

SolarSystem Version 09-JAN-2015

Author: Gerhard HOLTKAMP

SolarSystem is a C++ Class which provides functions for positions and physical parameters of solar system objects (the Sun, planets, the Earth-Moon and various Jupiter and Saturn moons) which can then be included by other programs.

INITIALIZATION AND SETTINGS

The constructor **SolarSystem()** will initialize with the current date and time and the Earth as the central body. The timezone will be set to 0. The difference TT - UTC will be set to automatic. The epoch of the (equatorial) coordinates will be set to Mean of J2000.0.

Use **setTimezone** to set the timezone in which times are being displayed. The functions **setDeltaTAI_UT** and **setAutoTAI_UTC** let you set the difference between TT and UTC but there is normally no need for these unless you have very specific ideas of what you want.

With **setCentralBody** you can have all the information calculated with respect to that planet (or the Moon) rather than with respect to the Earth.

Use **setEpoch** and **setNutation** to set the epoch for which the (equatorial) coordinates are calculated.

To set the time for which the calculations are to be performed use **setCurrentMJD**.

GETTING THE EPHEMERIDES

For getting the Right Ascension and Declination of the various solar system objects with regard to the selected central body (default the Earth) call **getSun**, **getMoon**, **getMercury**, **getVenus**, **getEarth**, **getMars**, **getJupiter**, **getSaturn**, **getUranus**, **getNeptune**, **getIo**, **getEuropa**, **getGanymede**, **getCallisto**, **getRhea**, **getTitan**, **getMimas**, **getEnceladus**, **getDione**, **getUser**.

PHYSICAL EPHEMERIDES

To get the physical ephemerides (apparent diameter, magnitude and phase) of the various objects call **getPhysSun**, **getPhysMercury**, **getPhysVenus**, **getPhysEarth**, **getPhysMars**, **getPhysJupiter**, **getPhysSaturn**, **getPhysUranus**, **getPhysNeptune**, **getPhysIo**, **getPhysEuropa**, **getPhysGanymede**, **getPhysCallisto**, **getPhysRhea**, **getPhysTitan**, **getPhysMimas**, **getPhysEnceladus**, **getPhysDione**, **getPhysUser**, **putConstUser**, **getDiamMoon**, **getLunarPhase**, **getLunarLibration**.

NOTE: The calculations for the Moon are done only if the Earth is selected as central body.

PLANETOGRAPHIC COORDINATES

To find the directions in the sky with regard to the surface of the selected central body use **getPlanetocentric** and **getPlanetographic**. This allows to map the sky with regard to any of the solar system objects. **getSkyRotAngles** provides the three rotation angles for transforming from Equatorial Celestial coordinates into planetocentric ones.

COMETS AND ASTEROIDS

Positions, apparent magnitude and distance of comets and asteroids can be calculated with the following functions: **putOrbitElements**, **putEllipticElements**, **getOrbitPosition**, **getDistance**, **getCometMag** and **getAsteroidMag**.

EXAMPLES

Here are two examples of how to use SolarSystem. (Examples of calculating comets and asteroids as well as defining a User Object is at the end.)

Example 1:

Find the position of the Moon and of Mars with regard to the Earth at the current system time. Also determine the apparent size, phase and magnitude of these objects.

```
...
double ra_moon, dec_moon, mag_moon, size_moon, phase_moon, ildisk;
double ra_mars, dec_mars, mag_mars, size_mars, phase_mars;
SolarSystem plts;

// Right Ascension and Declination in format HH.MMSS and DDD.MMSS in J2000.0
plts.getMoon(ra_moon,dec_moon);
plts.getMars(ra_mars, dec_mars);

plts.getPhysMars( size_mars, mag_mars, phase_mars);
size_mars *= 206264.806247; // convert from radians into arcsec

size_moon = getDiamMoon();
size_moon *= 3437.746771; // convert from radians into arcmin.

getLunarPhase(phase_moon, ildisk, mag_moon);
...
```

Example 2:

Find where on the surface of Mars the Sun, Earth and the star Sirius are at the zenith at 27-SEP-2013, 12:35:00 UTC.

```
...
double ra, decl;
double long_sun, lat_sun;
double long_earth, lat_earth;
double long_sirius, lat_sirius;
char ltxt[10];

strcpy(ltxt,"Mars");

plts.setCentralBody(ltxt); // refer calculations to the center of Mars.

plts.setCurrentMJD (2013, 9, 27, 12, 35, 0);

plts.getSun (ra, decl);
plts.getPlanetographic(ra, decl, long_sun , lat_sun); // coordinates in decimal degrees

plts.getEarth (ra, decl);
plts.getPlanetographic(ra, decl, long_earth , lat_earth); // coordinates in decimal degrees

// the coordinates of Sirius are R.A. 6h45m08.4s and Decl. -16°43'14.8"
plts.getPlanetographic(6.45084, -16.43148, long_sirius, lat_sirius);
...
```

CREDITS

The following books have been invaluable for coming up with the necessary calculations:

Explanatory Supplement to the Astronomical Almanac, University Science Books, Mill Valley, California, 1992. Chapter 7 of that same book gives details about the calculation of physical ephemerides and chapter 2 contains formulas for the calculation of historical delta TT - UT data.

Formulas for the calculation of the phases of the Moon as well as a number of other useful topics can be found in

Jean Meeus, "Astronomical Formulae for Calculators", 4th edition, Willmann-Bell, Inc, Richmond, Virginia, 1988.

The calculation of Sun and the Moon with reasonably high precision suitable for use in PCs, calculation of solar eclipses, coordinate conversions and many other useful algorithms are covered in O.Montebruck and T.Pfleger, "Astronomy with a PC", Springer Verlag, Berlin, Heidelberg, New York, 1989. This book also contains a good formula for the delta TT - UTC calculation.

Back issues of the Astronomical Almanac in addition to the above mentioned books provided useful data for testing the program and on occasion to modify some of the formulas to better fit the needs.

COMPILATION:

The following program modules are needed for including **SolarSystem** in your program:

solarsystem.cpp and **solarsystem.h** for the definition of the respective class.

attlib.cpp and **attlib.h** for vector and matrix calculations and **astrolib.cpp**, **astrolib.h**, **astr2lib.cpp** and **astr2lib.h** for needed astronomical functions.

LICENSE:

SolarSystem is free open source software under GNU LGPL Version 2+.

Author: Gerhard HOLTKAMP

Detailed Description of public functions of SolarSystem:

SolarSystem::SolarSystem()
SolarSystem::~~SolarSystem()

SolarSystem will be initialized with the following parameters: Current system time, Timezone 0, TT-UTC on automatic, sky coordinates refer to J2000.0, no nutation, central body: Earth.

Any of the public functions can be called immediately and should yield meaningful data.

void SolarSystem::setTimezone(double d)

Set the timezone to *d* hours. Central European Time would be +1, American Eastern Standard Time -5, Indian Standard Time +5.5.

SolarSystem uses UTC (Universal Time Coordinated) internally, irrespective of the timezone. The timezone is only needed for entering a time with **setCurrentMJD** or retrieving a time via **getDatefromMJD**.

void SolarSystem::setDeltaTAI_UTC(double d)

Set the difference between TAI (International Atomic Time) and UTC (Universal Time Coordinated) to *d*. This parameter is provided by the International Earth Rotation Service (IERS) and is always an integral number of seconds. In some years leap seconds are inserted (either on December 31 or June 30) to make sure UTC stays within 0.9 sec of the astronomically determined Earth rotation. At the beginning of 1-JANUARY-2009 TAI - UTC was 34 seconds. A leap second will be introduced at the end of 30-JUNE-2012 and then TAI - UTC will be 35 seconds.

Internally, SolarSystem works with the difference TT - UTC (Terrestrial Time - UTC) which differs from TAI-UTC by a constant offset of 32.184 seconds. Thus at the beginning of 1-JULY-2012 TT-UTC will be 67.183 seconds.

Normally there is no need for you to worry about **setDeltaTAI-UTC** because the automatic calculation done by SolarSystem stays within one or two seconds of the actual values for at least the past century and is likely to stay close for the next one or two decades at least. This is accurate enough given the general accuracy of SolarSystem. (But TAI-UTC cannot properly be predicted into the future because of irregularities of the Earth rotation and have to be determined by observation.)

void SolarSystem::setAutoTAI_UTC()

Set the difference TAI-UTC (needed for internal calculations) to automatic. This is the recommended way for SolarSystem. See also **setDeltaTAI_UTC**.

void SolarSystem::setCurrentMJD(int year, int month, int day, int hour, int min, double sec)

Sets the (MJD-) time used for calculations to the date and time given by *year*, *month*, *day*, *hour*, *min* and *sec*. This time is supposed to be local time which takes into account the local timezone.

void SolarSystem::setCurrentMJD()

An overloaded function to set the (MJD-) time used for calculations to the current system time.

double SolarSystem::getMJD(int year, int month, int day, int hour, int min, double sec)

Returns the (MJD-) time corresponding to the date and time given by *year*, *month*, *day*, *hour*, *min* and *sec*. This time is supposed to be local time which takes into account the local timezone. This function is provided simply as a convenience to convert into MJD. This time will not be stored internally unlike the time set by **setCurrentMJD**.

void SolarSystem::getDatefromMJD(double mjd, int & year, int & month, int & day, int & hour, int & min, double & sec)

Convert the Modified Julian Date *mjd* into the corresponding *year*, *month*, *day*, *hour*, *min* and *sec*. The date will be corrected for the respective timezone (so if *mjd* signified UTC, the output will be in local time).

This function was provided as a convenience but will normally not be needed in SolarSystem.

void SolarSystem::setEpoch (double yr)

Set the epoch to which the Right Ascension and Declination refer in the various calls to the year *yr*. SolarSystem uses J2000.0 as default.

void SolarSystem::setNutation (bool nut)

If *nut* is *true* nutation Right Ascension and Declination will be corrected for nutation. Otherwise Mean of Epoch will be assumed (this is the default for SolarSystem).

void SolarSystem::setCentralBody (char* pname)

Set the body to which the various calculations refer to *pname*. The following names are possible: "Sun", "Moon", "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune", "Io", "Europa", "Ganymede", "Callisto", "Rhea", "Titan", "Mimas", "Enceladus", "Dione", "User".

Default is "Earth".

void SolarSystem::includeUser (bool uact)

A user-defined solar system object (typically a comet or an asteroid) can be included and even set as a central body like any other pre-defined object. To do so you must first store the (ecliptic) Keplerian elements via **putOrbitUser** or **putEllipticUser**. This will enable the user-defined object which will then be calculated together with all the pre-defined objects. To enable this object call **includeUser (false)**. To re-enable this object call **includeUser (false)**.

Typically this only makes sense if you also store the rotation elements of this object with **putConstUser** to show how the sky looks from the surface of this asteroid or comet if you have set it as the central object or which part of this object you can see from the set central body.

If you just want to show the position in the sky of an asteroid or comet you can simply use **getOrbitPosition** after storing the orbit elements without the need to include this object directly as a User Defined Object.

```

void SolarSystem::getSun (double& ra, double& decl)
void SolarSystem::getMoon (double& ra, double& decl)
void SolarSystem::getMercury (double& ra, double& decl)
void SolarSystem::getVenus (double& ra, double& decl)
void SolarSystem::getEarth (double& ra, double& decl)
void SolarSystem::getMars (double& ra, double& decl)
void SolarSystem::getJupiter (double& ra, double& decl)
void SolarSystem::getSaturn (double& ra, double& decl)
void SolarSystem::getUranus (double& ra, double& decl)
void SolarSystem::getNeptune (double& ra, double& decl)
void SolarSystem::getIo (double& ra, double& decl)
void SolarSystem::getEuropa (double& ra, double& decl)
void SolarSystem::getGanymede (double& ra, double& decl)
void SolarSystem::getCallisto (double& ra, double& decl)
void SolarSystem::getRhea (double& ra, double& decl)
void SolarSystem::getTitan (double& ra, double& decl)
void SolarSystem::getMimas (double& ra, double& decl)
void SolarSystem::getEnceladus (double& ra, double& decl)
void SolarSystem::getDione (double& ra, double& decl)
void SolarSystem::getUser (double& ra, double& decl)

```

Get the right ascension *ra* and declination *decl* of the Sun, Moon, Mercury etc as referred to the center of the currently selected central body (usually the Earth). The epoch of the elements will be J2000.0 unless set otherwise via **setEpoch**.

If the selected central body is equal to the one for which the elements are called *ra* is set to *-100.0* and *decl* to *0*. (Example: you call **getEarth** while the *Earth* is the central body - the default).

The right ascension is given in format *HH.MMSSS* and the declination in *DDD.MMSSS*. To convert these numbers into decimal degrees you can use the auxilliary function

double SolarSystem::DmsDegF (double h)

as in the following example:

```

...
double ra, decl;
SolarSystem plts;

plts.getMercury(ra, decl);

ra = ra * 15.0 * plts.DmsDegF(ra);
decl = decl * DmsDegF(decl);
...

```

NOTE: If the time has been set via **setCurrentMJD()** to system time the moment of the first call of a SolarSystem function will determine this time. All subsequent calls will be likewise with this same time until another call of **setCurrentMJD()** has been done. So if you want to update the calculation you have to call **setCurrentMJD()** again each time before the update. The apparent position of the planets change very slowly over the course of a day so you normally don't need to update the data frequently if at all. The exception being the Moon which moves faster across the sky as seen from the Earth. The lunar coordinates should be updated more often. The same could be true for comets.

NOTE: The elements of the Moon will only be calculated correctly if the Earth is the selected central body.

NOTE: See the special example at the end of this manual for a User Defined Object.

```

void SolarSystem::getPhysSun (double &pdiam, double &pmag)
void SolarSystem::getPhysMercury(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysVenus(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysEarth(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysMars(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysJupiter(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysSaturn(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysUranus(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysNeptune(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysIo(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysEuropa(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysGanymede(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysCallisto(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysRhea(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysTitan(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysMimas(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysEnceladus(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysDione(double &pdiam, double &pmag, double &pphase)
void SolarSystem::getPhysUser(double &pdiam, double &pmag, double &pphase)

```

Calculate the physical ephemerides (apparent diameter *pdiam*, apparent magnitude *pmag* and phase *pphase*) for the Sun, Mercury, etc. as referred to the currently selected central body (default *Earth*). The phase is defined as the ratio of the illuminated portion to the total area of the visible planetary disk. (The phase is missing in case of the *Sun* as it is meaningless here).

The apparent diameter is given in radians. To convert this value into arcsec (as is usually done for planets) multiply by 206264.806247. To convert from radians into arcmin (as is usually done for the Sun and the Moon) multiply by 3437.746771.

If you call the physical ephemerides of the currently selected central body all values will be returned as 0 as these values would be meaningless here.

Note: The magnitude of Saturn does not take into account the position of the rings. The magnitude of the Earth is only approximate and would need refining for a more accurate value.

double SolarSystem::getDiamMoon ()

Returns the apparent diameter of the Moon (as seen from the center of the Earth) in radians.
If the Earth is NOT the selected central body 0 will be returned.

void SolarSystem::getLunarPhase (double &phase, double &ildisk, double &amag)

Calculates the *phase*, the value of the illuminated disk *ildisk* and the apparent magnitude *amag* of the Moon as referred to the center of the Earth.

If the Earth is NOT the selected central body these values will be set to 0.

Here the *illuminated disk* is the fraction of the visible disk which is illuminated (1 for Full Moon, 0.5 for Quarter Moon) while *phase* is 0 for New Moon, 0.25 for First Quarter, 0.5 for Full Moon and 0.75 for Last Quarter. (So the *illuminated disk* corresponds to the value returned for *phase* for the physical ephemerides of the planets - a little confusing, sorry!)

void SolarSystem::getLunarLibration (double &lblon, double &lblat, double &termt)

Calculates the libration in longitude *lblon*, in latitude *lblat* as well as the selenographic longitude *termt* of the terminator of the Moon with respect to the center of the Earth. These values are given in decimal degrees.

If the Earth is NOT the selected central body these values will be set to 0.

Vec3 SolarSystem::getPlanetocentric (double ra, double decl)

Returns a vector which points from the center of the selected central body into to direction of the sky given by the right ascension *ra* (in *HH.MMSS*) and declination *decl* (in *DDD.MMSS*). The z-component of this vector (*v[2]*) points toward the north pole along the rotation axis of the planet. The x-component (*v[0]*) points toward the null meridian and the y-component (*v[1]*) completes the rectangular system.

If you have to convert right ascension and declination from decimal degrees into the *HH.MMSS* or *DDD.MMSS* format you can use the auxilliary function:

double SolarSystem::DegFDms (double h)

```
ra = DegFDms (ra / 15.0);  
decl = DegFDms(decl);
```

void SolarSystem::getPlanetographic (double ra, double decl, double &lng, double &lat)

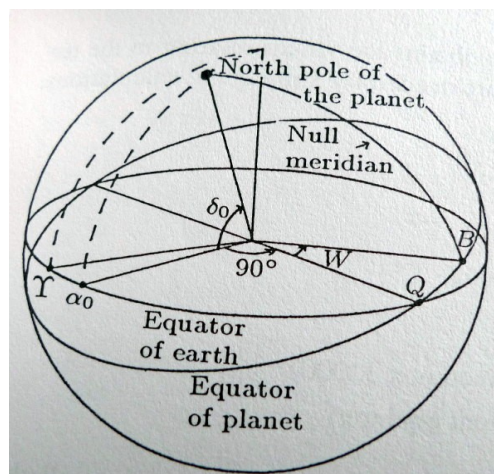
Return the planetographic longitude *lng* and latitude *lat* (in decimal degrees) where the point in the sky given by its right ascension *ra* (in *HH.MMSSS*) and declination *decl* (in *DDD.MMSSS*) is in the zenith.

The latitude is corrected for the flattening of the planet. The longitude is normalized to -180° to $+180^\circ$. East is positive West is negative. Note that this is different from the convention used for planetary longitudes in astronomy but is usually used for cartography and for Earth and Moon coordinates.

If you want to plot the sky around a globe in 3D fashion you should use the planetocentric vector rather than the planetographic coordinates unless you reverse the flattening of the latitude correspondingly.

void SolarSystem::getSkyRotAngles (double &raz1, double &rax, double &raz2)

Return the three rotation angles *raz1*, *rax* and *raz2* (in radians) which are needed for transformation from the (equatorial) celestial sphere into the planetocentric coordinates. To do so you first rotate around the z-axis by *raz1*, then around the x-axis by *rax* and finally around the new z-axis by *raz2*. These angles are defined as follows (also see diagram): The rotation axis of the planet points to the sky position given by Right Ascension α_0 and Declination δ_0 . *raz1* is defined as $\alpha_0 + 90^\circ$, *rax* is $90^\circ - \delta_0$ and *raz2* is equal to the angle *W* which gives the position of the null meridian of the planet.



void SolarSystem::putOrbitElements (double t0, double pdist, double ecc, double ran, double aper, double inc, double eclep)

Stores the heliocentric ecliptic orbit elements of a comet or asteroid in case of hyperbolic or highly elliptical orbits.

t0 is the MJD time of the perihelion.

pdist is the perihelion distance in *AU*.

ecc is the eccentricity.

ran is the right ascension, *aper* the argument of perihelion and *inc* the inclination (all in degrees).

eclep is the epoch of these angles (typically 2000.0).

The orbit elements of comets or asteroids can be obtained in various formats from the Minor Planet Center at

cfa-www.harvard.edu/iau/mpc.html

Note that only one set of elements will be stored. If you call **putOrbitElements** repeatedly the old elements will be overwritten. So you have to do your position calculations of the old elements before storing new elements.

void SolarSystem::putEllipticElements (double t0, double a, double m0, double ecc, double ran, double aper, double inc, double eclep)

Stores the heliocentric ecliptic orbit elements of a comet or asteroid or spacecraft in case of normal elliptical orbits.

t0 is the MJD time of the epoch of the elements.

a is the semimajor axis in *AU*.

m0 is the mean anomaly at epoch (in degrees)

ecc is the eccentricity.

ran is the right ascension, *aper* the argument of perihelion and *inc* the inclination (all in degrees).

eclep is the epoch of these angles (typically 2000.0).

If instead of the semimajor axis *a* the perihelion distance is given (which happens often for highly elliptical orbits) use **putOrbitElements** instead.

void SolarSystem::getOrbitPosition (double& ra, double& decl)

Get the right ascension *ra* and declination *decl* of the object for which the (Keplarian) orbit elements have been stored via **putOrbitElements** or **putEllipticElements** as referred to the center of the currently selected central body (usually the Earth). The epoch of the elements will be J2000.0 unless set otherwise via **setEpoch**.

If no orbit elements have been stored yet via **putOrbitElements** or **putEllipticElements** *ra* is set to -100.0 and *decl* to 0.

The right ascension is given in format *HH.MMSSS* and the declination in *DDD.MMSSS*.

To convert these numbers into decimal degrees you can use the auxilliary function

double SolarSystem::DmsDegF (double h)

.

double SolarSystem::getDistance()

Returns the distance of the object for which the (Keplarian) orbit elements have been stored via **putOrbitElements** or **putEllipticElements** as referred to the center of the currently selected central body (usually the Earth). If no orbit elements have been stored yet 0 will be returned.

double SolarSystem::getCometMag(double g, double k)

Returns the apparent visual magnitude of a comet for which the (Keplarian) orbit elements have been stored via **putOrbitElements** or **putEllipticElements** as referred to the center of the currently selected central body (usually the Earth). The parameters *g* and *k* are the absolute magnitude and the comet constant which differ from comet to comet and are usually given as part of the orbit elements of the Minor Planet Center. (Typical values are 6.0 and 4.0 respectively)

double SolarSystem::getAsteroidMag(double h, double g)

Returns the apparent visual magnitude of an asteroid for which the (Keplarian) orbit elements have been stored via **putOrbitElements** or **putEllipticElements** as referred to the center of the currently selected central body (usually the Earth). The parameters *h* and *g* are the absolute magnitude and the slope parameter which are usually given as part of the orbit elements of the Minor Planet Center.

void SolarSystem::putOrbitUser (double t0, double pdist, double ecc, double ran, double aper, double inc, double eclep)

Stores the heliocentric ecliptic orbit elements for a User Defined Object in case of hyperbolic or highly elliptical orbits.

t0 is the MJD time of the perihelion.

pdist is the perihelion distance in AU.

ecc is the eccentricity.

ran is the right ascension, *aper* the argument of perihelion and *inc* the inclination (all in degrees).

eclep is the epoch of these angles (typically 2000.0).

void SolarSystem::putEllipticUser (double t0, double a, double m0, double ecc, double ran, double aper, double inc, double eclep)

Stores the heliocentric ecliptic orbit elements of a User Defined Object in case of normal elliptical orbits.

t0 is the MJD time of the epoch of the elements.

a is the semimajor axis in AU.

m0 is the mean anomaly at epoch (in degrees)

ecc is the eccentricity.

ran is the right ascension, *aper* the argument of perihelion and *inc* the inclination (all in degrees).

eclep is the epoch of these angles (typically 2000.0).

If instead of the semimajor axis a the perihelion distance is given (which happens often for highly elliptical orbits use **putOrbitUser** instead.

void SolarSystem::putConstUser(double j2, double r0, double flat, double axl0, double axb0, double axb1, double w, double wd, double gm)

Stores the physical constants of a User Defined Object (*j2* and *gm* are not currently used).

j2 is the J2 gravitation term

r0 is the equatorial radius in km

flat is the flattening factor

axl0 is the R.A. of the rotation axis (J2000)

axl1 is the delta of axl0 per Julian Centuries

axb0 is the Declination of the rotation axis (J2000)

axb1 is the delta of axb1 in Julian Centuries

w is the location of the prime meridian at 2000.0

wd is the daily variation of *w*

gm is the gravitational constant (m^3 / s^2)

EXAMPLE:

Calculate the position and magnitude of Vesta and of the comet C/2012 S1 ISON for 3-OCT-2013, 0 UTC. with respect to the Earth.

The following elements were obtained from the Minor Planet Center:

Vesta:

Inclination : 7.13407
Mean Anomaly : 307.801
Semimajor Axis; 2.3619124
Eccentricity: 0.0886225
Right Ascension: 103.9097
Argument of Perihelion: 149.8373
Epoch of elements: 23.0-JUL-2010
Epoch of ecliptic and equinox: 2000.0

mag parameters H 3.20, G: 0.32

C/2012 S1 (ISON)

Time of perihelion: 11/28.7744/2013
perihelion distance: 0.012443
Inclination: 62.3960
Eccentricity: 1.000003
Right Ascension: 295.6531
Argument of Perihelion: 345.5647
Epoch of ecliptic and equinox: 2000.0

mag parameters g, k: 7.5, 3.2

...

```
double t0, ra_vesta, decl_vesta, mag_vesta;  
double ra_ISON, decl_ISON, mag_ISON;  
SolarSystem plts;
```

```
plts.setCurrentMJD (2013, 10, 03, 0, 0, 0); // time for which to calculate position
```

```
t0 = plts.getMJD(2010, 7, 23, 0, 0, 0); // epoch for Vesta elements  
plts.putEllipticElements(t0, 2.36191, 307.801, 0.0886225, 103.91, 149.837, 7.13407, 2000.0);
```

```
plts.getOrbitPosition(ra_vesta, decl_vesta);  
mag_vesta = plts.getAsteroidMag(3.2, 0.32);
```

```
t0 = plts.getMJD(2013, 11, 28, 18, 35, 8.16); // time of perihelion passage  
plts.putOrbitElements (t, 0.012443, 1.000003, 295.6531, 345.5647, 62.3960, 2000.0);
```

```
plts.getOrbitPosition(ra_ISON, decl_ISON);  
mag_ISON = plts.getCometMag(7.5, 3.2);
```

EXAMPLE:

Calculate the position in the sky and the apparent magnitude of Earth as seen from the minor planet Vesta for 9-JAN-2015, 0 UTC. Also calculate the planetographic coordinates of Vesta, where the Earth is in the zenith.

This is an example of a User-Defined Object.

We use the same orbital elements for Vesta as in the previous example:

Inclination : 7.13407
Mean Anomaly : 307.801
Semimajor Axis; 2.3619124
Eccentricity: 0.0886225
Right Ascension: 103.9097
Argument of Perihelion: 149.8373
Epoch of elements: 23.0-JUL-2010
Epoch of ecliptic and equinox: 2000.0

In addition we need the following physical elements for Vesta:

J2 gravitation term: 0.127832
Equatorial radius: 284.5 km
Flattening factor: 0.195
R.A. of the rotation axis (J2000): 305.8°
Delta of the rotation axis R.A. ax10 per Julian Centuries: 0 (ignored in this example)
Declination of the rotation axis (J2000): 41.4°
Delta of rotation axis declination in Julian Centuries: 0 (ignored in this example)
Location of the prime meridian at 2000.0: 292°
Daily variation of prime meridian: 1617.332776°
Gravitational constant (m^3 / s^2): 1.8297E+10

At present the Solar System software does not make use of the parameters for the gravitational term and the gravitation constant so these could be set to 0 in this example.

...

```
double t0, double ra_earth, decl_earth, mag, diam, phase, lng, lat;  
SolarSystem plts;
```

```
plts.setCurrentMJD (2015, 1, 9, 0, 0, 0); // time for which to calculate position
```

```
t0 = plts.getMJD(2010, 7, 23, 0, 0, 0); // epoch for Vesta elements  
plts.putEllipticUser(t0, 2.36191, 307.801, 0.0886225, 103.91, 149.837, 7.13407, 2000.0);  
plts.putConstUser(0.127832, 284.5, 0.195, 305.8, 0, 41.4, 0, 292.0, 1617.332776, 1.8297E10);
```

```
plts.setCentralBody ("User"); // Do the calculation as viewed from Vesta
```

```
plts.getEarth(ra_earth, decl_earth); // sky position of the Earth as viewed from Vesta  
plts.getPhysEarth(diam, mag, phase); // mag is the apparent magnitude as viewed from Vesta
```

```
plts.getPlanetographic(ra_earth, decl_earth, lng, lat);  
// lng and lat are the planetographic sub-Earth coordinates on Vesta.
```