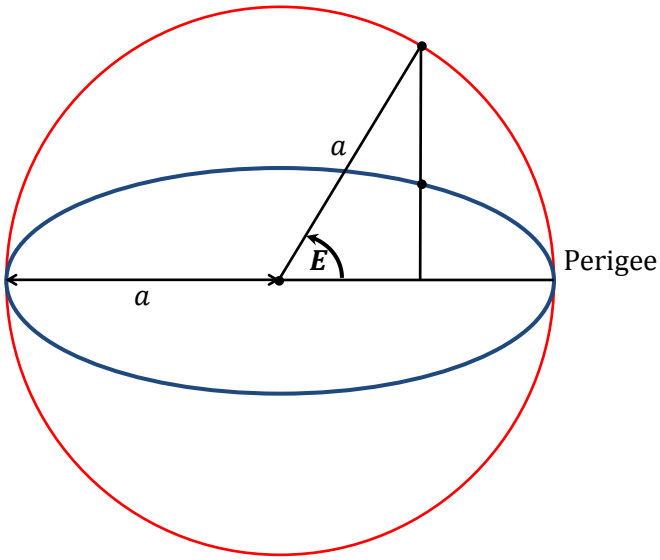

Figure 8. *Eccentric anomaly, E*

Other than the anomalies, the Keplerian elements describe the orbit itself and so do not change over time in a simple two-body problem. The advantage of this is that a **closed form** has thus been specified for the orbit – given Keplerian elements at a particular time, one can, in constant time, compute the position of the satellite at any time in the future without requiring integration over the intervening time.

However, in reality, many effects, called **perturbations**, cause even the first five Keplerian elements to change slightly over time. Some oscillate about equilibrium with various periods of oscillation; others precess. Perturbations vary in significance, and many can be ignored, depending on the application.

Some perturbations can be approximately modeled such that the orbit remains closed-form with additional computation. Examples include correcting for the fact that the Earth is not a point mass or a uniform sphere but rather an oblate spheroid due to its spin, which can be roughly approximated using spherical harmonic expansion and considering only the dominant J2 term.

However, if certain perturbations are desired, or particular accuracy is needed, applying closed-form corrections becomes difficult or impossible. In the case of KPS, full atmospheric drag

modeling coupled to attitude (in which the attitude of the spacecraft can significantly affect the frontal drag area and the precise effects of this coupling are desired) requires integrating varying aerodynamic forces and torques over time, and magnetic torque is dependent on Earth's fluctuating magnetic field *and* the satellite's instantaneous attitude. A closed-form solution does not exist; instead, **differential equations** govern the satellite's behavior and can be propagated forward in time via **numerical integration**.

The obvious disadvantage of direct integration is that computation is now required *per unit time propagated*. However, once this price is accepted, more perturbations can be considered with relative ease. It is no longer necessary to apply complex corrections for approximate gravitational effects, for example; instead, simply integrate with the local gravitational field from a spherical harmonic gravity model at each point and let Newton's laws do the rest.

In this mode of propagation, the Keplerian orbital elements are replaced by two Cartesian 3-vectors in the ECI frame: one for **position**, denoted $\vec{r}$, and one for **velocity**, denoted $\vec{v}$. These are known as **state vectors**. See Table 2.

Notice that two 3-vectors mean a total of *six degrees of freedom*, just as with the Keplerian elements! The same fundamental ideas about a satellite are described by both – *where it is, and where it's going if unperturbed* (and if someone specifies a gravitational parameter!). It is thus possible to convert losslessly between Keplerian elements and state vectors.

KPS operates exclusively with state vectors but includes utilities allowing users to convert state vectors to Keplerian elements and vice versa.

Keplerian Elements	State Vectors
a	$\vec{r}\begin{cases} r_x = x \\ r_y = y \\ r_z = z \end{cases}$
e	
i	
Ω	$\vec{v}\begin{cases} v_x = \dot{r}_x = \dot{x} \\ v_y = \dot{r}_y = \dot{y} \\ v_z = \dot{r}_z = \dot{z} \end{cases}$
ω	
ν	

Table 2. *Keplerian elements vs. state vectors.*

The forces considered by KPS are *aerodynamic force* and *gravity*. The acceleration vector due to gravity can be obtained by considering Equation 4 in vector form:

$$\overrightarrow{a_g} = \dot{\vec{v}} = \ddot{\vec{r}} = \frac{GM}{\|r\|^3}\vec{r} \tag{6}$$

Kareem Omar
kareem.omar@uah.edu

Aerodynamic force computation is discussed at length in the Aerodynamics section. The result is divided by satellite mass as per Equation 2 to produce aerodynamic acceleration, which can be summed with the acceleration due to gravity to produce net acceleration.

Background – Attitude Dynamics

The **attitude** of a spacecraft refers to its orientation relative to a specified coordinate system. In KPS, attitude is referenced to the ECI frame. As described in the section on coordinate frames, the Body frame is attached to the spacecraft, translating and rotating with it, so to positively specify the satellite's attitude is to specify the orientation of the Body frame relative to the ECI frame.

In orbital dynamics, the position is specified as a 3-vector pointing from the origin of the ECI frame to the satellite. In attitude dynamics, the situation is more complicated, as one must describe the orientation of the Body frame relative to the ECI frame.

Consider trying to describe this orientation. Since it is an orientation relative to the ECI frame, start in the ECI frame. Look at its axes. Rotate the axes in a controlled manner until they line up with the axes of the Body frame. If one keeps track of the rotations required to transform the ECI frame into the Body frame, one has fully specified the orientation. A body in one frame could be rotated into the other frame, or vice versa, given that information.

The problem now becomes one of how best to represent a rotation numerically. Consider first a rotation by some angle θ about one of the principle axes – x, y, or z. The right-hand rule is once again used to resolve a directional ambiguity. For positive θ, rotation occurs in the direction one's right hand fingers curl when the thumb points in the direction of the axis, i.e. *counterclockwise* if the axis (and thus the thumb) is pointed at one's eyes.

The axis of rotation will not change, of course. The other two principle axes will rotate counter-clockwise by θ. Consider looking down on such a rotation about the z-axis, say, as in Figure 9. A point P, or equivalently, a vector from the origin to P, is shown, with coordinates (x, y) in the old

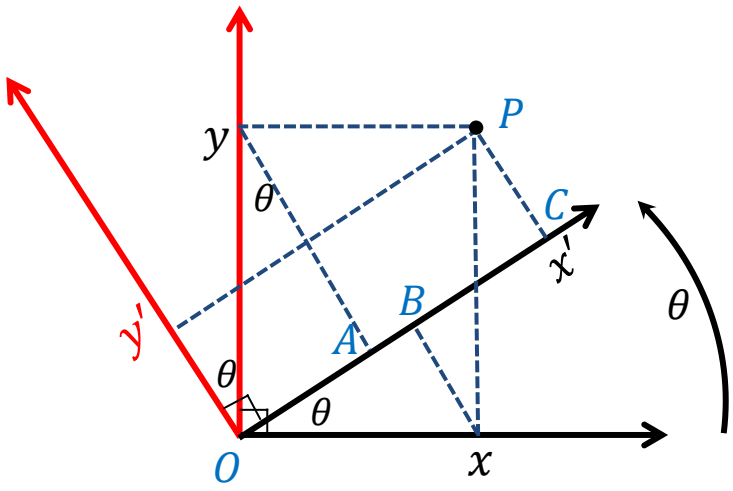

Figure 9. *Rotation of the frame about the z-axis*

frame and coordinates (x', y') in the new frame. Note that the *axes* are being rotated, not the point. Note also that the same result could be achieved by leaving the axes alone and rotating the *point* by $-\theta$ counter-clockwise (i.e. by θ *clockwise*) instead. Rotations of the frame are also known as **alias** or **passive** rotations. Rotations of the vector itself are also known as **alibi** or **active** rotations. This is an ambiguity in rotation schemes that must be specified for a particular rotation sequence before the effect of that sequence can be determined.

Consider Figure 9. Imagine starting the rotation of the axes from 0° and steadily increasing it. At first, x' is identical to x. As rotation begins, x' begins to increase as the moving axis starts to point more directly at P. Similarly, y' begins to decrease as the new y axis points more perpendicularly from P. x' consists of the sum of the **projection** of x onto the x' axis and the projection of y onto the x' axis. Graphically, these are $\overline{OB}$ and $\overline{BC}$, respectively. However, note that $\overline{OA} = \overline{BC}$, and thus $x' = \overline{OA} + \overline{OB}$. These are the opposite and adjacent sides, respectively, to the angle θ of right triangles with hypotenuses equal to x and y, respectively. Thus:

$$x' = x\cos(\theta) + y\sin(\theta) \tag{7}$$

Similar arguments can be made for the y-axis, resulting in:

$$y' = -x\sin(\theta) + y\cos(\theta) \tag{8}$$

The z-axis can be added in as well:

$$\begin{aligned} x' &= x\cos(\theta) + y\sin(\theta) \\ y' &= -x\sin(\theta) + y\cos(\theta) \\ z' &= z \end{aligned} \tag{9}$$

Notice that each new coordinate receives contributions from each of the old coordinates (even though some contributions are zero). Writing this explicitly:

$$x' = x * \cos(\theta) + y * \sin(\theta) + z * 0 \tag{10}$$
$$y' = x * -\sin(\theta) + y * \cos(\theta) + z * 0 \tag{11}$$
$$z' = x * 0 + y * 0 + z * 1 \tag{12}$$

Notice also that whenever new values depend on the sum of contributions from constant multiples of the old values, **matrix multiplication** can be used, as that is precisely the definition of a matrix multiplying a vector! The **operation** of rotating the frame about the z-axis can evidently be expressed as a **rotation matrix**:

Kareem Omar
kareem.omar@uah.edu

$$\boldsymbol{R}_{z_{frame}} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{13}$$

A frame axis a can be rotated by θ into its position in the new frame, a', via:

$$\boldsymbol{a}' = \boldsymbol{R}_{z_{frame}}\boldsymbol{a} \tag{14}$$

A vector, too, can be rotated in this manner – remember to negate θ:

$$\boldsymbol{R}_{z_{vec}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

$$\boldsymbol{v}' = \boldsymbol{R}_{z_{vec}}\boldsymbol{v} \tag{16}$$

Notice something interesting about the coefficients in the rotation matrix $\boldsymbol{R}_{z_{frame}}$. Express them all as cosines:

$$\boldsymbol{R}_{z_{frame}} = \begin{bmatrix} \cos(\theta) & \cos(90° - \theta) & \cos(90°) \\ \cos(90° + \theta) & \cos(\theta) & \cos(90°) \\ \cos(90°) & \cos(90°) & \cos(0°) \end{bmatrix} \tag{17}$$

Now name the columns after the original axes, and the rows after the new axes:

$$\begin{matrix} \\ x' \\ y' \\ z' \end{matrix} \begin{matrix} \begin{matrix} x & y & z \end{matrix} \\ \begin{bmatrix} \cos(\theta) & \cos(90° - \theta) & \cos(90°) \\ \cos(90° + \theta) & \cos(\theta) & \cos(90°) \\ \cos(90°) & \cos(90°) & \cos(0°) \end{bmatrix} \end{matrix} \tag{18}$$

Each element of the matrix is simply the *cosine* of the angle between the old axis from its column and the new axis from its row! These elements are known as **direction cosines**, and thus the whole rotation matrix is sometimes known as a **direction cosine matrix** (**DCM**).

In this manner, a DCM encodes information about how all three axes relate to each other. Note that the sign of the elements does not matter since cosine is an even function.

For completeness the rotation matrices for rotation about the x- and y-axes are also presented. The vector (active rotation) formulations are used:

$$\boldsymbol{R}_{x_{vec}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{19}$$

$$\boldsymbol{R}_{y_{vec}} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{20}$$

How can these rotations be combined to achieve an arbitrary orientation? It is intuitive that in three dimensions, one can do so by rotating *three* times about principal axes. One may choose to either repeatedly rotate about the original axes **(extrinsic rotation)**, or rotate about an original axis, then about one of the new principle axes, then about one of the again-new principle axes **(intrinsic rotation)**. In either case, if the three axes of rotation all differ (e.g. x-y-z, or y-x-z, or z-y-x, etc.) the angles in the rotation sequence are known as **Tait-Bryan Angles**. If the first and third axes repeat (e.g. x-y-x, or z-y-z), they are known as **Proper Euler angles**. Note that if during a series of rotations, one axis rotates into another, such that they are no longer independent, a DoF is lost. This phenomenon is known as **gimbal lock** and can be quite problematic. Having to mitigate gimbal lock is one of the primary disadvantages of Euler angle representations.

Nonetheless, a series of three Euler angles provides a mechanism for specifying an arbitrary orientation.

Note, however, that if one stacks, or **concatenates**, three such rotations in sequence, by multiplying the three matrices, a single 3x3 matrix is created. Another way of thinking about this is that it is still possible to define the angles between the very first initial axes and the very last final axes unambiguously. Because this is all that is required to create a DCM, *a single rotation matrix can describe any rotation, not just a rotation around a principal axis*.

This provides the second mechanism for specifying an orientation – a single 3x3 *rotation matrix*.

Furthermore, such a rotation can be done directly, i.e. in a single rotational motion by some angle about a single axis. This is known as **Euler's rotation theorem**. All that is necessary, then, to define an orientation relative to a known frame is to specify an **axis** of rotation and an **angle** by which to rotate about that axis.

If one represents this axis as a **unit vector**, a vector with magnitude of unity, the constraints on the combined axis and angle suggest a particular kind of mathematical representation known as a **quaternion**, and specifically a **unit quaternion**, or **versor**. A complete discussion of quaternion algebra and the reasons for its suitability for representing rotations is beyond the scope of this

paper. Nevertheless, the basics of their application to rotation will be covered here for the interested reader.

Quaternions are an extension of the complex numbers that supports *three* imaginary components, $i, j,$ and k. The fundamental identities linking them, famously carved onto a bridge by their discoverer, William Rowan Hamilton, in 1843 in a flash of inspiration, are as follows:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{21}$$

A quaternion can be written in the form:

$$q = \{a, b, c, d\} = a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k} \tag{22}$$

It can also be written using Euler's Formula:

$$q = e^{\frac{\theta}{2}(b\boldsymbol{i}+c\boldsymbol{j}+d\boldsymbol{k})} =$$
$$\cos\left(\frac{\theta}{2}\right) + (b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k})\sin\left(\frac{\theta}{2}\right) = \left\{\cos\left(\frac{\theta}{2}\right), b\sin\left(\frac{\theta}{2}\right), c\sin\left(\frac{\theta}{2}\right), d\sin\left(\frac{\theta}{2}\right)\right\} \tag{23}$$

The grouping {b, c, d} and the emergence of an angle θ suggest a rotation, and, indeed, such a quaternion can be used to rotate about axis {b, c, d} by angle θ!

Such a rotation is performed as follows:

$$\boldsymbol{v}' = qvq^{-1} \tag{24}$$

The opposite rotation, i.e. about the same axis but by the negative of the angle, is given by:

$$\boldsymbol{v}' = q^{-1}vq \tag{25}$$

Quaternion multiplication is implemented via a **Hamilton product**. q^{-1} refers to the **quaternion inverse**, which for versors is identical to the **quaternion conjugate**, q^*, in which the real part is unchanged but the imaginary components are negated:

$$q^* = a - b\boldsymbol{i} - c\boldsymbol{j} - d\boldsymbol{k} \tag{26}$$

In Equation 24, the vector v is treated as a quaternion with no real component, i.e. $\{0, v_x, v_y, v_z\}$ for the multiplication process.

Quaternions, then, provide a third mechanism for specifying an orientation. There are other ways, but the three aforementioned methods are the most common in aerospace, computer graphics, virtual reality, robotics, and other fields. Each has certain advantages, as summarized very briefly in Table 3. The Apollo missions used Euler angles for guidance and control[6]. Most modern satellite attitude determination, control, and propagation software, including KPS, uses quaternions for specifying the attitude of the spacecraft, and for internal computation. KPS occasionally makes use of DCMs for other functions.

Formalism	DoF	Storage (Elements)	Strengths	Weaknesses
Rotation Matrices (DCMs)	3	9	- Simple implementation - Simple rotation concatenation	- Verbose - Numerically unstable - Difficult renormalization
Euler Angles	3	3	- Concise storage - Intuitive	- Gimbal lock - Slow
Quaternions	3	4	- Concise storage - Simple rotation concatenation - Simple renormalization - Free of discontinuities - Simple differentiation	

Table 3. *Comparison of rotational formalisms*

In KPS, attitude is specified as a quaternion which can rotate the ECI frame into the Body frame, or equivalently, rotate a vector in the Body frame into the ECI frame.

Computing the derivative of such an attitude is not as straightforward as the equivalent problem in orbital dynamics, where the derivative of the position is simply the velocity, but will nonetheless be required for propagation as explained in the Numerical Integration section.

Consider stepping forward in discrete steps from some initial attitude q_0, where no external torques are being applied, so the angular velocity is constant, and thus the change in attitude is a constant per unit time.

After one step, say the attitude is $q(1) = qq_0$, where q is the rotation that occurred during that single step. After two steps the attitude is $q(2) = qqq_0 = q^2q_0$. In general:

$$q(t) = q^t q_0 \tag{27}$$

Writing q in its Euler formulation as in Equation 23:

$$q = e^{\frac{\theta}{2}(b\boldsymbol{i}+c\boldsymbol{j}+d\boldsymbol{k})}$$

Thus, exponentiating and replacing $\{b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}\}$ with $\boldsymbol{a}$ for simplicity:

$$q^t = e^{t\frac{\theta}{2}\boldsymbol{a}} \tag{28}$$

$$q(t) = q_0 e^{t\frac{\theta}{2}\boldsymbol{a}} \tag{29}$$

Taking the derivative:

$$\dot{q}(t) = \frac{\theta}{2}\boldsymbol{a}q_0\, e^{t\frac{\theta}{2}\boldsymbol{a}} = \frac{\theta}{2}\boldsymbol{a}q(t) = \frac{1}{2}\theta\boldsymbol{a}q(t) \tag{30}$$

But if at each step, a rotation of q occurs, and a rotation of q is a rotation of θ about $\boldsymbol{a}$, then θ rotation occurs per unit time about $\boldsymbol{a}$, so $\theta\boldsymbol{a}$ is the **angular velocity**, ω:

$$\theta\boldsymbol{a} = \omega \tag{31}$$

Thus:

$$\dot{q} = \frac{1}{2}\omega q \tag{32}$$

Note that q has been defined as rotating the ECI frame into the Body frame. It starts in the ECI frame. Therefore, the ω in Equation 31 is *in the ECI frame as well.* If the angular velocity is to be specified in the Body frame:

$$\omega_b = q^{-1}\omega q \tag{33}$$

$$\omega = q\omega_b q^{-1} \tag{34}$$

Substituting Equation 34 into Equation 32:

$$\dot{q} = \frac{1}{2}(q\omega_b q^{-1})q = \frac{1}{2}q\omega_b(q^{-1}q)$$

$$\boxed{\dot{q} = \frac{1}{2}q\omega_b} \tag{35}$$

KPS uses Equation 35 to compute the quaternion derivative, as it tracks angular velocity in the Body frame.

Just as the first derivative is more difficult in attitude than in orbital dynamics, so, too, is the second derivative. In orbital dynamics, the forces are simply divided by mass to obtain an acceleration, which can be integrated to directly obtain velocity.

In attitude dynamics, obtaining angular acceleration begins with the rotational equivalent of Newton's Second Law[7]:

$$\tau = I\alpha = I\dot{\omega} \tag{36}$$

However, this *only applies in an inertial frame!* Therefore, the **angular momentum relations** will be used first, as they are valid in a rotating frame:

$$L = I\omega \tag{37}$$
$$\dot{L} = I\dot{\omega} \tag{38}$$

The **rate of change transport theorem** will be applied to transport this quantity from the rotating frame into an *inertial frame which instantaneously coincides with the Body frame*. The theorem states, for any vector v:

$$\dot{v}_{inertial} = \dot{v} + \omega \times v \tag{39}$$

In Equation 39, $\times$ refers to a cross product, and ω is the angular velocity of the rotating frame relative to the inertial frame. Recall that the Body frame is attached to the satellite, so the satellite's angular velocity is also the angular velocity of the Body frame relative to any inertial frame, including ECI and the instantaneous inertial frame referenced here.

Thus for angular momentum:

$$\dot{L}_{inertial} = \dot{L} + \omega \times L \tag{40}$$

Substituting in from Equations 37 and 38:

$$I\dot{\omega}_{inertial} = I\dot{\omega} + \omega \times I\omega \tag{41}$$

In the inertial frame on the left-hand side, Equation 36 is valid and can be substituted:

$$\tau = I\dot{\omega} + \omega \times I\omega \tag{42}$$

Solving for $\dot{\omega}$, which equals α, the angular acceleration:

$$I\dot{\omega} = \tau - \omega \times I\omega \quad (43)$$

$$\boxed{\alpha = \dot{\omega} = I^{-1}(\tau - \omega \times I\omega)} \quad (44)$$

Equation 44 is used by KPS to compute angular acceleration based on applied torques, just as Newton's Second Law is used to compute acceleration based on applied forces for orbital dynamics.

Three such applied torques are considered in KPS:

1. Aerodynamic torque, or **aerotorque**, which is discussed at length in the Aerodynamics section.
2. Magnetic torque**,** or **magnetorque**, from onboard magnetorquers which react with Earth's magnetic field. This is discussed at length in the Magnetorque section.
3. **Gravity gradient torque**. This is discussed at length in the Gravity Gradient Torque section.

Table 4 summarizes orbital and attitude dynamics in the context of external impetuses.

	Orbital	Attitude
External Impetus	Forces, F	Torques, τ
Specific Examples in KPS	Aerodynamic Force, Gravity	Aerodynamic Torque, Magnetorque, Gravity Gradient Torque

Table 4. *Comparison of orbital and attitude external impetuses*

Background – Numerical Integration

Numerical techniques exist to compute to high accuracy a numerical solution for the integral of an **ordinary differential equation (ODE)** given **initial conditions (ICs)**. If one knows how fast a function is changing, and where it started, one can plot the trajectory of the function.

The simplest mechanism for this is known as **Euler's method**. If rate of change is *change per unit time*, then multiply an amount of time by that rate gives you the *change*. Add this change on to the initial condition, and the function has been stepped forward by that amount of time. This is the *exact* solution for functions with constant derivatives, i.e. **linear** functions. For other functions, some error is introduced because the rate of change varies continuously. Using more time steps, each a shorter amount of time, will more accurately approximate the function's trajectory.

Euler's method is actually the simplest method in a class of integrators known as **Runge-Kutta (RK) methods**. It is first-order method, meaning that the total error is proportional to the step size. This is unacceptable for precision work, and higher-order RK methods can do much better.

Perhaps the most commonly used is RK4, a fourth-order method. Higher orders are available, but the computational complexity increases so much above fourth-order that it is often faster to remain at fourth-order and simply decrease the step size for more accuracy. RK4 computes multiple predictions based on the (estimated) slopes at different points within the step interval, then computes a final result using a weighted average of these predictions.

This is a good start, and in fact is sufficient for most ODE problems. However, if particularly fast and robust propagation is needed, fixed step size is problematic. Many ODEs have regions of rapid change, where small step sizes are needed, and regions of relative stability, where a larger step size can be used with no loss of precision. **Adaptive ODE solvers** take advantage of this by computing the next step with two different orders and comparing the results. If they differ by too much, the step size is reduced and the step is repeated. If they differ by very little, the next step size is increased slightly. In this manner, accuracy is preserved but computation speed for a typical ODE increases dramatically.

Perhaps the most common adaptive solver is the **Runge-Kutta Dormand-Prince (RKDP)** method, which computes fourth- and fifth-order solutions. The coefficients are carefully chosen to allow maximum re-use of computations between the two orders, and the last stage of one step is recycled in the first stage of the next step for additional savings. This is one of the integrators offered by KPS.

Although adaptive RK methods are quite good, for even better performance, further optimizations can be made. An RK method computes several intermediate values, but even with RKDP (where the last value is re-used), most are discarded after that step. The next step begins without any knowledge of the previous step, except the recommended step size.

A family of solvers exists to take advantage of stored information from previous steps. They are known as **linear multistep methods**. They can be thought of as similar to RK methods, except that some of the intermediate computations are actually borrowed from previous steps. This improves performance, *particularly when the cost of ODE evaluation is high* – which it is for attitude propagation due to the aerodynamics simulation.

The catch with linear multistep methods is difficulty of implementation! Linear multistep methods are far more complicated to develop. KPS offers one such integrator for users, known as an **Adams-Bashforth-Moulton (ABM)** solver.

Special consideration is needed in the case of **stiff** systems. Although a precise definition is elusive, stiff systems cause trouble for most solvers due to sharply varying derivatives. Special **implicit** solvers can be deployed to attack particularly stiff systems.

Kareem Omar
kareem.omar@uah.edu

Stiff systems are often confused with **chaotic** systems. A chaotic system is one in which even very minimal changes in the exact value of the integration result (due, for example, to adjusting relative error thresholds in a solver) rapidly cause divergence to a different-looking solution further along in the integration process.

Orbital and attitude ODEs are *chaotic**, but only moderately stiff***. The author has experimented at length with hand-tuned stiff solvers; however, they did not surpass the aforementioned ABM solver in accuracy or performance. Due to the chaotic nature of the system, however, users should not be alarmed if slight changes to error threshold settings in KPS cause somewhat different-looking output. This is akin to a slight unaccounted-for torque on the satellite causing its exact orientation to differ, say, 10,000 seconds later – the overall trends of the satellite's state and stability should agree, but it is not surprising that minor differences may accumulate. Conditions surrounding the satellite, such as the precise density, are not known with enough precision to require extreme accuracy from the integrators, and even the accumulation of machine precision truncation error is easily enough to cause a chaotic divergence over the course of a mission.

In the case of orbital dynamics, acceleration can be obtained from the forces applied to the satellite. Position is desired. Acceleration is the *second* derivative of position, however. The first derivative is velocity. The solution is to instead integrate a **derivative vector** that jointly considers position and velocity.

Similarly, in attitude dynamics, angular acceleration can be obtained from the torques applied to the satellite. Orientation is desired. Angular acceleration is the second derivative of orientation. Once again a joint vector is integrated.

Figure 10 shows the initial conditions and how the integration process is accomplished for orbital propagation. From an initial position and velocity, the derivative vector is produced. The derivative of position is simply the velocity, and the derivative of velocity is acceleration, obtained from the forces acting on the satellite. This joint derivative vector is numerically integrated to advance the satellite forward in time by one step. The process then repeats: an updated derivative vector is produced and integrated to advance another step.

Initial Conditions (Step 0)	Produce Derivative Vector (ODE)	Integrate (Step 1)	Produce Derivative Vector (ODE)	Integrate (Step 2)
r_0	v_0	r_1	v_1	r_2
v_0	$a_0 = \frac{F_0}{m}$	v_1	$a_1 = \frac{F_1}{m}$	v_2

External Forces, F_0 → $a_0 = \frac{F_0}{m}$

External Forces, F_1 → $a_1 = \frac{F_1}{m}$

Figure 10. *Orbital propagation.*

Figure 11 shows the same process for attitude dynamics.

Initial Conditions (Step 0)	Produce Derivative Vector (ODE)	Integrate (Step 1)	Produce Derivative Vector (ODE)	Integrate (Step 2)
q_0	$\frac{1}{2} q_0 \omega_0$	q_1	$\frac{1}{2} q_1 \omega_1$	q_2
ω_0	$\alpha_0 = I^{-1}(\tau_0 - \omega_0 \times I\omega_0)$	ω_1	$\alpha_1 = I^{-1}(\tau_1 - \omega_1 \times I\omega_1)$	ω_2

External Torques, τ_0 → $\alpha_0 = I^{-1}(\tau_0 - \omega_0 \times I\omega_0)$

External Torques, τ_1 → $\alpha_1 = I^{-1}(\tau_1 - \omega_1 \times I\omega_1)$

Figure 11. *Attitude propagation.*

Background – Gravitational Modeling

KPS offers a variety of gravity models from which users can select. In order of increasing accuracy (and decreasing performance!):

- **POINT** – this is the simplest model. It refers to a **point mass model** in which gravity simply pulls directly toward the center of the Earth no matter where the satellite is, as if the Earth were a point mass or a perfectly uniform sphere.
- **WGS84** – this model considers the gravitational acceleration that would be produced if the Earth were an ellipsoid that more closely matches its actual shape. This ellipsoid model is known as the **World Geodetic System 1984 (WGS84)**[8] and improves the gravitational model without significant performance penalty.
- **EGM84** – this model, released by the National Geospatial-Intelligence Agency, is known as the **Earth Gravitational Model 1984**[8]. It is based on the WGS84 ellipsoid as

well, but it expands further variations and disturbances in the gravitational field due to variations in Earth's shape and density using spherical harmonics. It is more accurate but much slower.

- **EGM96** – this model is an update to the EGM84 model containing more spherical harmonic terms[8]. It is even more accurate but even slower.
- **EGM2008** – this model is an update to the EGM96 model containing *significantly* more spherical harmonic terms[8]. It is highly accurate but *very* slow.

These options are provided to allow users to select the tradeoff between performance and accuracy that suits their application, or to experiment with different models to learn about the effects of gravitational perturbations on satellites and on propagator performance.

Background – Magnetic Modeling

Satellites may wish to control their attitude by reacting against Earth's magnetic field using magnetorque. This, of course, requires knowledge of the Earth's magnetic field pseudovector at the satellite's position.

To this end, KPS offers a variety of magnetic models from which users can select. Unlike Earth's gravity, Earth's magnetism changes fairly rapidly, so a date, or at least a year, is specified during initialization of the model, and models are typically intended only for a particular range of years. In order of increasing accuracy (and decreasing performance!):

- **WMM2010** – the National Geospatial-Intelligence Agency **World Magnetic Model 2010**[9]. It expands Earth's magnetic field using spherical harmonics. It is intended to model the years 2010-2015. It has the fewest spherical harmonic terms and is thus the fastest model.
- **WMM2015** – this model has the same number of terms as WMM2010 and so it shares the WMM2010's speed and accuracy[9]. It updates the WMM for the years 2015-2020.
- **IGRF11** – the International Association of Geomagnetism and Aeronomy (IAGA) **International Geomagnetic Reference Field 2011**[9]. It is intended to model the years 1900-2015 and has one more term than the WMM.
- **IGRF12** – this model has the same number of terms as IGRF11 and so it shares the IGRF11's speed and accuracy[9]. It extends the IGRF's validity to the years 1900-2020.
- **EMM2010** – the NOAA **Enhanced Magnetic Model 2010**[9]. It enhances the WMM by adding crustal magnetic fields and *significantly* more spherical harmonic terms. It is intended to model the years 2010-2015. It is highly accurate but *very* slow.
- **EMM2015** – the NOAA **Enhanced Magnetic Model 2015**[9]. It extends the EMM's validity to the years 2000-2020.

As with the gravitational models, these options are provided to allow users to select the tradeoff between performance and accuracy that suits their application, or to experiment and learn about variations in Earth's magnetic field.

Background – Magnetorque

Satellites which wish to stabilize, passively or actively, in a particular orientation must be able to offload angular momentum received from satellite deployment or aerodynamic torques. Reacting against Earth's magnetic field can accomplish this without resorting to chemical thrusters.

The goal, then, is to produce a magnetic torque that opposes the satellite's angular velocity, diminishing it over time. Magnetorquers are essentially coils of wire through which an electric current is passed to generate a magnetic field to react with Earth's. The **magnetic moment** generated by such a coil is[10]:

$$\mu = nIA \tag{45}$$

n is the number of loops of wire, I is the current, and A is the loop's area. The torque produced by such a loop when exposed to Earth's magnetic field, β, is[10]:

$$\tau = \mu \times \beta \tag{46}$$

Notice that this torque will always be perpendicular to both the moment and the Earth's magnetic field. An arbitrary magnetic moment can be produced by three magnetorquers configured such that one is in each of the three Body axes. However, Earth's magnetic field cannot be changed except by waiting for the satellite to rotate (which changes the magnetic field vector in the Body axes) or travel to a different part of its orbit (where the field itself may differ somewhat). For a given orientation and position, then, the field cannot be altered, so the torque produced *will always reside somewhere in the plane perpendicular to Earth's magnetic field*.

This means that perfect opposition to the satellite's angular velocity cannot be achieved unless the angular velocity is also perpendicular to β. However, it suffices to oppose the angular velocity as much as possible given this restriction. Over time, as the satellite rotates and orbits, successful angular velocity cancellation can be achieved.

The ideal torque would simply directly oppose the angular velocity by some (positive) gain factor k:

$$\tau_{ideal} = -k\omega \tag{47}$$

Kareem Omar
kareem.omar@uah.edu

With the limitation of being perpendicular to Earth's magnetic field, the best one can do is request a magnetic moment perpendicular to both β and ω:

$$\mu = -k(\beta \times \omega) \tag{48}$$

The resulting torque will be:

$$\tau = \mu \times \beta \tag{49}$$

$$\boxed{\tau = \big(-k(\beta \times \omega)\big) \times \beta} \tag{50}$$

This torque will be perpendicular to μ and β and still oppose ω as much as possible. Note that in the form of Equation 50, it is proportional to the square of the magnitude of β; if this is not desired, it can simply be divided by the square of the magnitude.

Equation 50 is used in KPS to generate magnetic torque, and the gain factor, k, is specified by the user.

Note that if implementing an actual on-board controller for a satellite's attitude control system, the exact ω need not be known precisely. The rate of change of the magnetic field suffices as an approximation, since over short periods of time the *actual* field doesn't change much, so the *apparent* change is due mainly to the satellite's rotation:

$$\beta \times \omega = \dot{\beta} \approx \frac{\Delta\beta}{\Delta t} = \frac{\beta_2 - \beta_1}{t_2 - t_1} \tag{51}$$

$$\mu \approx -k\left(\frac{\beta_2 - \beta_1}{t_2 - t_1}\right) \tag{52}$$

In this manner, an onboard controller can sample the magnetic field twice using magnetometers, a short time apart, and obtain an approximate magnetic moment to command from its magnetorquers.

Background – Atmospheric Modeling

Atmospheric modeling is a vast and complicated field. Density and pressure vary rapidly with multiple cycles of different periods, from one day to one year to one solar cycle. Myriad factors influence the properties of air at different altitudes, and at sufficiently high altitudes, the composition itself begins to change. Its components occur in abnormal quantities and ionize. Consequently, the most accurate models require frequent inputs of solar activity and other corrective data, and are thus only accurate for the short time period specified in this data.

Because KPS is capable of rapidly simulating years of orbit, at present, such models are not an excellent fit, and approximate density modeling is sufficient. Time-specific models such as those described above may be supported in a future version of KPS. Currently, KPS uses the **U.S. 1976 Standard Atmosphere** in both its low- and high-altitude modes for altitudes from 0 to 1000 km. Above 1000 km, atmospheric effects are very slight, and are ignored. Figure 12 shows the US1976 density vs. altitude[11]. The descent is quite rapid. The author chooses to handle this by displaying separate linear plots rather than using a log plot to retain a sense of how rapidly the density drops.

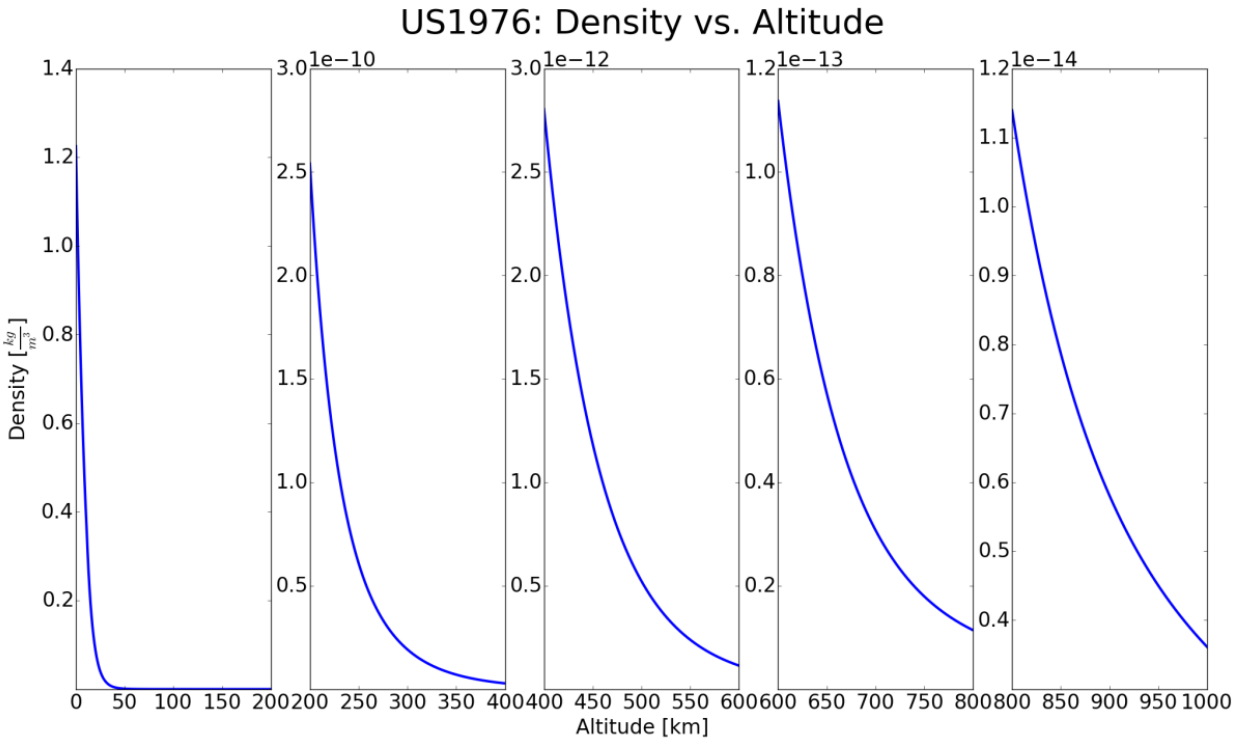

Figure 12. *US1976 Density vs. altitude.*

Background – Aerodynamics

As the section on atmospheric modeling might have suggested, fluid dynamics is a complex field. Properly modeling the forces and torques on a satellite is no small matter, but with some assumptions made, the author has implemented this functionality in two different manners in the hope of provided both utility and instruction.

The density is known from the atmospheric model. The **mean free path** is assumed to be quite large[12], meaning that atmospheric particles are sufficiently sparse that they hardly interact with each other, only with the satellite. This is critical, as at sea level, complex flows develop due to the vast number of collisions made between particles every second. In orbit, particles can be

considered separate entities with which the satellite collides. These particles themselves can have considerable random velocity, but this quantity is zero on average.

Consider a flat plate with the flat face oriented perpendicular to the direction of travel, moving with velocity v. View this system from a frame moving with the plate, such that the plate appears stationary and the *particles* are approaching with velocity v. From this perspective, incoming particles collide and are assumed to undergo **specular reflection**, meaning that they depart with velocity v in the opposite direction.

Linear momentum is defined by:

$$p = mv \tag{53}$$

The particle's initial momentum is mv, and after impact it is $m(-v) = -mv$, so the change in each particle's momentum on collision is:

$$\Delta p = mv - (-mv) = 2mv \tag{54}$$

By conservation of momentum, the satellite *loses* the same amount of momentum from each collision. If σ is the number of particles per unit volume (the particle density), A is the area of the plate, and dt is the amount of time considered, then q is the total number of particle collisions that occur in that time period:

$$q = \sigma v A(dt) \tag{55}$$

The total momentum transferred over the time period dt is:

$$dp_{total} = 2mvq = 2mv\sigma vA(dt) = 2m\sigma Av^2(dt) \tag{56}$$

Now, $m\sigma$ is just the mass density, ρ, so:

$$dp_{total} = 2\rho Av^2(dt) \tag{57}$$

Expressing Newton's Second Law in derivative form:

$$F = \frac{dp}{dt} \tag{58}$$

$$Fdt = dp \tag{59}$$

Substituting Equation 57 into Equation 59:

$$Fdt = 2\rho A v^2 (dt) \quad (60)$$

$$F = 2\rho A v^2 \quad (61)$$

Equation 61 allows the computation of forces on a flat plate given its area and velocity and the atmospheric density.

It remains to generalize this formula to faces not necessarily perpendicular to the velocity vector. If a particle strikes such a face and reflects off at the same angle as its angle of incidence, one needs only consider the component of its velocity perpendicular to the face. Thus:

$$\boxed{F = 2\rho A v_{\perp}{}^2} \quad (62)$$

This force is exerted in the direction opposite the face normal.

KPS can operate in **Analytical** mode, in which the three-dimensional collection of polygons describing the satellite is projected into 2-D and **clipped** with **z-ordering**, meaning the geometry is analyzed to analytically determine all portions of the satellite polygons visible to the oncoming air, taking into account occlusion by the other polygons of the body. The force of Equation 62 is applied to the **centroid** of each polygon presentation surface.

For edification and also for even greater performance (at the cost of accuracy), KPS can also generates grids of **test particles** to collide with the satellite, each representing a small panel of area to sample. Note that these test particles are simply for evenly colliding with the geometry of the satellite and are unrelated to physical atmospheric particles. Each of these test particles strikes the satellite, and Equation 62 is used to compute the force vector added by that individual particle, with A equal to the area of the grid represented by that particle (i.e. the square of the linear pitch of the particle grid). These forces are summed to compute an approximation of the net force on the satellite, the accuracy of which increases with the number of particles.

If the satellite's center of mass is not on the **line of action** of any of these collisions, aerotorque is generated in addition to force. The torque from a single collision is computed in the usual way:

$$\boxed{\tau = r \times F} \quad (63)$$

r in this context is the vector from the center of mass of the satellite to the collision site. These torques are summed to compute the net torque on the satellite.

Dividing the aeroforce by the satellite mass and the square of the velocity results in a metric known as the **starred ballistic coefficient**:

$$B^* = \frac{F}{mv^2} \tag{64}$$

This coefficient is one of many parameters that is output by KPS during simulation and is available for plotting in real-time or after a simulation run has finished, as is explained at length in the Visualization section.

Background – Gravity Gradient Torques

Consider for purposes of illustration a long, rigid cylinder in orbit, pointed nadir, as in Figure 13. Recall from Equation 1 that the force of gravity diminishes as r^2. Thus, the force on the end of the cylinder nearest Earth's surface would be stronger than the force on the far end. The cylinder would experience a net force trying to rip it apart! This remains true even for small satellites, although the effect is, of course, quite small. Consider a smaller version of such a cylinder. Now disturb the cylinder's equilibrium by slightly rotating it, as in Figure 14a. The rotation is too small to make a significant difference in the *direction* of the field – consider the force at all points on the cylinder to still be uniformly "down" in the figure. However, the force is still a gradient that increases toward the bottom of the cylinder. For simplicity, Figure 14a only shows this effect at the ends and middle of the cylinder. Figure 14b shows the *net* forces by subtracting the middle's average force from all three points. Figure 14c breaks these forces into components along the long axis of the cylinder, and perpendicular to

Figure 13. *Gravitational force is stronger at the end of the cylinder nearest Earth.*

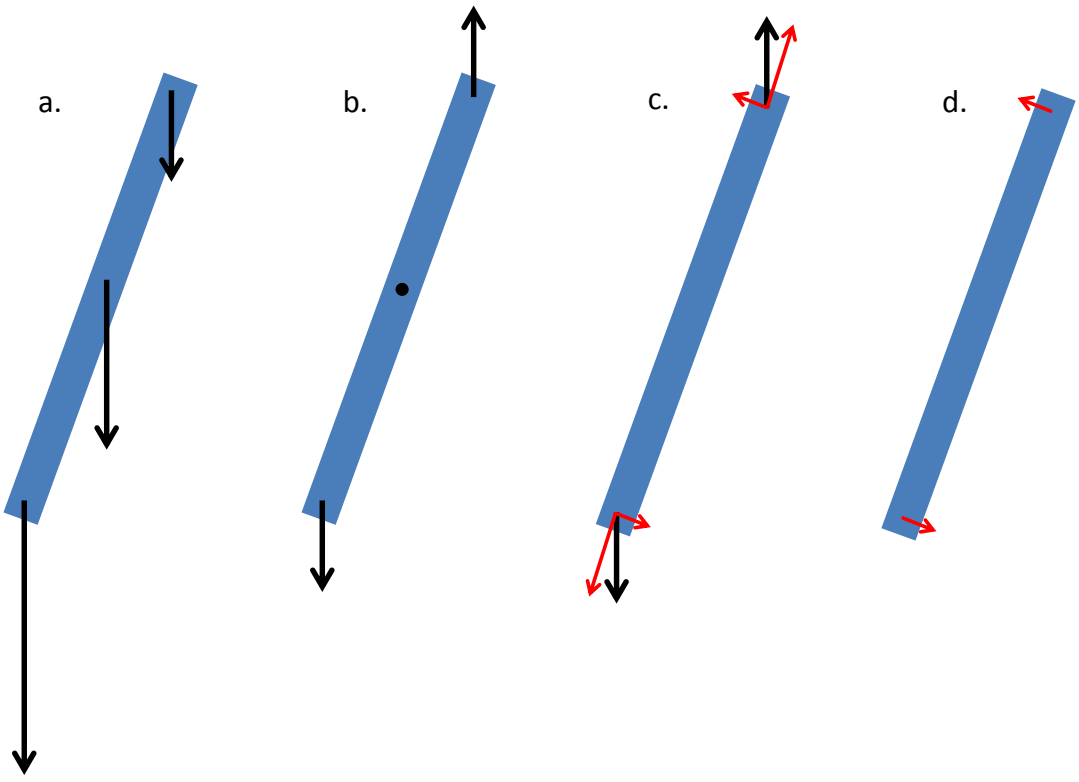

Figure 14. *Gravity gradient torque.*

it. The forces along the cylinder on either end *cancel*, leaving just the perpendicular components as seen in Figure 14d. A counter-clockwise torque is produced!

The author wishes to clarify once more that this effect is *not* due to the forces on the two ends being in different *directions*. All three arrows in Figure 14a point in almost exactly the same direction. However, the net torques on the two ends are "up and down", resulting in a component perpendicular to the long axis of the cylinder that is in opposite directions for the ends. This confers a small torque that attempts to *align the long axis with the nadir vector*.

Torques of this kind are small for most satellites, but they do exist. They can be approximated based on the moment of inertia matrix of a satellite:

$$\boxed{\begin{aligned}\tau_x &= \frac{3\mu}{\|r\|^5} r_{b_y} r_{b_z} [I_z - I_y] \\ \tau_y &= \frac{3\mu}{\|r\|^5} r_{b_z} r_{b_x} [I_x - I_z] \\ \tau_z &= \frac{3\mu}{\|r\|^5} r_{b_x} r_{b_y} [I_y - I_x]\end{aligned}} \tag{65}$$

r_b is the satellite position vector *in the Body frame*. Equation 65 is used by KPS to compute gravity gradient torques.

For a proof of Equation 65, see *Orbital Mechanics for Engineering Students* by Howard Curtis, published by Butterworth-Heinemann press[12].

KPS Usage – Overview

KPS is the software infrastructure written by the author. It consists of one **solution** which contains seven **projects**:

1. **KPS** – the main propagator
2. **KPS_GenPoly** – a utility to easily generate a satellite configuration from basic polygons
3. **KPS_PlotPoly** – a utility to examine a polygon generated by KPS_GenPoly in 3-D
4. **KPS_GenOrbit** – a utility to easily produce initial state vectors for a desired orbit
5. **KPS_Kepler2State** – a utility to convert Keplerian orbital elements into state vectors
6. **KPS_State2Kepler** – a utility to convert state vectors into Keplerian orbital elements
7. **KPS_Vis** – a utility to visualize propagation results in real-time or afterward

The main propagator is written in C++ for performance and can execute on both Microsoft® Windows® and Linux. The author provides a Microsoft® Visual Studio® 2013 solution for viewing or compiling the source code on Windows®, and a Makefile for compiling with g++ on

Linux. The author also provides precompiled binaries, however, so *compiling is not necessary to run KPS*.

The source code and/or binaries can be obtained from the author's GitHub, at https://github.com/komrad36/KPS.

The other six projects are each provided in both MATLAB® *and* Python for convenience. The functionality provided is identical across both languages. Furthermore, the MATLAB® scripts are compatible with GNU Octave, so a user has three different options, including fully open source alternatives, to run or, for the interested programmer, to study.

The author believes strongly in the value of open-sourcing this software in its entirety. Other dynamics and orbital packages exist, but the author emphasizes the uniqueness of a joint orbital and attitude propagation system that is flexible, efficient, and fully **free and open-source (FOSS)** for education, execution, and modification.

KPS Usage – KPS Main Propagator

The main propagator is run with two arguments: a **polygon file**, and a **configuration file**. For example:

```
> KPS poly.kps config.kps
```

The polygon file contains the satellite body geometry, as will be discussed further in the section on KPS_GenPoly. The configuration file specifies 21 parameters required by KPS to operate, in any order. Blank lines are ignored. Comments can be entered by beginning a line with #. A sample configuration file looks like this:

```
# Sample configuration file for KPS
# https://github.com/komrad36/KPS
MAG_GAIN = 25e3
TIME_SINCE_EPOCH_AT_DEPLOY = 0.0
GRAV_MODEL = wgs84
PROPAGATOR = abm
AERO_MODE = ANALYTICAL
ABS_TOL = 1e-6
REL_TOL = 2.3e-14
AERO_PITCH = 0.01
MAX_STEP_SIZE = 26000
MAG_MODEL = wmm2015
```

```
MAG_YEAR = 2016
BINARY_OUTPUT = true
REALTIME_OUTPUT = true
SAT_CM = 0.0, 0.0, 0.0
SAT_INIT_POS = 6871000, 0, 0
SAT_INIT_V = 0, 5385.7217960362, 5385.7217960362
SAT_INIT_Q = 1.0, 0.0, 0.0, 0.0
SAT_INIT_W = 0.008, -0.05, 0.01
SAT_MASS = 8.0
SAT_MOI = 0.0667, 0.0, 0.0, 0.0, 0.0867, 0.0, 0.0, 0.0, 0.0333
TIME_SPAN = 80000
```

Specific details on each option are available in the README. Briefly, the configuration file allows users to select a gravitational model, a magnetic model, initial satellite state, and various other parameters.

In order to facilitate use of and experimentation with KPS, the author has designed the main propagator to be entirely self-contained so *no external libraries must be installed to run the software*. The only other requirement to run KPS is the data for the magnetic and gravitational model specified in the configuration file.

Magnetic models are provided by GeographicLib in the form of a small download available here: http://geographiclib.sourceforge.net/html/magnetic.html#magneticinst

If the user specifies a gravity model of POINT, no download is needed. For a more complex model, the appropriate data can similarly be downloaded from here: http://geographiclib.sourceforge.net/html/gravity.html#gravityinst

The user is now ready to run KPS:

```
> KPS poly.kps config.kps
```

The system responds by echoing the parameters and other diagnostic information. If all initialization is successful, the system reports:

```
KPS READY.
```

Propagation begins:

```
Propagating:
 Step 37878 | 17042.926685 sec
```

The counter updates in real-time with the current step count and the number of seconds of orbit simulated. Propagation continues until the `TIME_SPAN` requested in the configuration file is achieved, *or* until satellite **deorbit** occurs, whichever comes first. Upon completion, the system displays:

```
Step 25054 | 10000.000000 sec
Propagation complete in 5.522 sec realtime.
```

Or, if deorbit occurred:

```
Step 9448 | 63.861708 sec
Satellite DEORBITED! Propagation complete in 3.423 sec realtime.
```

As KPS propagates, it constantly writes out 18 satellite parameters to disk for real-time or later visualization, as will be discussed at length in the Visualization section.

KPS Usage – Polygon Generator

The polygon files expected by KPS specify the geometry of the satellite as a series of polygons. By default, these polygons *must* be quadrilaterals, although this can be changed if necessary (a recompile is required).

Up to 512 quadrilaterals are supported. Each quadrilateral consists of four coplanar points, each listed on its own line in the polygon file. Each point is a 3-vector in the Body frame with components in meters, separated by commas. The points must be in order, i.e. a line drawn from the first, through the second, through the third, through the fourth, and back to the first must complete the outline of the polygon. For example, the square from Figure 15, starting at the origin, with area 1, lying entirely in the x-y plane, could be defined by:

```
0, 0, 0
0, 1, 0
1, 1, 0
1, 0, 0
```

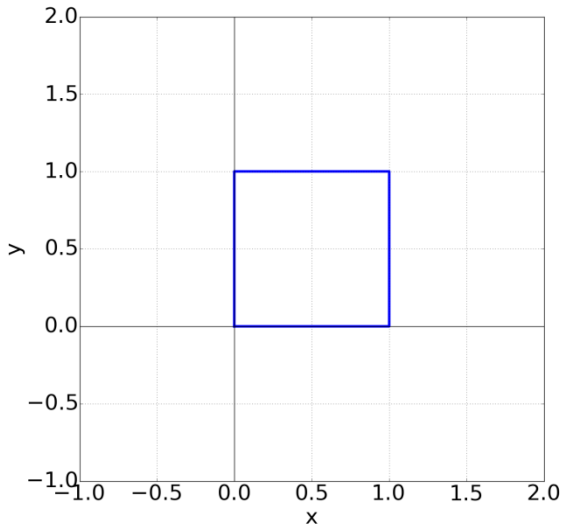

Figure 15. *Example polygon.*

There are other valid representations; what matters is that the points "flow" either clockwise or counter-clockwise such that a line drawn through them in order (and back to the first point again) would outline the polygon.

After listing one polygon, simply list the next after it. Leaving a blank line is optional; they are ignored. List as many polygons as is required to define your satellite. The order of the polygons does not matter. Example geometries are included with KPS.

The KPS_GenPoly tool helps generate these files by allowing the user to customize a Python or MATLAB® script to generate numerical values.

For example, consider a satellite whose geometry includes a solar panel that can deploy at different angles. Say the user wishes to run KPS on several different panel angles. Rather than having to manually compute the numerical values of the solar panel polygon for every deployment angle, the user can enter an expression such as $0.5 + \cos(panel\_angle)$ in the KPS_GenPoly script.

The entire polygon can be constructed this way, in terms of variables chosen freely by the user. These variables can then be set at the top of the script, and the script can then be executed to output a polygon file with the correct numerical values. Changing the panel angle, in this case, would simply require changing the variable at the top and re-running the script, and KPS would be ready to simulate the new angle immediately.

KPS Usage – Polygon Plotter

After producing a polygon file, the user should verify that the satellite has been described correctly. For this reason the author has provided the KPS_PlotPoly utility, which plots a KPS polygon file of the sort generated by KPS_GenPoly in 3-D for a user to inspect. It takes one argument – the name of the polygon file to load. It parses the file and produces an interactive 3-D plot for the user. Figures 16 and 17 contain examples of a dart-style 6U CubeSat viewed from two different orientations, and at two different panel angles.

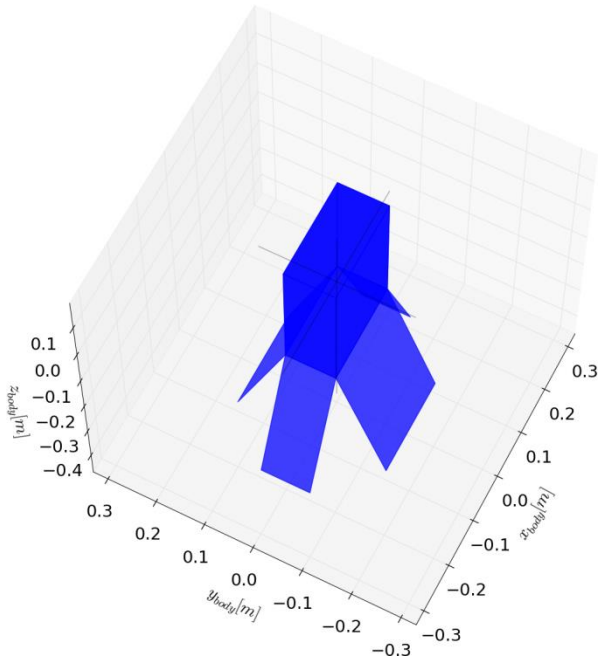

Figure 16. *Dart at* 160° *panel angle*

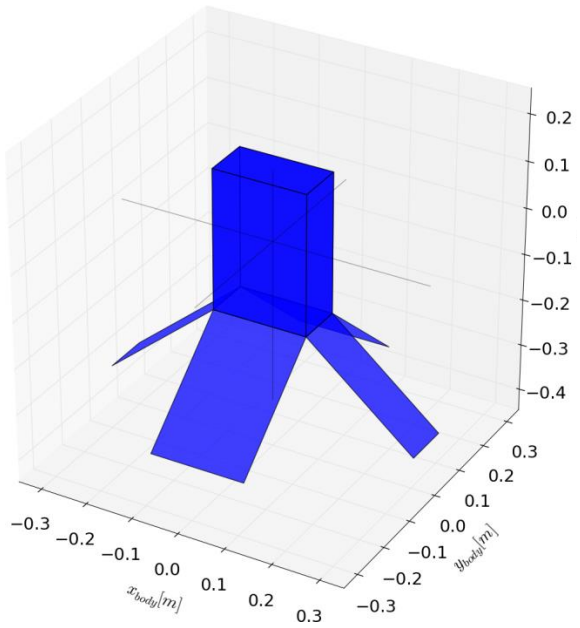

Figure 17. *Dart at* 135° *panel angle*

Verify that all polygons appear as expected, and that none are clipped or crossed by others. If all looks good, the polygon file is ready to be passed to KPS for propagation!

KPS Usage – KPS Orbit Generator

KPS_GenOrbit is the simplest of the utilities. It allows the user to turn a requested orbit, such as "500 x 600 km at 40° inclination", into the `SAT_INIT_R` and `SAT_INIT_V` (initial position and velocity) parameters required by the configuration file.

For an x by y kilometer altitude elliptical orbit at i inclination (in degrees), simply call the utility like:

```
> KPS_GenOrbit <x> <y> <i>
```

The satellite will *begin* at the x portion of the orbit. For example, a 500 x 600 km orbit at 40° inclination in which the satellite begins at 500 km is requested as follows:

```
> KPS_GenOrbit 500 600 40
```

The system responds:

```
Generating stats for a 500x600 km orbit at 40°...
```

```
SAT_INIT_POS = 6871000, 0, 0
SAT_INIT_V = 0, 5855.6619518398, 4913.4837840876
```

That's it! The bottom two lines can be copied and pasted directly into a KPS configuration file.

KPS Usage – Converting Keplerian Elements to State Vectors

KPS_Kepler2State allows the user to perform this conversion. The utility takes the six Keplerian elements as arguments, as follows:

```
> KPS_Kepler2State <a> <e> <w> <Omega> <i> <M>
```

Table 5 elaborates on each argument and its units:

Argument	Name	Units
a	Semi-major Axis	Meters
e	Eccentricity	(Dimensionless)
i	Inclination	Degrees
Ω	RAAN or LAN	Degrees
ω	Argument of Perigee	Degrees
M	Mean Anomaly	Degrees

Table 5. *Arguments to KPS_Kepler2State, in order*

For example, the user might wish to experiment with propagating an existing satellite. The user can obtain the TLEs for that satellite from NORAD online and input the Kelperian elements into KPS_Kepler2State, obtain state vectors, and use the state vectors to start a KPS propagation run.

Example:

```
> KPS_Kepler2State 6871000 0 45 0 0 0
```

The system responds:

```
Position Vector: 6871000, 0, 0 m
Velocity Vector: 0, 5385.7217960362, 5385.7217960362 m/s
```

Kareem Omar
kareem.omar@uah.edu

KPS Usage – Converting State Vectors to Keplerian Elements

KPS_State2Kepler allows the user to perform this conversion. One way to use the utility is to directly specify position and velocity vectors, in that order. The utility returns the six Keplerian elements (and some additional ones). The MATLAB® version directly takes vectors:

```
> KPS_State2Kepler([6871000, 0, 0], [0, 5385, 5385])
```

The Python version takes the same arguments, but split up into six scalars:

```
> KPS_State2Kepler 6871000 0 0 0 5385 5385
```

In either case, the system responds:

```
True Anomaly: 0°
Mean Anomaly: 0°
Eccentric Anomaly: 0°
Semi-major Axis: 6871000.0015514 m
Eccentricity: 2.2579582648063e-10
Argument of Periapsis: 0°
Longitude of Ascending Node: 0°
Inclination: 45°
```

There is however, a second way to call the utility. The user can specify just a single argument – a time, in seconds. KPS_State2Kepler will search through the most recent KPS simulation run and find the first step time at or after that value. The utility will extract the position and velocity at that time and convert it to Keplerian elements.

For example:

```
> KPS_State2Kepler 500
```

The system responds:

```
Time: 500.02489376158 s
Position Vector: 5840925.18459, 2557032.12153, 2556656.26142 m
Velocity Vector: -4013.6839168, 4578.4141753, 4576.2060569 m/s
True Anomaly: 255.25022284075°
Mean Anomaly: -104.66942826045°
Eccentric Anomaly: -104.70960455682°
Semi-major Axis: 6868328.9059404 m
```

```
Eccentricity: 0.00072496937822618 m
Argument of Periapsis: 136.51435783804°
Longitude of Ascending Node: 359.99391990602°
Inclination: 44.988845461891°
```

KPS Usage – Visualization

The final component of KPS is KPS_Vis, the real-time visualization tool. As KPS runs, it outputs 18 satellite parameters. Any combination of these can be plotted simultaneously, either in real-time as the simulation runs, or at any time afterward.

The function takes no arguments, but the user can comment out lines near the top of the program to choose which parameters they wish to plot.

Table 6 explains the available parameters:

Parameter	Symbol	Name
R	r	Position
ORIENTATION	-	View of Satellite Polygons
V	v	Velocity
Q	q	Attitude Quaternion
Q_ORB	q_{orb}	Quaternion to Orbital Frame
ALT	-	Altitude
B_STAR	B^*	Starred Ballistic Coefficient
W	ω	Angular Velocity
V_B	v_b	Velocity (in Body Frame)
E	-	Pointing Error
SEMI_MAJOR	a	Semi-major Axis
ECC	e	Eccentricity
INC	i	Inclination
RAAN	Ω	Longitude of Ascending Node
PERIAPSIS	ω	Argument of Periapsis
MEAN_ANOM	M	Mean Anomaly
TRUE_ANOM	ν	True Anomaly
ECC_ANOM	E	Eccentric Anomaly

Table 6. *Available parameters for plotting with KPS_Vis*

The only novel parameter is the **Pointing Error**. For satellites designed to passively stabilize, this parameter is the angle between the $+z$ axis in the Body frame and the velocity vector in the Body frame. If the satellite stabilizes pointing in the $+z$ direction, this value will diminish to zero.

Although other utilities, such as KPS_State2Kepler or the main propagator's ASCII output mode, provide a means of examining simulation outcomes, the main educational tool in the solution is KPS_Vis.

The ability to visualize a simulation in real-time means users can assess the simulation quality and performance – do options need to be adjusted to increase accuracy? Increase speed? Users can quickly modify a parameter and restart the simulation, encouraging free experimentation with simulation or polygon parameters.

A series of demonstrations follows.

Figure 18 shows the orbital position of a satellite in a 500 x 500 km circular orbit over a 200,000 second simulation (2.3 days), with the `POINT` gravity model. Aerodynamic forces are not sufficient to cause noticeable orbital decay in this time, so the integrator is essentially solving the two-body problem, and a perfect ellipse is produced!

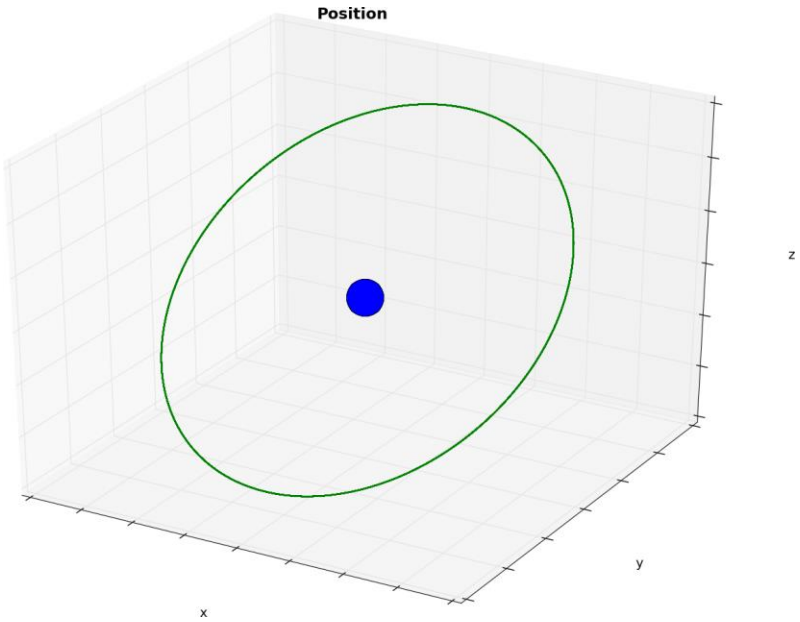

Figure 18. *200,000 seconds of circular orbit, point gravity*.

Figure 19 shows the orbital position for the exact same parameters, except that `POINT` gravity has been replaced by `WGS84`. The ellipsoidal Earth causes perturbations as discussed in the Gravitational Modeling section, and the satellite remains in a stable orbit but experiences drift in its Keplerian elements:

Kareem Omar
kareem.omar@uah.edu

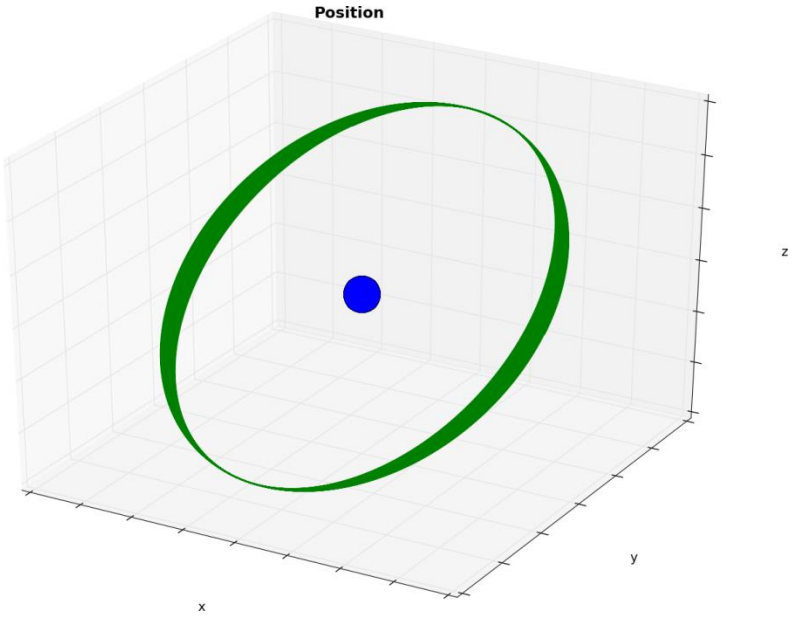

Figure 19. *200,000 seconds of circular orbit, WGS84 gravity.*

Figure 20 shows the small but existent effect of orbital energy loss due to drag. Returning to `POINT` gravity to avoid missing the subtle effect and zooming in on a section of the orbital path, the slight decay of the altitude with each orbit is evident.

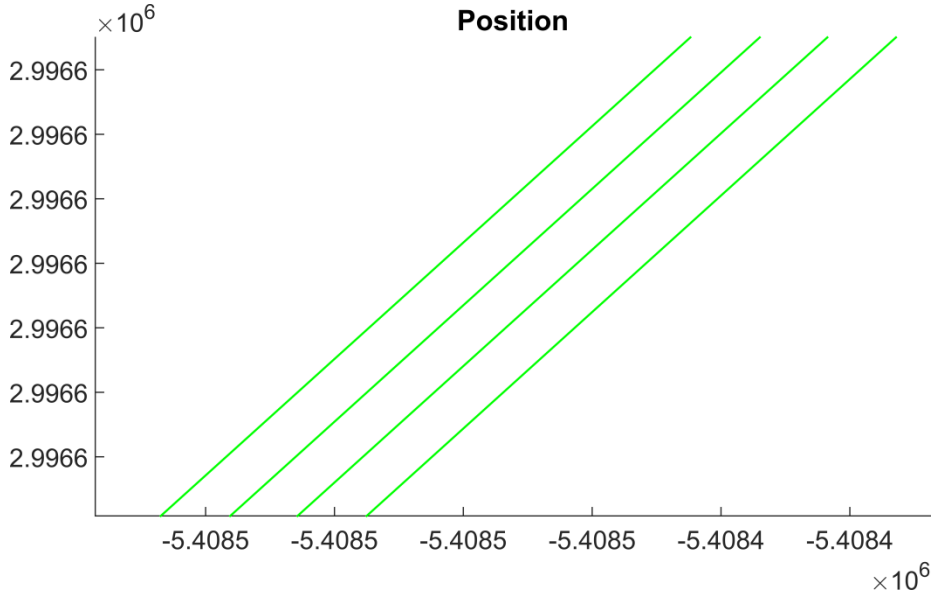

Figure 20. *Detail on first 4 orbits showing orbital decay*

KPS is capable of rapidly simulating significant amounts of mission time, including following a satellite to deorbit in only a few minutes of real time. Figure 21 shows the output of KPS_Vis

Kareem Omar
kareem.omar@uah.edu

configured to plot both altitude and B^* for a satellite in 500 x 600 km elliptical orbit over its entire mission life. At first, the altitude oscillates between 500 and 600 km, as expected. Over time, two effects occur. Firstly, the entire orbit decays as the satellite loses orbital energy due to drag. This begins to occur more and more rapidly as the satellite drops into a denser and denser atmosphere, as seen in the rising B^* term. Secondly, however, the orbit *circularizes*! Notice how much smaller the variation in altitude is at $7x10^7$ seconds than at deploy. Because the force of drag is strongest at *perigee*, it has the effect of applying retrograde thrust at perigee, stealing velocity from the satellite and affecting how far it can reach on the other side of its orbit – which lowers the apogee!

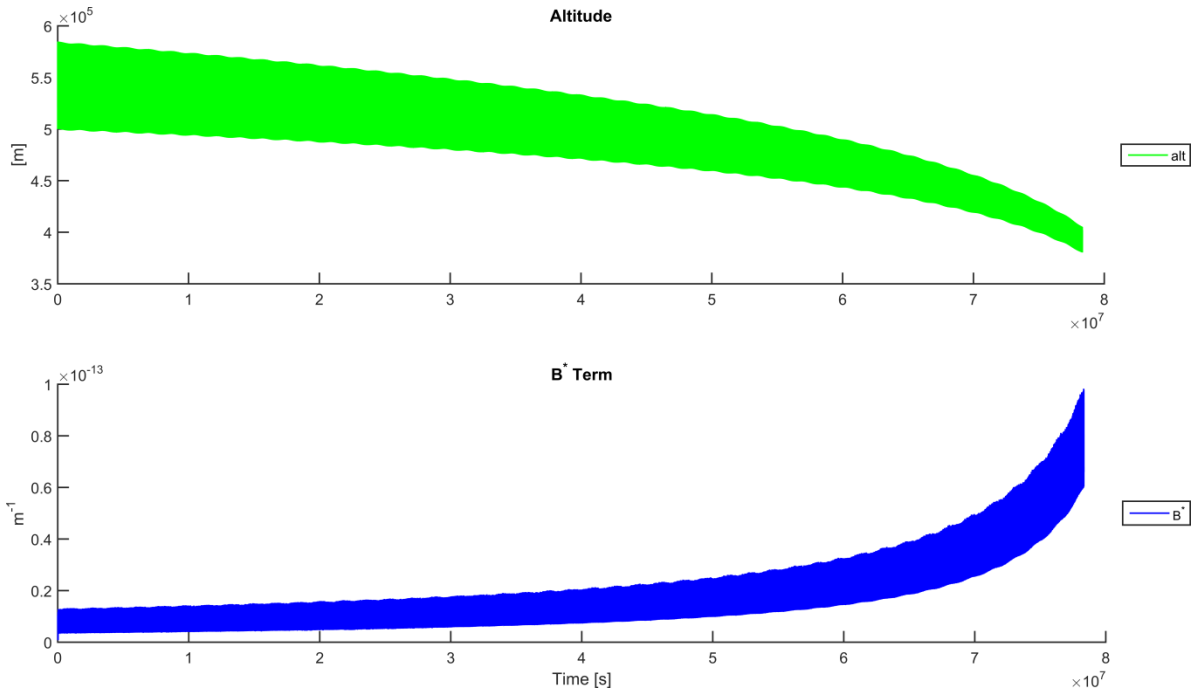

Figure 21. *Orbital decay.*

Figure 22 demonstrates the same mission, but followed all the way to deorbit (< 150 km altitude). The strongly exponential nature of decay becomes obvious. The entire mission – 2.7 *years* of orbit – was simulated by KPS in less than 15 minutes of real time.

Kareem Omar
kareem.omar@uah.edu

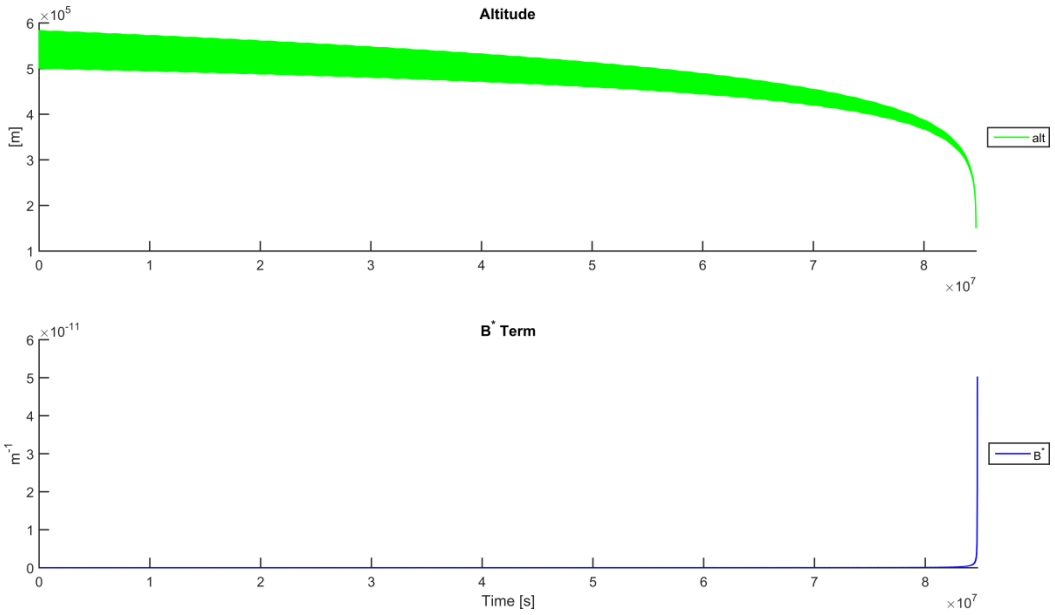

Figure 22. *Orbital decay and deorbit.*

The next series of demonstrations investigate satellite stability by plotting angular velocity, velocity in the body frame, and pointing error. Figure 23 shows a control test – the satellite is a

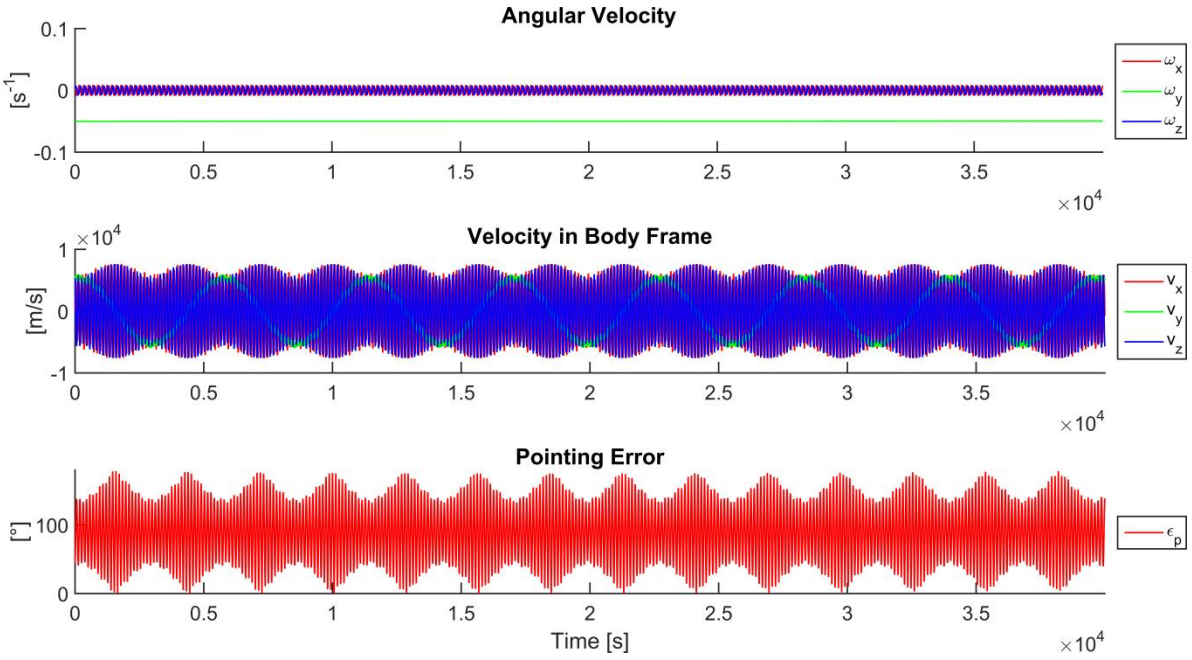

Figure 23. *Control test – rectangular prism with no magnetorque.*

simple, homogeneous rectangular prism. No magnetorque is applied. The satellite deploys with some initial tumble. As expected, these oscillations continue and no stabilization occurs.

Kareem Omar
kareem.omar@uah.edu

Notice that in Figure 23, the angular velocity, even though it is measured in the body frame, *does not remain constant*, despite the lack of net torques. The *z*-component does, but the x and y components oscillate about each other. This is the **coupling of Euler's rotation equations of rigid body dynamics** that occurs in a proper physics model. In KPS this behavior is introduced by the effects of Equation 44.

This coupling is responsible for the strange behavior of spinning objects and does not require KPS or a real satellite to observe. Try spinning something like a phone or a book about each of its three axes in turn. You will find that it spins stably about its minimum and maximum moment of inertia axes, but is unstable if spun about its middle axis! This phenomenon is Euler's equations at work.

Figure 24 retains the same satellite model but adds magnetorque, so there is no stabilizing torque, but there is damping. The angular velocity diminishes! The satellite does not stabilize but occupies a fairly random orientation after detumbling.

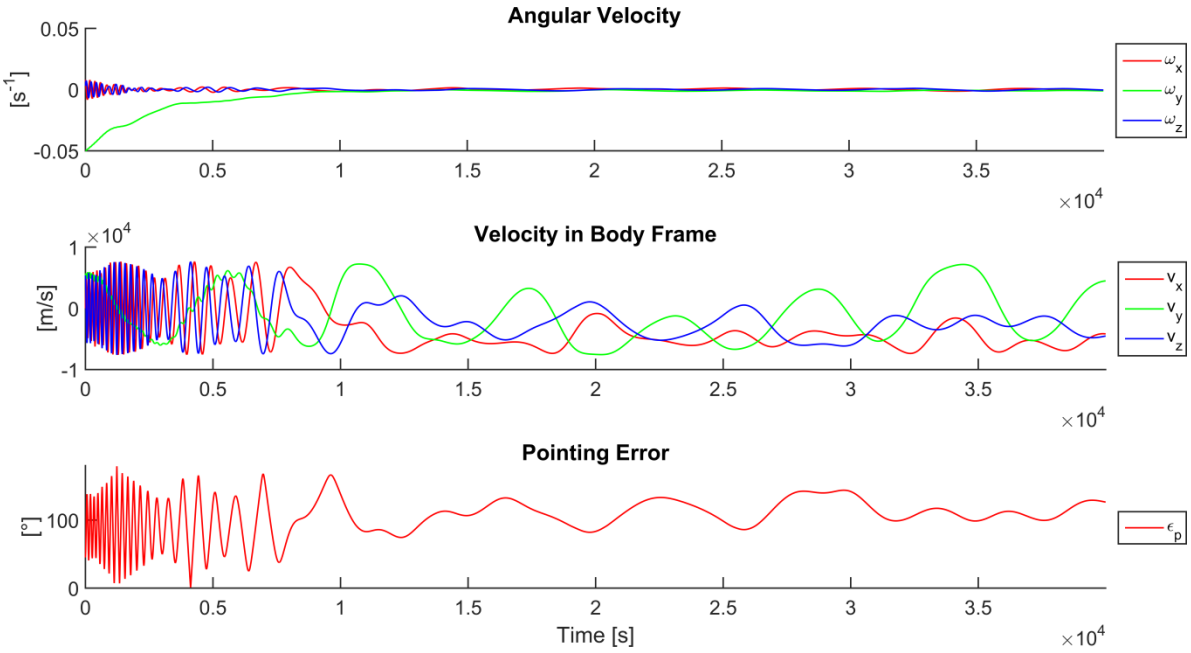

Figure 24. *Damping only test – rectangular prism with magnetorque.*

Figure 25 does the opposite – magnetorque is not used, but the satellite model is changed to the "dart" configuration of Figure 17, so there is a stabilizing torque, but no damping. Interesting – the dart *does not stabilize*. In fact, it looks almost exactly like Figure 23. This is due to the nature of air at extremely low density. As discussed at length in the Aerodynamics section, the extremely high mean free path means that air acts like bullets, imparting momentum, not like a

flow. The air can still generate restoring torques, but it does *not* provide the damping that, say, an arrow fired at sea level would have due to skin friction and several other flow phenomena.

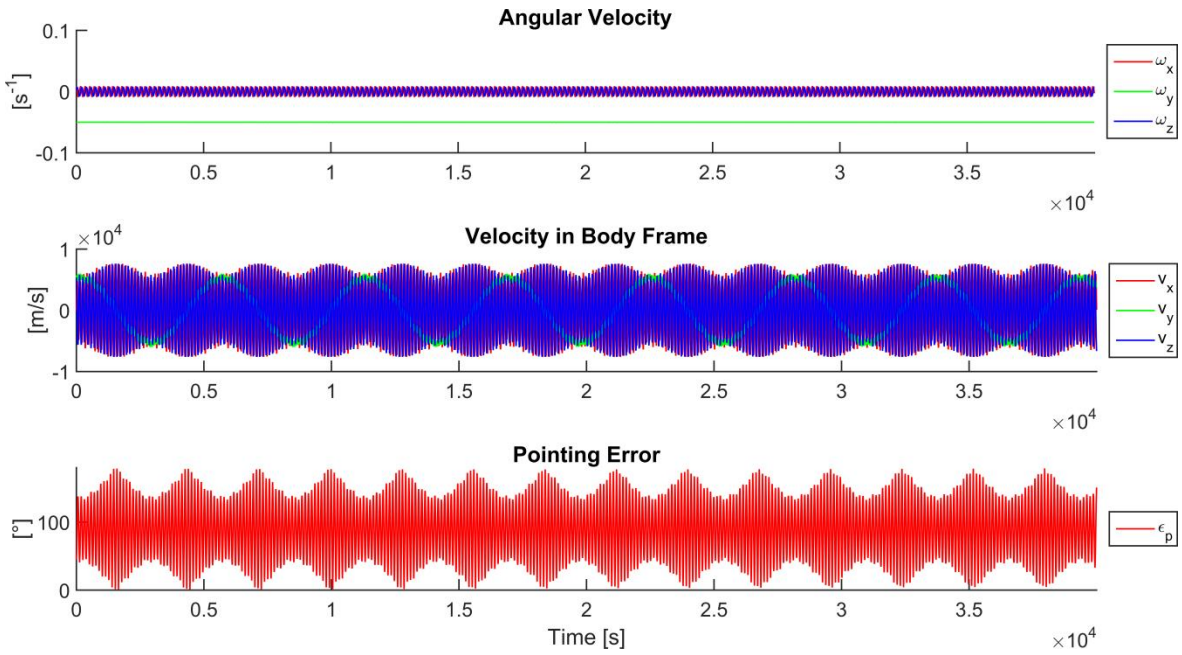

Figure 25. *Torque only test – dart with no magnetorque.*

The dart configuration must be used in *conjunction* with magnetorque to produce prograde

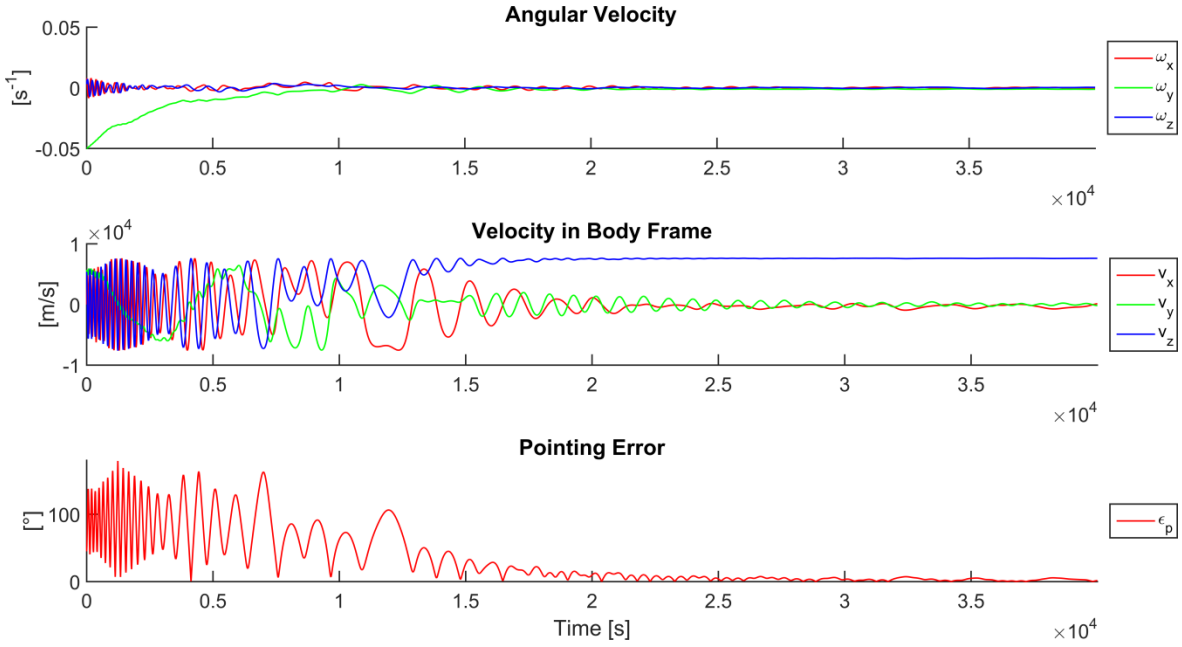

Figure 26. *Torque and damping – dart with magnetorque.*

stabilization. Figure 26 adds magnetorque to the dart model.

Kareem Omar
kareem.omar@uah.edu

With the combination of torque and damping, the satellite stabilizes with its long axis pointing in the direction of travel, just like an arrow on Earth! Note that this process is an *active* one. If all torques on the satellite were to cease, the satellite would *not* maintain pointing, because as it travels around the Earth, it must constantly turn toward the Earth to follow its shifting velocity vector, as illustrated in Figure 27.

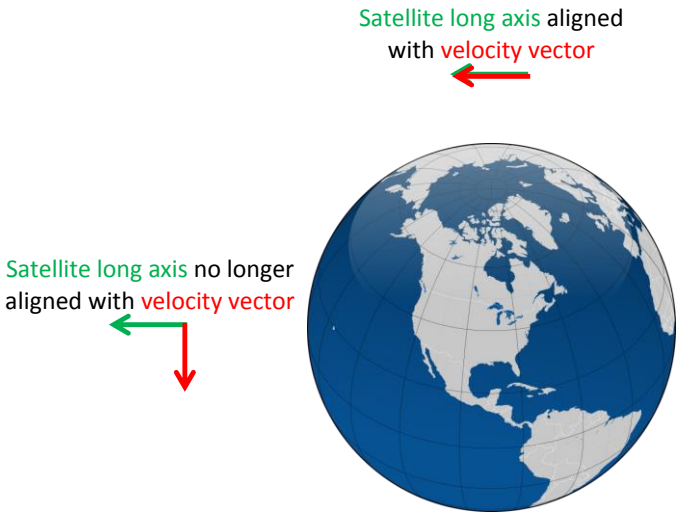

Figure 27. *A satellite must rotate once per orbit to remain in stable pointing.*

The aerodynamic torques driving this stabilization are proportional to density, so at higher altitudes, **residual pointing error** tends to occur, and at some altitude (for a particular satellite configuration), stabilizing shapes like the dart will fail to stabilize altogether and tumble. KPS can be used to assess what combinations of parameters are likely to produce stable pointing. Keep in mind that density fluctuates enormously, so marginally stable designs may still tumble during portions of their orbits.

KPS Internals – Libraries

KPS is designed for ease of use and comprehensibility. The author has endeavored to minimize external dependencies required to run KPS. Nevertheless, four excellent libraries have been employed for some tasks to guarantee correctness and help solve difficult problems with high performance. The author deliberately selected only libraries that are **header-only** or support **static compilation** so that the end user does *not* have to install the libraries to run KPS; they are embedded in the binary itself. Users who wish to compile KPS do, of course, need the libraries.

The first of these excellent tools is the **OpenGL Mathematics (GLM)** header-only library by G-Truc Creation, available at http://glm.g-truc.net/0.9.7/index.html. GLM is intended for high-speed GLSL-style graphics math, so it has built-in vector, matrix, and quaternion handling with convenience functions for operations such as cross products, dot products, and quaternion rotations, all with an emphasis on performance. Even more importantly, it provides CUDA versions of most functions, greatly reducing CUDA development effort and helping the CUDA version of the KPS aerodynamics module more closely resemble the CPU version.

The second library employed by KPS is the outstanding **Eigen** linear algebra template library, offering extremely high-performance and reliable arbitrary vector and matrix math. Eigen is available at http://eigen.tuxfamily.org/. Whereas GLM excels at 3-vectors and quaternions and is thus ideally suited to individual operations on components of the satellite state, the numerical propagators consider an entire vector in one go, as described in Figures 10 and 11. A full description of the operation of the author's numerical propagators is beyond the scope of this paper, but in brief, they have been designed in a modular fashion, independent of their particular application in KPS. The propagators are templated for performance and can integrate vectors of arbitrary length. Their internals require arbitrary, sometimes dynamic, matrices and vectors. Eigen provides these capabilities.

The third library employed by KPS is the fast and elegant **Clipper** polygon clipping tool, available at http://www.angusj.com/delphi/clipper.php. Clipper uses a modified and extended version of **Vatti's Clipping Algorithm** to perform 2-D polygon differencing and intersection operations. This functionality assists KPS in computing the precise vertices of the frontal area projections of unoccluded polygons for the Analytical aerodynamics mode.

The final library has already been mentioned; it's the invaluable GeographicLib, available at http://geographiclib.sourceforge.net/. In KPS it is used for gravitational modeling, magnetic modeling, and coordinate system conversions. It has widespread use and support in the scientific community and offers rigorous test suites to verify correctness and stability.

KPS Internals – Aerodynamics

The heart of KPS is the aerodynamics simulation. The basics were discussed in the Aerodynamics background section, but the computational implementation, particularly for the collision mode, will be explored here.

On initialization, the polygon file specified by the user is parsed into a flat array of vertices (points), with each group of four vertices describing a polygon. This set of polygons constitutes the geometry of the satellite in the Body frame.

There are many ways to attack this problem. For KPS the two approaches provided were chosen to jointly offer suitable compromises between performance, simplifying assumptions, accuracy, suitability for CUDA, and ease of explanation.

When the aerodynamics module is called, it receives the density and the satellite velocity vector in the Body frame. It must simply return the aerodynamic force and torque based on those inputs.

The analytical model operates as described in the Aerodynamics background section.

Kareem Omar
kareem.omar@uah.edu

The collision model operates in three steps: **rotate, precompute,** and **collide**.

The satellite polygons are first rotated such that the velocity points in the $+x$ direction, i.e. such that the relative wind is arriving *from* the $+x$ direction. Unit normal for each polygon are then precomputed to increase performance, as are some intermediate quantities that will be useful later. The **rectangular extents** of each polygon, as well as of the entire satellite, are precomputed, meaning the maximum and minimum x and y coordinates.

The system is now ready to simulate collisions from test particles. A rectangular grid of test particles is generated based on the aforementioned satellite extents – any test particles generated outside the extents would miss the satellite – with the linear pitch (spacing between particles) specified by the user in the configuration file.

Each of these particles must be tested to see whether it collides with the satellite, and, if so, where.

Algorithm design is essentially the task of restructuring human problems in a way that can be solved by circuitry, and collision testing is a prime example. This is where the cost of rotating all the polygons at the outset pays for itself, as it means the test particles travel purely in the $-x$ direction until they hit the satellite (or miss). See Figure 28. The problem of testing a particle, then, can be restructured as follows: *for a given test particle, what is the first face hit when traveling in the $-x$ direction?* This can be further restated: *for a given y and z, what is the largest x-value that resides in a face?*

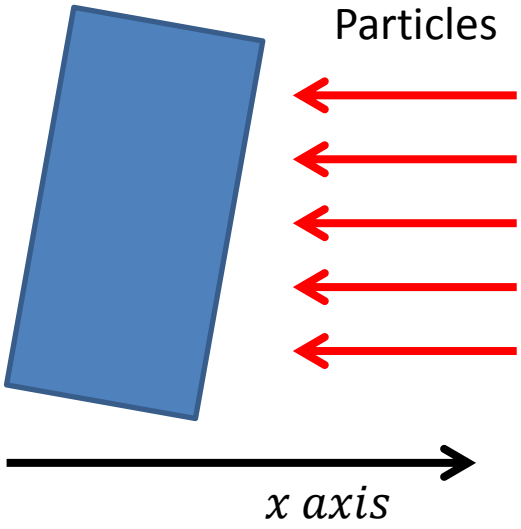

Figure 28. *Test particles travel in the –x direction to collide with the rotated satellite geometry –in this case, a rectangular prism.*

The system can test each polygon by checking whether the y and z of the current test particle reside within the bounds of the polygon. If they do, the particle would collide with that polygon if it had no other polygons in the way. The x-coordinate of the collision is computed.

Keeping track of the polygon with the highest x-value allows the system to cull occluded polygons. For example, in Figure 28, all five particles shown could hit both the front and rear faces of the rectangular prism. However, because the x-value of the front face's collision location

for each particle is greater, only *that* collision is kept. The particle does not make it to the rear face.

It only remains to clarify how exactly "checking whether the y and z of the current test particle reside within the bounds of the polygon" is accomplished. This is a classic problem in simulation and computer graphics known as the **point-in-polygon** problem. There are various solutions, some faster than others, with various optimizations possible for certain cases. This is the *hottest code path* in KPS, meaning it is executed more frequently than any other path by far, because it must run for every test particle for every polygon for every aerodynamics simulation call. Thus, optimization is critical.

Consider Figure 29. The goal is to determine whether a point is in a polygon. A human can simply look at Figure 29 and say that points P and Q are inside, but points R and S are outside. How can the problem be expressed suitably for a computer? Notice that, starting from a test point inside the polygon, an *odd* number of edges must be crossed to escape in any direction. Starting from a test point outside the polygon, an *even* number of edges will be crossed if traveling far enough in any direction. (Try moving to the right, say, from each of the test points.) This works even for concave polygons like the one in Figure 29.

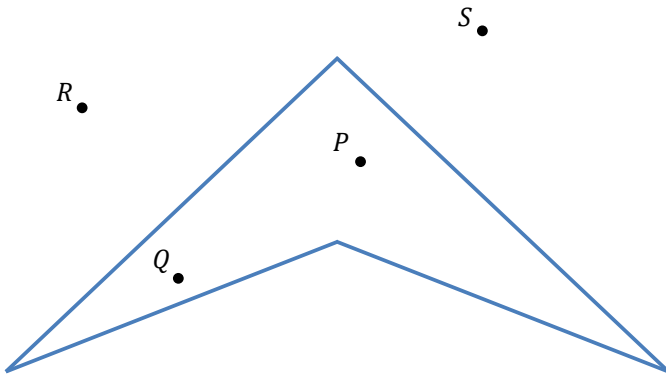

Figure 29. *The point-in-polygon problem.*

Thus, finding the number of edges between the point and $+\infty$ in a particular direction is an *equivalent problem*. Now, translate the entire system so that the test point is at the origin, and arbitrarily select $+x$ as the direction it will travel. See Figure 30. Notice that the path the "escaping" point takes is simply the entirety of the $+x$ axis. Thus, restructuring the problem: *test*

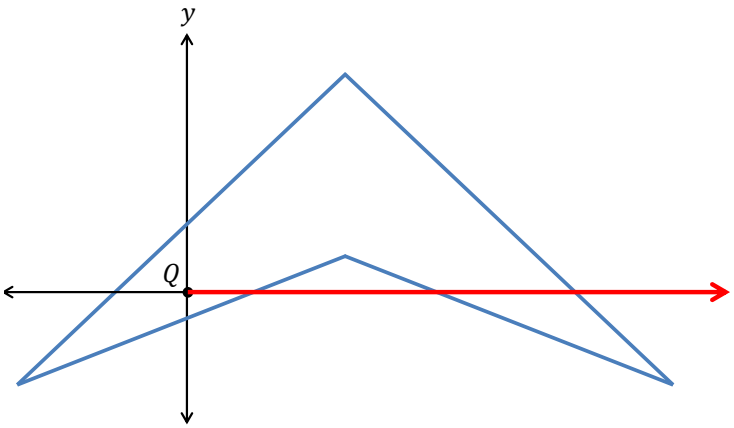

Figure 30. *The point-in-polygon problem.*

every edge of the polygon to see if it crosses the +x axis. If the total number of crossings is even, the point is outside. If it's odd, the point is inside.

This formulation is suitable for a computer. Checking whether an edge crosses the +x axis is computationally efficient, as well:

- If both y-coordinates share the same sign (i.e. the edge resides wholly above the axis, or wholly below it), the edge does NOT cross. Move on to the next edge.
- If both x-coordinates are negative (i.e. the edge resides wholly to the left of the y-axis), the edge does NOT cross. Move on to the next edge.

These fast checks eliminate most edges immediately. Any remaining candidate edges do cross the x-axis; all that is required is to solve for the crossing x-coordinate to determine whether it's positive or negative. If it's positive, the edge crosses.

KPS Internals – CUDA Aerodynamics

The author has made every effort to minimize differences between the CPU and CUDA versions of the collision-mode aerodynamics module. The collision problem involves several steps in which identical operations are performed repeatedly on different data, making it suitable for parallel processing.

Functions written to execute on a CUDA device are called **kernels**. The instructions of one kernel, when called, execute in a **thread** of execution. Multiple threads can run simultaneously, however, allowing each one to process a different portion of the data. For example, a kernel which simply doubles a value could be **launched** with ten threads to simultaneously double ten values together. Note that this number cannot scale indefinitely; threads execute in batches of 32, known as **warps**. Furthermore, there is a limitation (which varies by GPU) on the number of threads that can execute per **block**, allowing those threads access to **shared memory**. Nevertheless, the speedup for compute-bound parallel problems is impressive.

Each thread consists of the same instructions, but unique **block IDs** and **thread IDs** are provided to each one, allowing each thread to know which data to operate on at runtime.

The CPU aerodynamics module simply iterates over all the polygons to rotate and precompute. The CUDA version launches a kernel with one thread for each polygon.

Similarly, the CPU aerodynamics module iterates over individual test particles to collide them. The CUDA version launches a kernel with one thread for each *particle*.

However, note that the precompute and collide steps are not purely parallel. The precompute thread finds the extents for its polygon, but finding the extents of the whole *satellite* requires checking each of the threads' local results to identify global minima and maxima.

Similarly, each particle collision contributes force and torque to a running total. Thus a summation is required across all of the threads.

Finding the minimum or maximum, as in the precompute kernel, and computing a summation, as in the collide kernel, are operations known as **reductions**. Such an operation, which considers every element and produces a single final answer, but in which the order of access does not matter, can be performed in **logarithmic time** in CUDA. Consider Figure 31. If the goal is to sum the eight values, the following procedure may be used:

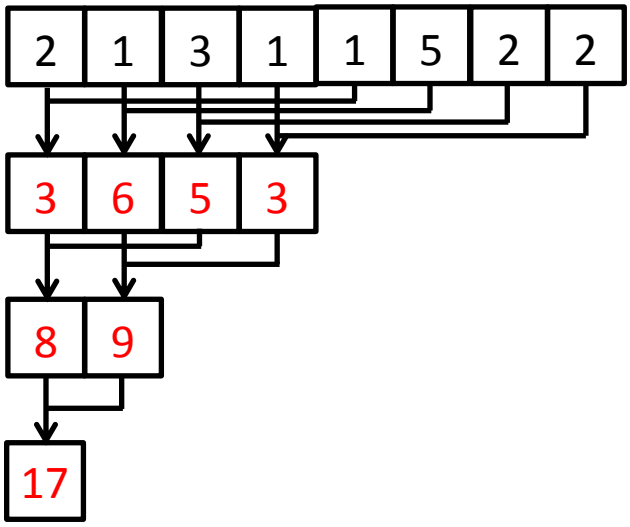

Figure 31. *CUDA reduction.*

- Split the data into two halves, "top" and "bottom"
- All "top" threads terminate; they are not needed
- Each "bottom" thread adds the corresponding value from the top half to itself
- Repeat on the "bottom" data

In the example from Figure 31, the eight values are summed in just three instructions – logarithmic time! The efficiency of reductions in CUDA further contributes to its performance, and the use of two different reductions in KPS provides opportunity for study.

Conclusion

KPS allows users to simulate and visualize satellite trajectories and orientations, all through a completely free and open source framework. Real-time plotting allows users to immediately see the effects of changes in initial conditions or the configuration of the propagator, and six ancillary utilities allow users to experiment with orbits and satellite geometry.

In addition, the software itself has been carefully designed as a modular system for inspection and modification for interested programmers, and a unifying background has been laid in this

paper. In this way the author hopes to generate interest in using CUDA and high-performance computing as a teaching tool for aerospace visualization and education.

Kareem Omar
kareem.omar@uah.edu

References

1. NVIDIA Corporation. *CUDA*. n.d. 29 1 2016. <http://www.nvidia.com/object/cuda_home_new.html>.

2. —. *What's the Difference Between a CPU and a GPU?* n.d. 29 1 2016. <http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/>.

3. —. *CUDA GPUs*. n.d. 29 1 2016. <https://developer.nvidia.com/cuda-gpus>.

4. —. *CUDA Downloads*. n.d. 29 1 2016. <https://developer.nvidia.com/cuda-downloads>.

5. Analytical Graphics, Inc. *Astronautics Primer*. n.d. 1 2 2016. <http://www.agi.com/resources/educational-alliance-program/astro-primer/primer1.htm>.

6. Fjeld, Eric M. Jones and Paul. *Gimbal Angles, Gimbal Lock, and a Fourth Gimbal for Christmas*. 2000. 1 2 2016. <https://www.hq.nasa.gov/alsj/gimbals.html>.

7. Nave, R. *Newton's 2nd Law: Rotation*. n.d. 1 2 2016. <http://hyperphysics.phy-astr.gsu.edu/hbase/n2r.html>.

8. GeographicLib. *Gravity models*. n.d. 1 2 2016. <http://geographiclib.sourceforge.net/html/gravity.html>.

9. —. *Magnetic models*. n.d. 1 2 2016. <http://geographiclib.sourceforge.net/html/magnetic.html>.

10. Nave, R. *Magnetic Dipole Moment*. n.d. 1 2 2016. <http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/magmom.html>.

11. NASA. *U.S. Standard Atmosphere 1976*. n.d. 1 2 2016. <http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770009539.pdf>.

12. Curtis, Howard. *Orbital Mechanics for Engineering Students*. Butterworth-Heinemann, 2013.