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 of them except DEBUG are accessed as members of the api object; that is, they are called as api.function(Arguments).

BASIC

void setPositionTarget(float posTarget[3])	Sets a point as the position target Argument: array of three floats—x, y, and z position Return value: None
void setAttitudeTarget(float attTarget[3])	Sets a unit vector direction for the satellite to point toward Argument: array of three floats—x, y, and z components of unit vector Return value: None
void setVelocityTarget(float velTarget[3])	Sets the closed-loop x, y, and z components of the target velocity vector Argument: array of three floats—x, y, and z velocity Return value: None
void setAttRateTarget(float attRateTarget[3])	Sets the closed-loop target rotation rate components on the body frame Argument: array of three floats—rotation rates about the x, y, and z axes Return value: None
void setForces(float forces[3])	Sets the open-loop x, y, and z forces to be applied to the satellite Argument: array of three floats—x, y, and z forces Return value: None
void setTorques(float torques[3])	Sets the open-loop x, y, and z torques to be applied to the satellite Argument: array of three floats—torques about the x, y, and z axes Return value: None
void getMyZRState(float myState[12])	Gets the current state of the satellite in the following format: Places/indices 0-2: Position 3-5: Velocity 6-8: Attitude vector 9-11: Rotation rates Arguments: Array of 12 floats to store the state Return value: None
void getOtherZRState(float otherState[12])	Same as getMyZRState but gets the state of the opponent's satellite

unsigned int getTime()	Gets the time (in seconds) elapsed since the beginning of the game NOTE: This function is new for the 2013 season. Arguments: None Return value: Unsigned int containing time in seconds
DEBUG(("Some text!"))	Prints the supplied text to the console. Accepts formatted strings in the same format as the standard C printf function. NOTE: Make sure to use double parentheses. Do not type api. before this function. Arguments: String to be printed Return value: None

ADVANCED

void setQuatTarget(float quat[4])	Specifies a SPHERES quaternion attitude target for the satellite. Note that the scalar part of the quaternion Argument: array of four floats—quaternion components Return value: None
void getMySphState(float myState[13])	Gets the current SPHERES state (with quaternion attitude) for the satellite in the following format: Places/indices 0-2: Position 3-5: Velocity 6-9: Attitude quaternion 10-12: Rotation rates Arguments: Array of 13 floats to store the state Return value: None
void getOtherSphState(float otherState[13])	Same as getMySphState but gets the state of the opponent's satellite
void spheresToZR(float stateSph[13], float stateZR[12])	Converts a 13-element state SPHERES state to a 12-element ZR state Arguments: Array of 13 floats containing a SPHERES state and an array of 12 floats to store the ZR state Return value: None
void attVec2Quat(float refVec[3], float attVec[3], float baseQuat[4], float quat[4])	Finds the quaternion that rotates refVec to attVec. This function determines the quaternion rotation from a user unit vector in the global frame. 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, baseQuat is typicaly {1,0,0,0} 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, and 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. Arguments: 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—unit vector specifying the desired pointing direction baseQuat—quaternion specifying if there should be an initial rotation applied to the reference frame before calculating the output quaternion. For a tank-down

	nominal attitude, this should be {1,0,0,0} for a 180 degree rotation about X. quat—quaternion converted from attVec Return value: None
void quat2AttVec(float refVec[3], float quat[4], float attVec[3])	Converts a quaternion into a ZR attitude vector by rotating the supplied unit vector refVec using quat to determine the direction of attVec. NOTE: refVec is not copied to local storage, so it should be a different variable from attVec. Arguments: 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 to convert to ZR attitude vector attVec—converted attitude vector
void setPosGains(float P, float I, float D)	Sets the position controller gains Arguments: float P (proportional gain), float I (integral gain), float D (derivative gain) Return value: None
void setAttGains(float P, float I, float D)	Sets the attitude controller gains Arguments: float P (proportional gain), float I (integral gain), float D (derivative gain) Return value: None
void setCtrlMeasurement(float myState[13])	Sets the state measurement to be used in the standard ZR controllers instead of the default getMySphState() Arguments: float state[13] Return value: None
void setControlMode(CTRL_MODE posCtrl, CTRL_MODE attCtrl)	Sets the control mode for position and attitude control. The default is PD for position and PID for attitude. Arguments: Each of the two Arguments should be one of the two macros CTRL_PD and CTRL_PID Return value: None

void setDebug(float values[7])	Adds an array of 7 user-defined debugging values to the satellite telemetry. The data can then be plotted with the ZR plotting tools. Arguments: Array of 7 floats Return value: None

VECTOR, MATRIX FUNCTIONS

float mathSquare(float a)	Calculates a*a Arguments: float a Return value: float containing calculated value
void mathMatMatMult(float (c[], float *a[], int nra, int nca, int ncb)	Matrix multiply: c = a * b Arguments: float *c – matrix output float **a – left matrix float *b – right matrix int nra – number of rows in matrix a int nca – number of columns in matrix a int ncb – number of columns in matrix b Return value: None
void mathMatMatTransposeMult(float*c[], float *a[], float *b[], int nra, int nca, int nrb)	Matrix vector multiply with transpose: c = a * b' Arguments: float *c[] – matrix output float *a[] – left matrix float *b[] – right matrix int nra – number of rows in matrix a int nca – number of columns in matrix a int nrb – number of rows in matrix b (and columns in b') Return value: None
void mathMatTransposeMatMult(float *c[], float *a[], float *b[], int nra, int nca, int nrb)	Matrix vector multiply with transpose: c = a' * b Arguments: float *c[] – matrix output float *a[] – left matrix float *b[] – right matrix int nra – number of rows in matrix a (and rows in b) int nca – number of columns in matrix a int ncb – number of columns in matrix b Return value: None
void mathMatAdd(float *c[], float *a[], float *b[], int nrows, int ncols)	Matrix addition: c = a + b Arguments: float *c[] – matrix output float *a[] – left matrix float *b[] – right matrix int nrows – number of rows in matrices a, b, and c int ncols – number of columns in matrices a, b, and c Return value: None
int mathInvert3x3(float inv[3][3], float mat[3][3])	Inverts a 3x3 matrix Arguments: float mat[3][3] – matrix input float inv[3][3] – inverted matrix output Return value: 0 if successful
void mathSkewSymmetric(float *a, float *s)	Creates the skew symmetric matrix S(A), where A = [X; Y; Z] and

	S(A) = [0 –Z Y; Z 0 -X; -Y X 0] Arguments: float *a – vector of length 3 (x, y, z) float *s – output array of length 9 that represents matrix S Return value: None
void mathMatVecMult(float *c, float **a, float *b, int rows, int cols)	Matrix vector multiply: c = a * b Arguments: float *c – array for vector output of length rows float **a – matrix input rows x cols float *b – vector input of length cols int rows – number of matrix rows int cols – number of matrix columns Return value: None
void mathVecAdd(float *c, float *a, float *b, int n)	Vector addition: c = a + b Arguments: float *c – vector output float *a – first vector input float *b – second vector input int n – length of vectors (number of components – usually 3) Return value: None
void mathVecSubtract(float *c, float *a, float *b, int n)	Vector subtraction: c = a - b Arguments: float *c – vector output float *a – first vector input float *b – second vector input int n – length of vectors (number of components – usually 3) Return value: None
void mathVecOuter(float *c[], float *a, float *b, int nrows, int ncols)	Outer product: c = a * b' Arguments: float *c[] – matrix output, nrows by ncols float *a – nrows in length float *b – ncols in length int nrows – number of rows in output matrix int ncols – number of columns in output matrix Return value: None
float mathVecInner(float *a, float *b, int n)	Vector inner product: a' * b Arguments: float *a – first vector of length n float *b – second vector of length n int n – length of vectors Return value: float containing calculated value as a scalar

float mathVecNormalize(float *a, int n)	Normalizes the supplied vector Arguments: float *a – input vector int n – length of vector Return value: length of the vector before normalization – useful when simultaneously computing direction and distance
float mathVecMagnitude(float *a, int n)	Calculates the magnitude of the supplied vector Arguments: float *a – input vector int n – length of vector Return value: float containing calculated value
void mathVecCross(float c[3], float a[3], float b[3])	Calculates the 3x3 cross product c = a x b Arguments: float c[3] – vector ouput float a[3] – left vector input float b[3] - right vector input Return value: None
void mathBody2Global(float body2Glo[3][3], float *state)	Creates a body to global frame rotation matrix Arguments: float *state – SPHERES state vector (NOT quaternion) float body2Glo[3][3] – 3x3 rotation matrix output The matrix output converts body frame vectors to global frame vectors. Return value: None
void quat2matrixOut(float mat[3][3], float quat[4])	Produces the rotation matrix needed (specified by quat[4]) to transform a vector from the body frame to the global reference frame Note: function assumes [vector scalar] quat representation Arguments: float quat[4] – unit quaternion input representing satellite attitude float mat[3][3] – 2x3 rotation matrix output from body to global reference frame Return value: None
void quat2matrixIn(float mat[3][3], float quat[4])	Produces the rotation matrix needed (specified by quat[4]) to transform a vector from the global reference frame to the body frame Note: function assumes [vector scalar] quat representation Arguments: float quat[4] – unit quaternion input representing satellite attitude float mat[3][3] – 2x3 rotation matrix output from global o

	body reference frame Return value: None
void quatMult(float *q3, float *q1, float *q2)	Calculates the quaternion multiplication q3 = q1 * q2. This is equivalent to the composition of rotation matrices R3 = R1 * R2. The operation is commutative. Arguments: float *q3 – quaternion output float *q1 – left quaternion input float *q2 - right quaternion input Return value: None

MATH FUNCTIONS

float sqrtf(float x)	Calculates the square root of x Argument: float x Return value: float containing calculated value
float expf(float x)	Calculates e^x Argument: float x Return value: float containing calculated value
float logf(float x)	Calculates the natural logarithm of x: ln(x) Argument: float x Return value: float containing calculated value
float log10f(float x)	Calculates the base 10 logarithm of x: $\log_{10}(x)$ Argument: float x Return value: float containing calculated value
float powf(float x, float y)	Raises the supplied base to the supplied power: x^y Arguments: float x (base), float y (power) Return value: float containing calculated value
float sinf(float x)	Computes the trigonometric sine function: sin(x) Arguments: float x Return value: float containing calculated value
float cosf(float x)	Computes the trigonometric cosine function: cos(x) Arguments: float x Return value: float containing calculated value
float tanf(float x)	Computes the trigonometric tangent function: tan(x) Arguments: float x Return value: float containing calculated value
float asinf(float x)	Computes the trigonometric arcsine function: $\sin^{-1}(x)$ Arguments: float x Return value: float containing calculated value
float acosf(float x)	Computes the trigonometric arccosine function: $\cos^{-1}(x)$ Arguments: float x Return value: float containing calculated value
float atanf(float x)	Computes the trigonometric arctangent function: $\tan^{-1}(x)$ Arguments: float x Return value: float containing calculated value
float atan2f(float y, float x)	Computes the two-Argument arctangent function Arguments: float x, float y Return value: float containing calculated value – an angle from $-\pi$ to π appropriate to the quadrant of (x,y)
float sinhf(float x)	Computes the hyperbolic sine function: sinh(x) Arguments: float x Return value: float containing calculated value

float coshf(float x)	Computes the hyperbolic cosine function: cosh(x) Arguments: float x Return value: float containing calculated value
float tanhf(float x)	Computes the hyperbolic tangent function: tanh(x) Arguments: float x Return value: float containing calculated value
float ceilf(float x)	Rounds the supplied float up to the nearest integer Arguments: float x Return value: float containing calculated value
float floorf(float x)	Rounds the supplied float down to the nearest integer (equivalent to truncating the float) Arguments: float x Return value: float containing calculated value
float fabsf(float x)	Computes the absolute value of the Argument: $\|x\|$ Arguments: float x Return value: float containing calculated value
float idexpf(float x, int exp)	Multiplies the Argument by 2 to the integer power of the exponent: $x * 2^{exp}$ Arguments: float x, int exp Return value: float containing calculated value
float frexpf(float x, int *exp)	Separates the Argument into a fractional component and an integer exponent Arguments: float x, int exp Return value: float containing the fractional component The integer exponent is stored in the Argument exp.
float modff(float value, float *iptr)	Breaks value into fractional and integral parts Arguments: float value, float *iptr Return value: float containing the fractional component The integral component is stored as in the Argument iptr.
float fmodf(float numerator, float denominator)	Computes the floating point remainder of the operation numerator/denominator Arguments: float numerator, float denominator Return value: float containing calculated value