Autonomous Vehicle Simulation (AVS) Laboratory, University of Colorado

Basilisk Technical Memorandum

Document ID: Basilisk-avsLibrary

AVS LIBRARY OF ASTRODYNAMICS AND C-BASED LINEAR ALGEBRA SUPPORT SUB-ROUTINES

Prepared by	H. Schaub

Status: Initial Document
Scope/Contents
The AVS Library has released a series of C, Matlab and Mathematica support routines to perform common astrodynamics functions. This document outlines how these support files are automatically validated within the Basilisk environment. There are C-based functions that run each routine with sample input, and compare to hand computed answers if the function response is correct. This checking is performed automatically every time `pytest` is run on the root Basilisk folder.

Rev	Change Description	By	Date
1.0	Initial Revision	H. Schaub	08/2017
1.1	Small typo fixes	H. Schaub	10/18/2017

Contents

1 Library Description

The AVS lab has released a set of astrodynamics C-functions* that facilitate common orbital or rotational dynamics evaluations. The functions are openly available, and are included with the AIAA Education Series textbook *Analytical Mechanics of Space Systems*[1] since the first release in 2003. The most up to date public versions are found on the AVS web page.

1.1 `linearAlgebra` Library

The linear algebra library provides numerous C-based functions to perform basic matrix math operations. For a complete list of functions supported, consult the `linearAlgebra.h` file. The library provides common matrix and vector manipulation tools such as performing a matrix product, including the transpose operators, as well as doing the matrix inverse operations. The vector library supports doing dot and cross product operations, as well as taking a norm of a vector. Helper functions are provided to initial zero or identity matrices, as well as check if the matrix elements are zero

1.2 `rigidBodyKinematics` Library

The rigid body kinematics library provides a range of functions related to mapping between attitude coordinates descriptions, compute the addition or subtraction of orientations, as well as returning the differential kinematic equations. This library has been included with Reference 1 since 2003 and has been used extensively across a range of research projects. Reference 1 provides all the mathematical details of the transformation used here in Chapter 3, while Appendix E provides a comprehensive discussion of all the algorithms included.

1.3 `orbitalMotion` Library

Common celestial mechanics tools are implemented in the `orbitalMotion` library, including transforming between anomaly angles, converting between inertial Cartesian position and velocity vector components and classical orbit elements, as well as evaluating simple space environment parameters. The following developments use the notation and formulation of Chapter 9 in Reference 1. The variable naming is reflected within the software code.

1.3.1 Classical Orbital Elements

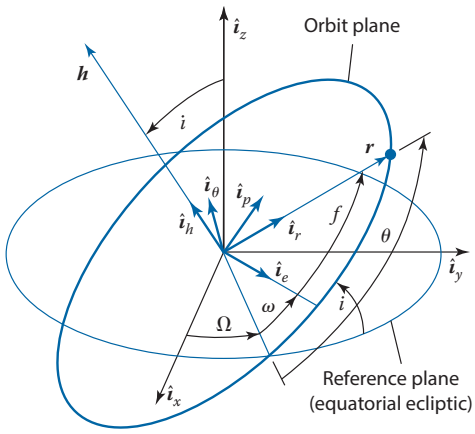
The classcial orbit elements are given by

$$(a, e, i, \Omega, \omega, f)$$

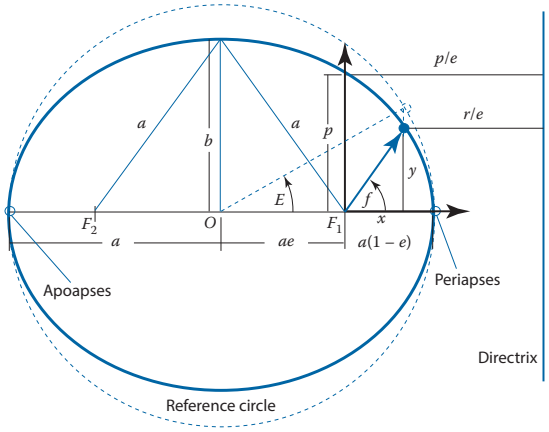
where a is the semi-major axis or SMA, e is the eccentricity, (Ω, i, ω) are the 3-1-3 Euler angle orbit plane orientation angles called the ascending node Ω, the inclination angle i and the argument of periapses ω which are illustrated in Figure 1(a). The semi-latus rectum p is defined as

$$p = a(1 - e^2) \tag{1}$$

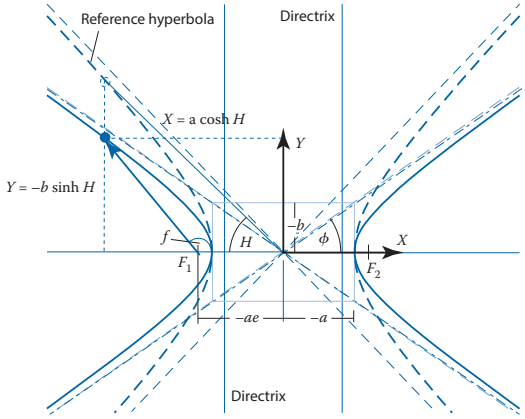
* `http://hanspeterschaub.info/AVS-Code.html`



(a) Orbit Frame Orientation Illustration



(b) Classical orbital parameter Illustration for an Elliptic Orbit



(c) Classical orbital parameter Illustration for a Hyperbolic Orbit

Fig. 1: Orbit Elements, Axes and Frames Illustrations.[1]

Finally, the true anomaly angle is given by f, while the eccentric anomaly angle is expressed through E as illustrated in Figure 1(b) for an elliptic orbit scenario. The true and eccentric anomaly are related through[1]

$$\tan\left(\frac{f}{2}\right) = \sqrt{\frac{1+e}{1-e}}\tan\left(\frac{E}{2}\right) \tag{2}$$

The mean anomaly angle M relates to the eccentric anomaly angle E through[1]

$$M = E - e\sin E \tag{3}$$

The classical orbit elements for a hyperbolic scenario are shown in Figure 1(c). Note that the convention is used where the SMA is a negative value for a hyperbolic orbit, and thus $a < 0$ in this case. The true anomaly angle f definition is universal for all orbit types, while the hyperbolic anomaly H relates to f through

$$\tanh\left(\frac{f}{2}\right) = \sqrt{\frac{e+1}{e-1}}\tanh\left(\frac{H}{2}\right) \tag{4}$$

The hyperbolic mean anomaly is defined in terms of the hyperbolic anomaly through

$$N = e\sinh H - H \tag{5}$$

While mapping from eccentric or hyperbolic anomalies to mean anomalies is analytical, the inverse is not. The sub-routines use Newton's method to numerically solve from mean anomalies to the corresponding eccentric or hyperbolic anomalies. This is called solving Kepler's equation. The iterations continue until a change tolerance of 10^{-13} radians is achieved, or a maximum of 200 iterations are performed. Note that solving this Kepler's equation for most cases converges very quickly within 3-5 iterations.

1.3.2 Orbit Element to Cartesian Coordinate Conversions

Let ${}^{\mathcal{N}}\boldsymbol{r}_{C/N}$ and ${}^{\mathcal{N}}\boldsymbol{v}_{C/N}$ be the inertial spacecraft center of mass C position and velocity vector matrix representations, expressed with respect to inertial frame $\mathcal{N}:\{\hat{\boldsymbol{n}}_1,\hat{\boldsymbol{n}}_2,\hat{\boldsymbol{n}}_3\}$ components. Functions are provided to map from classical orbit elements to these Cartesian coordinates through `elem2rv()`, as well as the inverse mapping from Cartesian to classical orbit elements through `rv2elem()`.

1.3.2.1 `elem2rv()` **Function** The general conversion of classical orbit elements to the inertial Cartesian components is outlined in detail in section 9.6.2 of Reference 1. Given the true anomaly angle f, the orbit radius is for any orbit type given by

$$r = \frac{p}{1 + e\cos f} \tag{6}$$

The true latitude angle θ is given by

$$\theta = \omega + f \tag{7}$$

The inertial position vector is then given by[1]

$$\boldsymbol{r}_{C/N} = r \, {}^{\mathcal{N}}\begin{pmatrix} \cos\Omega\cos\theta - \sin\Omega\sin\theta\cos i \\ \sin\Omega\cos\theta + \cos\Omega\sin\theta\cos i \\ \sin\theta\sin i \end{pmatrix} \tag{8}$$

while the inertial velocity vector is

$$\dot{\boldsymbol{r}}_{C/N} = -\frac{\mu}{h} \, {}^{\mathcal{N}}\begin{pmatrix} \cos\Omega(\sin\theta + e\sin\omega) + \sin\Omega(\cos\theta + e\cos\omega)\cos i \\ \sin\Omega(\sin\theta + e\sin\omega) - \cos\Omega(\cos\theta + e\cos\omega)\cos i \\ -(\cos\theta + e\cos\omega)\sin i \end{pmatrix} \tag{9}$$

where μ is the gravitational constant and h is the massless orbital angular momentum $\boldsymbol{h} = \boldsymbol{r} \times \dot{\boldsymbol{r}}$.

The `elem2rv()` function checks for a special rectilinear motion case. Here the eccentricity is $e = 1$ while the SMA is positive with $a > 0$. Under this condition the spacecraft is moving purely along a radial direction relative to the planet, and the true anomaly angle no longer can be used to determine the spacecrafts location. In this scenario the Eccentric anomaly angle E can still be used to determine spacecraft position, and the `elem2rv()` function assumes here that the anomaly angle provided is E and not f. The orbit radius is here computed using[1]

$$r = a(1 - e\cos E) \tag{10}$$

The radial unit direction vector along which all rectilinear motion is occuring is

$${}^{\mathcal{N}}\hat{\boldsymbol{i}}_r = {}^{\mathcal{N}}\begin{pmatrix} \cos\Omega\cos\omega - \sin\Omega\sin\omega\cos i \\ \sin\Omega\cos\omega + \cos\Omega\sin\omega\cos i \\ \sin\theta\sin i \end{pmatrix} \tag{11}$$

The inertial position vector is then given by

$${}^{\mathcal{N}}\boldsymbol{r}_{C/N} = r \, {}^{\mathcal{N}}\hat{\boldsymbol{i}}_r \tag{12}$$

The velocity magnitude v is determined through the orbit energy equation as[1]

$$v = \sqrt{\frac{2\mu}{r} - \frac{\mu}{a}} \tag{13}$$

The inertial velocity vector is a function of the eccentric anomaly E through

$$^{\mathcal{N}}\boldsymbol{v}_{C/N} = \begin{cases} -v\ ^{\mathcal{N}}\hat{\boldsymbol{i}}_r & 0 \leqslant E \leqslant \pi \\ +v\ ^{\mathcal{N}}\hat{\boldsymbol{i}}_r & -\pi \leqslant E \leqslant 0 \end{cases} \tag{14}$$

1.3.2.2 `rv2elem()` **Function** To convert from the inertial position and velocity vectors to the corresponding classical orbit elements, the following algorithm is used. First the semi-latus rectum is computed using

$$p = \frac{\boldsymbol{h} \cdot \boldsymbol{h}}{\mu} \tag{15}$$

The line of nodes vector $\boldsymbol{n}$ is

$$\boldsymbol{n} = \hat{\boldsymbol{n}}_3 \times \boldsymbol{h} \tag{16}$$

The eccentricity vector $\boldsymbol{e}$ is computed using[1]

$$\boldsymbol{e} = \frac{\dot{\boldsymbol{r}} \times \boldsymbol{h}}{\mu} - \frac{\boldsymbol{r}}{r} \tag{17}$$

The eccentricity is then simply

$$e = |\boldsymbol{e}| \tag{18}$$

Within this function the orbit radius is set through

$$r = \frac{\boldsymbol{r}_{C/N}}{|\boldsymbol{r}_{C/N}|} \tag{19}$$

and the radius at periapses is set to

$$r_p = \frac{p}{1+e} \tag{20}$$

The SMA is computed using a robust method where first the inverse of the SMA, called α, is evaluated using[1]

$$\alpha = \frac{2}{r} - \frac{v^2}{\mu} \tag{21}$$

If α is non-zero, then the orbit is not parabolic and a is determined through

$$a = \frac{1}{\alpha} \tag{22}$$

while the radius at apoapses is set to

$$r_a = \frac{p}{1-e} \tag{23}$$

If α is zero, then the motion is parabolic and the SMA is not defined (i.e. is infinite). In this case the code sets

$$a = -r_p$$

while $r_a = -1.0$ is returned as a value that is not defined in this case.

Next the orbit frame orientation angles Ω, i and ω must be determined. As classical elements are used, care must be given for particular singular orientations where some angles are ill-defined. First,

assume a non-circular non-equatorial orbit scenario. In this case Ω is the angle between $\hat{\boldsymbol{n}}_1$ and $\boldsymbol{n}$ where care must be taken that the right quadrant is used. The argument of periapses is the angle between $\hat{\boldsymbol{n}}$ and $\boldsymbol{e}$, where again quadrants must be checked. Finally, the inclination angle i is the angle between $\boldsymbol{h}$ and $\hat{\boldsymbol{n}}_3$, but here no quadrants must be checked as this angle is defined as $0 \leqslant i \leqslant \pi$. To find the true anomaly angle, the angle between $\boldsymbol{r}$ and $\boldsymbol{e}$ is determine while checking for quadrants again.

The 3-1-3 Euler angles representing the orbit frame orientation have singular configurations. The following list discusses how each case is handled:

- **Equatorial non-circular orbit:** Here the ascending node Ω is ill-defined, and is set to 0.0 radians.
- **Inclined circular orbit:** Here the argument of periapses ω is ill-defined, and is set to 0.0 radians.
- **Equatorial circular orbit:** Here both Ω and ω are ill-defined are are each set to 0.0 radians.

1.3.3 Space Environment Parameters

1.3.3.1 Atmospheric Density This program computes the atmospheric density based on altitude supplied by user. This function uses a 7-th order polynomial curve fit based on atmospheric data from the Standard Atmosphere 1976 Data. This function is valid for altitudes ranging from 100km to 1000km.

1.3.3.2 Mean Debye Length This program computes the Debye Length length for a given altitude and is valid for altitudes ranging from 200 km to GEO (35000km). However, all values above 1000 km are HIGHLY speculative at this point.

1.3.3.3 Atmospheric Drag Acceleration This program computes the atmospheric drag acceleration vector ${}^{\mathcal{N}}\boldsymbol{a}_d$ acting on a spacecraft. Note the acceleration vector output is inertial, and is only valid for altitudes up to 1000 km. Afterwards the drag force is zero. Only valid for Earth. The disturbance acceleration is defined as

$$\boldsymbol{a}_d = -\frac{\rho}{2} C_d \frac{Av^2}{m} \cdot \frac{\dot{\boldsymbol{r}}}{|\boldsymbol{r}|} \tag{24}$$

where ρ is the local atmospheric density, C_d is the drag coefficient, A is the projected area into the flight direction, and m is the spacecraft mass.

1.3.3.4 Gravitational Zonal Harmonics This function computes Earth's zonal harmonics from J_2 through J_6. For other planets only their J_2 value is used, while the higher order harmonics are set to zero. The gravitational zonal harmonic accelerations are then given by[1]

$$\boldsymbol{a}_{J_2} = -\frac{3}{2} J_2 \left(\frac{\mu}{r^2}\right)\left(\frac{r_{eq}}{r}\right)^2 {}^{\mathcal{N}}\begin{pmatrix} \left(1 - 5\left(\frac{z}{r}\right)^2\right)\frac{x}{r} \\ \left(1 - 5\left(\frac{z}{r}\right)^2\right)\frac{y}{r} \\ \left(3 - 5\left(\frac{z}{r}\right)^2\right)\frac{z}{r} \end{pmatrix} \tag{25}$$

$$\boldsymbol{a}_{J_3} = -\frac{1}{2} J_3 \left(\frac{\mu}{r^2}\right)\left(\frac{r_{eq}}{r}\right)^3 {}^{\mathcal{N}}\begin{pmatrix} 5\left(7\left(\frac{z}{r}\right)^3 - 3\left(\frac{z}{r}\right)\right)\frac{x}{r} \\ 5\left(7\left(\frac{z}{r}\right)^3 - 3\left(\frac{z}{r}\right)\right)\frac{y}{r} \\ 3\left(10\left(\frac{z}{r}\right)^2 - \frac{35}{3}\left(\frac{z}{r}\right)^4 - 1\right) \end{pmatrix} \tag{26}$$

$$\boldsymbol{a}_{J_4} = -\frac{5}{8} J_4 \left(\frac{\mu}{r^2}\right)\left(\frac{r_{eq}}{r}\right)^4 {}^{\mathcal{N}}\begin{pmatrix} \left(3 - 42\left(\frac{z}{r}\right)^2 + 63\left(\frac{z}{r}\right)^4\right)\frac{x}{r} \\ \left(3 - 42\left(\frac{z}{r}\right)^2 + 63\left(\frac{z}{r}\right)^4\right)\frac{y}{r} \\ -\left(15 - 70\left(\frac{z}{r}\right)^2 + 63\left(\frac{z}{r}\right)^4\right)\frac{z}{r} \end{pmatrix} \tag{27}$$

$$\boldsymbol{a}_{J_5} = -\frac{J_5}{8}\left(\frac{\mu}{r^2}\right)\left(\frac{r_{eq}}{r}\right)^5 \mathcal{N}\begin{pmatrix} 3\left(35\left(\frac{z}{r}\right) - 210\left(\frac{z}{r}\right)^3 + 231\left(\frac{z}{r}\right)^5\right)\frac{x}{r} \\ 3\left(35\left(\frac{z}{r}\right) - 210\left(\frac{z}{r}\right)^3 + 231\left(\frac{z}{r}\right)^5\right)\frac{y}{r} \\ \left(15 - 315\left(\frac{z}{r}\right)^2 + 945\left(\frac{z}{r}\right)^4 - 693\left(\frac{z}{r}\right)^6\right) \end{pmatrix} \tag{28}$$

$$\boldsymbol{a}_{J_6} = \frac{J_6}{16}\left(\frac{\mu}{r^2}\right)\left(\frac{r_{eq}}{r}\right)^6 \mathcal{N}\begin{pmatrix} \left(35 - 945\left(\frac{z}{r}\right)^2 + 3465\left(\frac{z}{r}\right)^4 - 3003\left(\frac{z}{r}\right)^6\right)\frac{x}{r} \\ \left(35 - 945\left(\frac{z}{r}\right)^2 + 3465\left(\frac{z}{r}\right)^4 - 3003\left(\frac{z}{r}\right)^6\right)\frac{y}{r} \\ \left(3003\left(\frac{z}{r}\right)^6 - 4851\left(\frac{z}{r}\right)^4 + 2205\left(\frac{z}{r}\right)^2 - 315\right)\frac{z}{r} \end{pmatrix} \tag{29}$$

1.3.3.5 Solar Radiation Disturbance Acceleration A simple solar disturbance acceleration $\boldsymbol{a}_d$ function is implemented where

$$\boldsymbol{a}_d = -\frac{C_r A \Pi}{mcs^3}\boldsymbol{s} \tag{30}$$

where $C_r = 1.3$ is the radiation pressure coefficient, $c = 299792458$m/s is the speed of light, $\Pi = 1372.5398$ W/m^2 is the solar radiation flux and $\boldsymbol{s}$ is the sun position vector relative to the spacecraft in units of AU. The area A is the projected area towards the sun.

2 Library Functions

The AVS astrodynamics support library is developed to facility writing C-code that must perform the associated mathematical functions. It is not a Basilisk module, but included with the main Basilisk distribution as a support library. The library goals are

- **C-Code implementation**: All function calls only use C-code.
- **Linear Algebra**: Basic matrix math function that are common in Matlab and Python environment quickly become tedious when programming in C. To avoid this, a custom written C library is developed to perform specialized 2, 3, 4, and 6 dimensional matrix math, but some linear algebra math on matrices of general size.
- **Rigid Body Kinematics**: Provides a comprehensive C-based software library to convert between a broad range of rigid body attitude coordinates, as well as add and subtract orientations. This library can also compute the differential kinematic equations of select coordinate types. The textbook Reference 1 contains a complete listing of all the attitude library function in Appendix E.
- **Orbital Motion Library**: Provides a comprehensive C-based software library to convert between orbit anomaly angles, as well as orbit elements and Cartesian coordinates. This library also contains some simple space environment modeling functions.

3 Library Assumptions and Limitations

3.1 Assumptions

3.1.1 Linear Algebra Library

The vector dimension can be declared either explicitly through

```
double vec[3]
```

or implicitly through

```
double *vec
```

and allocating the required memory dynamically. However, with the specialized matrix dimensions the matrices are assumed to be defined through commands like

```
double m33[3][3]
```

for a 3×3 matrix.

3.1.2 Rigid Body Kinematics

These attitude kinematic relationships are general enough that no assumptions are made on the mathematics. However, when the 3×3 DCM are defined, they must be of type

```
double BN[3][3]
```

3.1.3 Orbital Motion

This orbital motion library accepts the inertial position and velocity vectors as pointers to a 3-dimensional array. As with the linear algebra library, these can be declared explicitly or implicitly.

3.2 Limitations

3.2.1 Linear Algebra Library

There are no limitations on the provided linear algebra routines. They are written in a general manner.

3.2.2 Rigid Body Kinematics

There are no limitations on the provided linear algebra routines. They are written in a general manner.

3.2.3 Orbital Motion

The orbital motion support library has the following limitations.

- While the `elem2rv()` support rectilinear cases, the inverse mapping in `rv2elem()` does not.
- The gravitational zonal harmonics for J_3-J_6 are only implemented for Earth, and not other celestial objects.
- The atmospheric density and drag, as well as the Debye length models are only implemented for Earth orbiting scenarios.

4 Test Description and Success Criteria

The AVS support libraries undergo a series of unit test within C-functions that are called by the Swig'd interface `avsLibrarySelfCheck`. The automated unit test is performed via `pytest` through the script:

```
SimCode/utilitiesSelfCheck/_UnitTest/test_avsLibrarySelfCheck.py
```

In each unit test the C-code library function output is compared to a externally hand-computed solution. The success criteria is when the pre-computed solution and the C-code computed solution match to within a certain numerical tolerance. The success criteria absolute value tolerances are shown in Table 2.

5 Test Parameters

5.1 Linear Algebra

In these unit tests each function is provided non-trivial inputs, such as `v1 = [1,2,3]` to evaluate the associate matrix math. For specific information on the values used, look at the source code for the function `testLinearAlgebra()` in the file `avsLibrarySelfCheck.c`.

Table 2: Unit Test Absolute Difference Tolerances

Test	Absolute Tolerance
Linear Algebra	10^{-10}
Rigid Body Kinematics	10^{-10}
Orbital Motion: Anomalies	10^{-10}
Orbital Motion: Element conversion	10^{-10}
Orbital Motion: Environment	10^{-10}

5.2 Rigid Body Kinematics

The attitude kinematics sub-routines are provided with non-trivial inputs to test the functions. For specific information on the values used, look at the source code for the function `testRigidBodyKinematics()` in the file `avsLibrarySelfCheck.c`.

5.3 Orbital Motion

The orbital anomaly sub-routines are provided with non-trivial inputs to test the functions. For specific information on the values used, look at the source code for the function `testOrbitalAnomalies()` in the file `avsLibrarySelfCheck.c`.

Regarding the orbit element to Cartesian coordinate conversion routines, an Earth orbit is simulated that is either non-equatorial and non-circular (case 1), non-circular equatorial (case 2), circular and inclined (case 3) and circular equatorial (case 4). The resulting orbit elements used are shown in Table 3.

Table 3: Orbit Elements Used in Orbit Coordinate Conversion Routine Checking

Case	SMA [km]	e	$i[^\circ]$	$\Omega[^\circ$	$\omega[^\circ$	$f[^\circ]$
1	7500	0.5	40	133	113	123
2	7500	0.5	0	133	113	123
3	7500	0.0	40	133	113	123
4	7500	0.0	0	133	113	123

6 Test Results

An automated suite of tests are run to perform unit tests on all the AVS support library functions. The results are shown in Table 4.

Table 4: Integration test results.

Test	Pass/Fail	BSK Error Notes
`rigidBodyKinematics`	PASSED	
`orbitalElements`	PASSED	
`orbitalAnomalies`	PASSED	
`linearAlgebra`	PASSED	
`environment`	PASSED	

7 User Guide

7.1 `linearAlgebra` Library

The linear algebra library provides numerous C-based functions to perform basic matrix math operations. For a complete list of functions supported, consult the `linearAlgebra.h` file.

7.1.1 Vector Operations

The vector related functions all begin with a letter 'v' and are broken down in the following categories depending on if the linear algebra operation is performed on a matrix of a general size, or of a specific

size of 2, 3, 4 or 6.

- if the function is `vXXXX()` the code works on a nx1 matrix of arbitrary length n.
- if the function is `v2XXXX()` the code works on a matrix of length 2 defined as `double vec[2]`.
- if the function is `v3XXXX()` the code works on a matrix of length 2 defined as `double vec[3]`.
- if the function is `v4XXXX()` the code works on a matrix of length 2 defined as `double vec[4]`.

The most extensive set of linear algebra support functions are for operation on 3 and 4 dimensional matrices common with orbital mechanics (position and velocity vector representations) and attitude dynamics (working with 3- and 4-parameter attitude descriptions). The following list details what vector operation each function performs:

- `Copy`: Copies vector $\boldsymbol{v}_1$ into vector $\boldsymbol{v}_2$
- `SetZero`: Returns a zero vector
- `Set`: Sets the vector representation to specific values
- `Add`: Sums $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$
- `Subtract`: Returns the difference $\boldsymbol{v}_1$ - $\boldsymbol{v}_2$
- `Scale`: Returns $\boldsymbol{v}_1$ multiplied by a scalar α
- `Dot`: Returns the dot product $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2$
- `OuterProduct`: Returns the outer product $\boldsymbol{v}_1 \boldsymbol{v}_2^T$
- `Normalize`: Returns a normalized vector $\boldsymbol{v}_1/v_1$
- `MaxAbs`: Returns the largest element of a vector
- `IsEqual`: Checks if two vector represnetations are identical
- `IsZero`: Checks if a vector is full of zero elements
- `Print`: Prints the vector representation to a file
- `Sort`: Sorts the vector elements by size
- `Print`: Prints the vector representation to a file
- `PrintScreen`: Prints the vector representation to the screen

7.1.2 Matrix Operations

The matrix related functions are all labeled with a letter 'M' and are broken down into the following matrix dimension related categories:

- `mXXXXy()`: the code works on a $n \times m$ matrix of arbitrary dimension n and m.
- `m22XXXX()`: the code works on a 2×2 matrix defined through `mat[2][2]`.
- `m33XXXX()`: the code works on a 3×3 matrix defined through `mat[3]3]`.

- `m44XXXX()`: the code works on a 4×4 matrix defined through `mat[4][4]`.

The following list provides an overview of the supported matrix functions. Note that not al dimensions have all functions provided, but the 2, 3, and 4 dimensional matrix support is pretty complete.

- `SetIdentity`: Returns an identity matrix
- `SetZero`: Returns a zero matrix
- `Set`: Creates a matrix with specific values
- `Copy`: Copies the matrix $[M_1]$ into $[M_2]$
- `m33MultM33`: Performs the matrix to matrix multiplication $[M_1][M_2]$
- `m33MultM33t`: Performs the matrix to matrix multiplication $[M_1][M_2]^T$
- `m33tMultM33`: Performs the matrix to matrix multiplication $[M_1]^T[M_2]$
- `m33MultV3`: Computes $[M_1]\boldsymbol{v}_1$
- `m33tMultV3`: Computes $[M_1]^T\boldsymbol{v}_1$
- `v3tMultM33`: Computes $\boldsymbol{v}_1^T[M_1]$
- `v3tMultM33t`: Computes $\boldsymbol{v}_1^T[M_1]^T$
- `Tilde`: Returns the skew-symmetric matrix $[\tilde{\boldsymbol{v}}_1]$
- `Transpose`: Returns $[M_1]^T$
- `Add`: Returns the sum $[M_1] + [M_2]$
- `Subtract`: Returns the difference $[M_1] - [M_2]$
- `Scale`: Returns the scaled matrix $\alpha[M_1]$
- `Trace`: Compute the matrix trace $\sum M_{ii}$
- `Determinant`: Returns the square matrix determinant
- `IsEqual`: Checks if all matrix elements are equal
- `IsZero`: Checks if all matrix elements are zero
- `Print`: Prints the matrix representation to a file
- `PrintScreen`: Prints the matrix representation to the screen
- `Inverse`: Returns the matrix inverse $[M_1]^{-1}$
- `SingularValues`: Computes the singular values of $[M_1]$
- `EigenValues`: Computes the Eigenvalues of $[M_1]$
- `ConditionNumber`: Computes the condition number of $[M_1]$

7.2 `rigidBodyKinematics` Library

The following discussion is a brief overview of the `rigidBodyKinematics` library function notation. Please see Appendix E in Reference 1 for a complete description:

- `XXX2YYY`: Converts the attitude coordinates `XXX` to `YYY`
- `addXXX`: Add the two attitude descriptions $\boldsymbol{x}_1 = \boldsymbol{x}_{\mathcal{B}/\mathcal{F}}$ and $\boldsymbol{x}_2 = \boldsymbol{x}_{\mathcal{F}/\mathcal{N}}$ to return the sequential rotation of first $\boldsymbol{x}_1$ and then $\boldsymbol{x}_2$. Returns is the description $\boldsymbol{x}_3 = \boldsymbol{x}_{\mathcal{B}/\mathcal{N}}$
- `subXXX`: Subtract the two attitude descriptions $\boldsymbol{x}_1 = \boldsymbol{x}_{\mathcal{F}/\mathcal{N}}$ and $\boldsymbol{x}_2 = \boldsymbol{x}_{\mathcal{B}/\mathcal{N}}$ to return the relative orientation $\boldsymbol{x}_3 = \boldsymbol{x}_{\mathcal{F}/\mathcal{B}}$
- `BmatXXX`: Returns the matrix $[B]$ of the differential kinematic equations of the attitude parameters `XXX` in Reference 1. Note the scalar multiplier is not included here. For example, with MRPs[2] $[B]$ is defined as
$$\dot{\boldsymbol{\sigma}}_{\mathcal{B}/\mathcal{N}} = \frac{1}{4}[B(\boldsymbol{\sigma}_{\mathcal{B}/\mathcal{N}})]^{\mathcal{B}}\boldsymbol{\omega}_{\mathcal{B}/\mathcal{N}}$$
- `BinvXXX`: Returns the matrix inverse $[B]^{-1}$ of the attitude parameters `XXX`
- `dX`: Returns the time derivatives of the attitude parameters `XXX` as a function of these parameters and the body angular velocity vector $\boldsymbol{\omega}$
- `Mi`: Returns the single-axis DCM about the $\hat{\boldsymbol{b}}_i$ axis
- `wrapToPi`: Makes sure that an angle lies within $\pm\pi$

7.3 `orbitalMotion` Library

7.3.1 Orbit Anomaly Angle Conversions

To convert between the various anomaly angles, the following functions are defined:

- `double f2E(double f, double e)`
- `double E2f(double Ecc, double e)`
- `double E2M(double Ecc, double e)`
- `double M2E(double M, double e)`, with a change conversion tolerance of 10^{-13} and a maximum iteration limit of 200
- `double f2H(double f, double e)`
- `double H2f(double H, double e)`
- `double H2N(double H, double e)`
- `double N2H(double N, double e)`, with a change conversion tolerance of 10^{-13} and a maximum iteration limit of 200

7.3.2 Orbit Element Conversion

To convert from classical orbit elements to inertial Cartesian coordinates, the function

```
elem2rv(double mu, classicElements *elements, double *rVec, double *vVec)
```

is used where `mu` is the gravitational constant of the 2-body problem, the classical elements are defined through

$$(a, e, i, \Omega, \omega, f)$$

where the anomaly angle is typically given by f, unless the orbit is a rectilinear motion in which case the anomaly angle input is E. The function returns the inertial position and velocity vectors in the arrays `rVec` and `vVec`.

To convert from inertial Cartesian coordinates to classical orbit elements, the function

```
rv2elem(double mu, double *rVec, double *vVec, classicElements *elements)
```

Beyond the classical elements listed above, this routine also stores the radius of perapses r_p and apoapses r_a, as well as $\alpha = \frac{1}{a}$.

7.3.3 Space Environment Functions

7.3.3.1 Atmospheric Density The Earth's atmospheric density ρ is computed using

```
double atmosphericDensity(double alt)
```

The density is returned as a scalar value.

7.3.3.2 Mean Debye Length The mean Debye Length for the near-Earth environment is approximated, very crudely, through a polynomial fit. The function returns the scalar λ_d and is called through

```
double debyeLength(double alt)
```

7.3.3.3 Atmospheric Drag Acceleration To compute an estimate of the Earth's atmospheric drag acceleration, use the function:

```
void atmosphericDrag(double Cd, double A, double m, double *rvec, double *vvec,
double *advec)
```

The inputs are the ballistic drag coefficient `Cd`, the velocity-projected cross-sectional area `A`, as well as the spacecraft mass `m`. Given the inertial position vectors `rvec` and `vvec`, the function returns the drag acceleration `advec`.

7.3.3.4 Gravitational Zonal Harmonics This function returns the inertial acceleration due to a planets zonal Harmonic. The function call is:

```
void jPerturb(double *rvec, int num, double *ajtot, ...)
```

If not option argument is provided, then the zonal harmonics of Earth are simulated. About other celestial objects only the J_2 harmonic is implemented. Here the object is specified through the `CelestialObject_t` enumeration. For example, to get the J_2 zonal harmonic about Venus the argument `CELESTIAL_VENUS` is provided.

7.3.3.5 Solar Radiation Pressure Acceleration To compute the inertial disturbance acceleration due to the solar radiation pressure use the function

```
void solarRad(double A, double m, double *sunvec, double *arvec)
```

Here `A` is the projected surface area, `m` is the spacecraft mass, `sunvec` is the sun position vector to the Sun in units of AU.

REFERENCES

[1] Hanspeter Schaub and John L. Junkins. *Analytical Mechanics of Space Systems*. AIAA Education Series, Reston, VA, 3rd edition, 2014.

[2] Malcolm D. Shuster. A survey of attitude representations. *Journal of the Astronautical Sciences*, 41(4):439–517, 1993.