ZR User API

This is a quick guide to the functions used to control a SPHERES satellite in Zero Robotics. These functions do not change from game to game.

All C functions are accessed as methods of the *api* class (except for the DEBUG and mathematical functions); that is, they are called as: **api.function(arguments)**. Most of the MATLAB functions are also part of the *api* class, but some are part of the standard MATLAB library; the actual calling syntax is the one shown.

BASIC FUNCTIONS

setPositionTarget

Sets a point as the position target

C **void setPositionTarget(float posTarget[3])**

posTarget	array of three floats—x, y, and z position
Return value	None

MATLAB **api.setPostionTarget(posTarget)**

posTarget	three-elements vector—x, y, and z position
Return value	None

setAttitudeTarget

Sets a unit vector direction for the satellite to point toward

C **void setAttitudeTarget(float attTarget[3])**

attTarget	array of three floats—x, y, and z components of unit vector
Return value	None

MATLAB **api.setAttitudeTarget(attTarget)**

posTarget	three-elements vector—x, y, and z components of unit vector
Return value	None

setVelocityTarget

Sets the closed-loop x, y, and z components of the target velocity vector

C **void setVelocityTarget(float velTarget[3])**

posTarget	array of three floats—x, y, and z position
Return value	None

MATLAB **api.setVelocityTarget(velTarget)**

posTarget	three-elements vector—x, y, and z velocity
Return value	None

setAttRateTarget

Sets the closed-loop target rotation rate components on the body frame

C **void setAttRateTarget(float attRateTarget[3])**

posTarget	array of three floats—rotation rates about the x, y, and z axes
Return value	None

MATLAB **api.setAttRateTarget(attRateTarget)**

posTarget	three-elements vector—rotation rates about the x, y, and z axes
Return value	None

setForces

Sets the open-loop x, y, and z forces to be applied to the satellite

C **void setForces(float forces[3])**

forces	array of three floats—x, y, and z forces
Return value	None

MATLAB **api.setForces(forces)**

forces	three-elements vector—x, y, and z forces
Return value	None

setTorques Sets the open-loop x, y, and z torques to be applied to the satellite	C	**void setTorques(float torques[3])** torques — array of three floats—torques about the x, y, and z axes Return value — None
	MATLAB	**api.setTorques(torques)** torques — three-elements vector—torques about the x, y, and z axes Return value — None
getMyZRState Gets the current state of the satellite in the following format: C indices 0-2 Position 3-5 Velocity 6-8 Attitude vector 9-11 Rotation rates	C	**void getMyZRState(float myState[12])** myState — Array of 12 floats where the state will be stored Return value — None
	MATLAB	**myState = api.getMyZRState()** Return value — 12-elements vector with state **Remarks** — MATLAB uses 1-based indexing. E.g., position is in indices 1-3.
getOtherZRState Gets the current state of the *opponent's* satellite in the following format: C indices 0-2 Position 3-5 Velocity 6-8 Attitude vector 9-11 Rotation rates	C	**void getOtherZRState(float otherState[12])** otherState — Array of 12 floats where the state will be stored Return value — None
	MATLAB	**otherState = api.getOtherZRState()** Return value — 12-elements vector with state **Remarks** — MATLAB uses 1-based indexing. E.g., position is in indices 1-3.
getTime Gets the time (in seconds) elapsed since the beginning of the game	C	**unsigned int getTime()** Return value — Time in seconds
	MATLAB	**time = api.getTime()** Return value — Time in seconds
DEBUG Prints the supplied text to the console. Accepts formatted strings in the same format as the standard C printf function.	C	**DEBUG(("Hello World!"))** **DEBUG(("Hello %s!", name))** **DEBUG((const char *message, …))** message — String to be printed or format string using standard C format specifiers … — Arguments to be substituted in format specifiers Return value — None **Remarks** — Make sure to use double parentheses. Do not type *api.* before this function.
	MATLAB	**api.DEBUG('Hello World!')** **api.DEBUG('Hello %s!', name)** **api.DEBUG(message, …)** message — String to be printed or format string using standard C format specifiers … — Arguments to format specifiers Return value — None **Remarks** — Use a single parenthesis and do type *api.* before this function.

ADVANCED

setQuatTarget Specifies a SPHERES quaternion attitude target for the satellite	C	**void setQuatTarget(float quat[4])**	
		quat	target quaternion in [vector scalar] representation
		Return value	None
	MATLAB	**api.setQuatTarget(quat)**	
		quat	target quaternion in [vector scalar] representation
		Return value	None
getMySphState Gets the current SPHERES state (with quaternion attitude) of the satellite in the following format: C indices 0-2 Position 3-5 Velocity 6-9 Attitude quaternion 10-12 Rotation rates	C	**void getMySphState(float myState[13])**	
		myState	Array of 13 floats where the state will be stored
		Return value	None
	MATLAB	**myState = api.getMySphState()**	
		Return value	13-elements vector with state
		Remarks	MATLAB uses 1-based indexing. E.g., position is in indices 1-3.
getOtherSphState Gets the current SPHERES state (with quaternion attitude) of the *opponent's* satellite in the following format: C indices 0-2 Position 3-5 Velocity 6-9 Attitude quaternion 10-12 Rotation rates	C	**void getOtherSphState(float otherState[13])**	
		otherState	Array of 13 floats where the state will be stored
		Return value	None
	MATLAB	**otherState = api.getOtherSphState()**	
		Return value	13-elements vector with state
		Remarks	MATLAB uses 1-based indexing. E.g., position is in indices 1-3.
spheresToZR Converts a 13-elements state SPHERES state to a 12-elements ZR state	C	**void spheresToZR(float stateSph[13], float stateZR[12])**	
		stateSph	13-elements input array
		stateZR	12-elements output array
		Return value	None
	MATLAB	**stateZR = api.spheresToZR(stateSph)**	
		stateSph	13-elements state vector
		stateZR	12-elements state vector
attVec2Quat Finds the quaternion that rotates the unit vector *refVec* to *attVec*. *baseQuat* defines the orientation of the satellite when *refVec* points in the desired direction. Setting *baseQuat* to something other than [0,0,0,1] allows the satellite to be rotated around the reference vector. In ZR,	C	**void attVec2Quat(float refVec[3], float attVec[3], float baseQuat[4], float quat[4])**	
		refVec	unit vector that specifies the body direction corresponding to no rotation. In ZR this is typcially the velcro (-X) face of the satellites, so refVec is {-1,0,0}
		attVec	target attitude vector
		baseQuat	base quaternion (see description)
		quat	output computed quaternion
		Return value	None
		Remarks	All quaternions are in [vector scalar] representation

baseQuat is typically [1,0,0,0] (a 180° rotation about X) to point the tank toward global +Z.

When using this function to find the minimal rotation from the current attitude to a target attitude, it is advised to supply:

- the current pointing direction in *refVec*,
- the desired attitude in *attVec*,
- the current quaternion attitude in *baseQuat*.

Since one of the degrees of freedom is unconstrained, using another approach can result in unexpected rotations about the pointing direction.

MATLAB **quat = api.attVec2Quat(refVec, attVec, baseQuat)**

refVec	unit vector that specifies the body direction corresponding to no rotation. In ZR this is typcially the velcro (-X) face of the satellites, so refVec is [-1,0,0]
attVec	target attitude vector
baseQuat	base quaternion (see description)
quat	output computed quaternion
Remarks	All quaternions are in [vector scalar] representation

---

quat2AttVec

Converts a quaternion into a ZR attitude vector by rotating the supplied unit vector *refVec* with *quat* to determine *attVec*

C **void quat2AttVec(float refVec[3], float quat[4], float attVec[3])**

refVec	unit vector that specifies the body direction corresponding to no rotation. In ZR this is typically the velcro (-X) face of the satellites, so refVec is {-1,0,0}
quat	quaternion rotation applied to refVec
attVec	output attitude vector
Return value	None
Remarks	This function cannot do an in-place rotation, refVec and attVec should be two different variables. All quaternions are in [vector scalar] representation.

MATLAB **attVec = api.quat2AttVec(refVec, quat)**

refVec	unit vector that specifies the body direction corresponding to no rotation. In ZR this is typically the velcro (-X) face of the satellites, so refVec is {-1,0,0}
quat	quaternion rotation applied to refVec
attVec	output attitude vector
Remarks	All quaternions are in [vector scalar] representation

---

setPosGains

Sets the gains for the position controller

C **void setPosGains(float P, float I, float D)**

P	proportional position gain
I	integral position gain
D	derivative position gain
Return value	None

MATLAB **api.setPosGains(P, I, D)**

P	proportional position gain
I	integral position gain
D	derivative position gain

setAttGains Sets the gains for the position controller	C	void **setAttGains(float P, float I, float D)** P — proportional attitude gain I — integral attitude gain D — derivative attitude gain Return value — None
	MATLAB	**api.setAttGains(P, I, D)** P — proportional attitude gain I — integral attitude gain D — derivative attitude gain
setCtrlMeasurement Sets the state measurement to be used in the standard ZR controllers instead of the default from **getMySphState**	C	void **setCtrlMeasurement(float myState[13])** myState — 13-elements state array Return value — None
	MATLAB	**api.setCtrlMeasurement(myState)** myState — 13-elements state vector
setControlMode Sets the control mode for position and attitude. The default is PD for position and PID for attitude.	C	void **setControlMode(CTRL_MODE posCtrl, CTRL_MODE attCtrl)** posCtrl — either CTRL_PD or CTRL_PID attCtrl — either CTRL_PD or CTRL_PID Return value — None
	MATLAB	**api.setControlMode(posCtrl, attCtrl)** posCtrl — either CTRL_MODE.CTRL_PD or CTRL_MODE.CTRL_PID attCtrl — either CTRL_MODE.CTRL_PD or CTRL_MODE.CTRL_PID
setDebug Adds an array of 7 user-defined debugging values to the satellite telemetry. The data can then be plotted with the ZR plotting tools.	C	void **setDebug(float values[7])** values — 7 debug values array Return value — None
	MATLAB	**api.setDebug(myState)** values — 7 debug values vector

VECTOR, MATRIX FUNCTIONS

---

mathSquare

Calculates the square of a scalar number

C **float mathSquare(float a)**

a	input scalar float
Return value	square of input

MATLAB **b = a^2**

a	input scalar
b	squared value

---

mathMatMatMult

Matrix multiply:
c = a * b

C **void mathMatMatMult(float *c, float *a, float *b, int nra, int nca, int ncb)**

c	output matrix
a	left matrix
b	right matrix
nra	number of rows in matrix a
nca	number of colums in matrix a
ncb	number of colums in matrix b
Return value	None

MATLAB **c = a * b**

a, b	left, right matrices
c	output matrix

---

mathMatMatTransposeMult

Matrix vector multiply with transpose:
$c = a * b^T$

C **void mathMatMatTransposeMult(float *c, float *a, float *b, int nra, int nca, int nrb)**

c	output matrix
a	left matrix
b	right matrix
nra	number of rows in matrix a
nca	number of colums in matrix a
ncb	number of rows in matrix b (and columns in b')
Return value	None

MATLAB **c = a * b'**

a, b	left, right matrices
c	output matrix

---

mathMatTransposeMatMult

Matrix vector multiply with transpose:
$c = a^T * b$

C **void mathMatTransposeMatMult(float *c, float *a, float *b, int nra, int nca, int nrb)**

c	output matrix
a	left matrix
b	right matrix
nra	number of rows in matrix a (and rows in b)
nca	number of colums in matrix a
ncb	number of colums in matrix b
Return value	None

MATLAB **c = a' * b**

a, b	left, right matrices
c	output matrix

---

mathMatAdd

Matrix addition:
c = a + b

C **void mathMatAdd(float *c, float *a, float *b, int nrows, int ncols)**

c	output matrix
a	left matrix
b	right matrix
nrows	number of rows in matrices a, b, and c
ncols	number of colums in matrices a, b, and c
Return value	None

MATLAB **c = a + b**

a, b	input matrices (or vectors)
c	output matrix (or vector)

mathInvert3x3

Inverts a 3×3 matrix

C **int mathInvert3x3(float inv[3][3], float mat[3][3])**

inv	inverted output matrix
mat	input matrix
Return value	0 if successful

MATLAB **c = inv(a)**

a	input matrix
c	output matrix
Remarks	Accepts all matrix sizes.

mathSkewSymmetric

Creates the skew symmetric matrix S(A), where:

A = [x; y; z]

S(A) = [0 –z y; z 0 –x; –y x 0]
= $-S(A)^T$

C **void mathSkewSymmetric(float *a, float *s)**

a	vector of length 3 (*x, y, z*)
s	output array of length 9 that represents matrix S
Return value	0 if successful

MATLAB **s = [0 -a(3) a(2) ; a(3) 0 -a(1) ; -a(2) a(1) 0]**

a	vector of length 3 (*x, y, z*)
s	output 3×3 matrix S

mathMatVecMult

Matrix vector multiply:
c = a * b

C **void mathMatVecMult(float *c, float *a, float *b, int rows, int cols)**

c	output vector (of length *rows*)
a	input matrix (of size *rows*×*cols*)
b	input vector (of length *cols*)
rows	number of matrix rows
cols	number of matrix cols
Return value	None

MATLAB **c = a * b**

a	input matrix ($n \times m$)
b	input vector ($m \times 1$)
c	output vector ($n \times 1$)

mathVecAdd

Vector addition:
c = a + b

C **void mathVecAdd(float *c, float *a, float *b, int n)**

c	output vector
a	left vector
b	right vector
n	length of vectors
Return value	None

MATLAB **c = a + b**

a, b	input vectors (or matrices)
c	output vector (or matrix)

mathVecSubtract Vector subtraction: c = a – b	C	**void mathVecSubtract(float *c, float *a, float *b, int n)**	
		c	output vector
		a	left vector
		b	right vector
		n	length of vectors
		Return value	None
	MATLAB	**c = a - b**	
		a, b	input vectors (or matrices)
		c	output vector (or matrix)
mathVecOuter Outer product of column vectors: $c = a * b^T$	C	**void mathVecOuter(float *c, float *a, float *b, int nrows, int ncols)**	
		c	output matrix (of size *nrows*×*ncols*)
		a	input vector (of length *rows*)
		b	input vector (of length *cols*)
		rows	number of rows in output matrix
		cols	number of columns in output matrix
		Return value	None
	MATLAB	**c = a * b'**	
		a	input column vector (length *n*)
		b	input column vector (length *m*)
		c	output vector (size *n*×*m*)
mathVecInner Inner product of column vectors: $c = a^T * b$	C	**float mathVecInner(float *a, float *b, int n)**	
		a	input vector (of length *n*)
		b	input vector (of length *n*)
		n	length of vectors
		Return value	scalar result of inner product
	MATLAB	**c = a' * b**	
		a, b	input column vectors
		c	output scalar
mathVecMagnitude Calculates the magnitude of the supplied vector	C	**float mathVecMagnitude(float *a, int n)**	
		a	input vector
		n	length of vector (number of elements)
		Return value	Magnitude of vector
	MATLAB	**r = norm(a)**	
		a	input vector
		Return value	Magnitude of vector
mathVecNormalize Normalizes the supplied vector	C	**float mathVecNormalize(float *a, int n)**	
		a	input vector
		n	length of vector (number of elements)
		Return value	Magnitude of vector before normalization – useful when simultaneously computing direction and distance
	MATLAB	**a = a ./ norm(a)**	
		a	input vector

mathVecCross

Calculates the 3×3 cross product:
c = a × b

C **void mathVecCross(float c[3], float a[3], float b[3])**

c	output vector
a	left vector
b	right vector
Return value	None

MATLAB **c = cross(a, b)**

a	input vector
b	input vector
Return value	output vector

mathBody2Global

Creates a body to global frame rotation matrix. The output matrix converts body frame vectors to global vectors.

C **void mathBody2Global(float body2Glo[3][3], float *state)**

body2Glo	3×3 rotation matrix output
state	13-elements state vector returned by **getMySphState**
Return value	None

MATLAB **b2g = api.mathBody2Global(state)**

state	13-elements state vector returned by **getMySphState**
Return value	3×3 rotation matrix

quat2matrixOut

Calculates the rotation matrix needed to transform a vector from *body frame* →
to *global frame* from a given attitude quaternion.

C **void quat2matrixOut(float mat[3][3], float quat[4])**

mat	3×3 rotation matrix output
quat	quaternion in [vector scalar] representation
Return value	None

MATLAB **mat = api.quat2matrixOut(quat)**

quat	quaternion in [vector scalar] representation
Return value	3×3 rotation matrix

quat2matrixIn

Calculates the rotation matrix needed to transform a vector from *global frame* →
to *body frame* from a given attitude quaternion.

C **void quat2matrixIn(float mat[3][3], float quat[4])**

mat	3×3 rotation matrix output
quat	quaternion in [vector scalar] representation
Return value	None

MATLAB **mat = api.quat2matrixIn(quat)**

state	quaternion in [vector scalar] representation
Return value	3×3 rotation matrix

quatMult

Calculates the quaternion multiplication:
$q_3 = q_1 q_2$
This is equivalent to the composition of rotation matrices $R_3 = R_1 * R_2$

C **void quatMult(float *q3, float *q1, float *q2)**

q3	quaternion product output
q1	left quaternion input
q2	right quaternion input
Return value	None
Remarks	All quaternions are in [vector scalar] representation

MATLAB **q3 = api.quatMult(q1, q2)**

q1	left quaternion
q2	right quaternion
Return value	quaternion product
Remarks	All quaternions are in [vector scalar] representation

MATHEMATICAL FUNCTIONS

These are standard library functions and are not part of the *api* class, so they are called without prepending "api."

C: MATLAB:	**float sqrtf(float x)** **y = sqrt(x)**	Calculates the square root of x
C: MATLAB:	**float expf(float x)** **y = exp(x)**	Calculates e^x
C: MATLAB:	**float logf(float x)** **y = log(x)**	Calculates the natural logarithm of x: ln(x)
C: MATLAB:	**float log10f(float x)** **y = log10(x)**	Calculates the base 10 logarithm of x: $\log_{10}(x)$
C: MATLAB:	**float powf(float x, float y)** **x^y**	Raises the base x to the power y: x^y
C: MATLAB:	**float sinf(float x)** **y = sin(x)**	Computes the trigonometric sine function: sin(x)
C: MATLAB:	**float cosf(float x)** **y = cos(x)**	Computes the trigonometric cosine function: cos(x)
C: MATLAB:	**float tanf(float x)** **y = tan(x)**	Computes the trigonometric tangent function: tan(x)
C: MATLAB:	**float asinf(float x)** **y = asin(x)**	Computes the trigonometric arcsine function: $\sin^{-1}(x)$
C: MATLAB:	**float acosf(float x)** **y = acos(x)**	Computes the trigonometric arccosine function: $\cos^{-1}(x)$
C: MATLAB:	**float atanf(float x)** **y = atan(x)**	Computes the trigonometric arctangent function: $\tan^{-1}(x)$ The output is in the range $[-\pi/2, \pi/2]$
C: MATLAB:	**float atan2f(float y, float x)** **y = atan2(x)**	Computes the four quadrant arctangent function: $\tan^{-1}(y/x)$ The output is in the range $[-\pi, \pi]$
C: MATLAB:	**float sinhf(float x)** **y = sinh(x)**	Computes the hyperbolic sine function: sinh(x)
C: MATLAB:	**float coshf(float x)** **y = cosh(x)**	Computes the hyperbolic cosine function: cosh(x)
C: MATLAB:	**float tanhf(float x)** **y = tanh(x)**	Computes the hyperbolic tangent function: tanh(x)
C:	**float ceilf(float x)**	Rounds the supplied float up to the nearest integer towards $+\infty$

MATLAB:	**y = ceil(x)**	
C: MATLAB:	**float floorf(float x)** **y = floor(x)**	Rounds the supplied float down to the nearest integer towards $-\infty$
C: MATLAB:	**float fabsf(float x)** **y = abs(x)**	Computes the absolute value of the argument: $\lvert x \rvert$
C: MATLAB:	**float idexpf(float mant, int exp)** **y = mant * 2 ^ exp**	Calculates: $mant * 2^{exp}$
C: MATLAB:	**float frexpf(float value, int *exp)** **[mant, exp] = log2(value)**	Separates the floating point argument *value* into a normalized mantissa (returned value in C) and exponent (*exp*) so that: *mant* * 2 ^ *exp* = *x*
C: MATLAB:	**float fmodf(float num, float den)** **y = rem(num, den)**	Computes the floating point remainder of the operation *num*/*den*
C: MATLAB:	**float modff(float value, float *i)** **frac = rem(value, 1)** **i = fix(value)**	Separates the floating point argument *value* into fractional (returned value in C) and integral (*i*) parts. Note: Handling of ±Inf and NaN in C differs from MATLAB