Kepler Orbit Simulation Toolkit

programming reference

Last update: november 10, 2008
Author: CJP

About KOST

KOST is the Kepler Orbit Simulation Toolkit. It is a small programming library, intended to make life easier for programmers who have to deal with Keplerian two-body models, which are often useful for describing and predicting the motion of spacecrafts and celestial bodies.

KOST is available for the C programming language, and it contains some utility functions for use in C++. At the time of writing, no bindings are available for other programming languages.

KOST is available as free software, released under the GNU LGPL (Library General Public License). This gives you a lot more freedom than typical commercial licenses. For instance, you have access to the source code, and, under certain conditions, you are allowed to make modifications, and distribute the original or modified versions of KOST. For details, you should read the LGPL.

Some theory about two-body motion

Gravity

The motion of spacecrafts and celestial bodies is influenced by many factors, such as radiation pressure and the use of rocket engines in the case of spacecrafts, but the most influential factor by far is gravity. The equation that describes the gravity force between bodies is already known for a long time, as it was already described by Isaac Newton. In a simplified model, where gravitation is the only force, the motion of a number of bodies is described exactly by this law of gravitation together with the initial positions and velocities of the bodies.

Two-body systems

The problem is now that the law of gravitation gives a force as function of position, while we usually want to have position as function of time. To find the position as a function of time, one needs to do some mathematics. The type of mathematical problem we're dealing with here is known as a "differential equation". Solving differential equations is often difficult, and it is sometimes impossible to find an exact solution. In our case, it turns out that an exact solution can only be found if there are no more than two bodies[1]. Because of the importance of having an exact solution, the field of celestial mechanics has well-known terms like "two-body problem" (a problem that contains only two bodies, and can therefore be solved) and "two-body approximation" (an approximate solution to a problem with more than two bodies, which was acquired by only considering the two most important bodies, and ignoring all other bodies).

1 The case where there is only one body can also be solved, but that is not an interesting case, as there is no gravity at all. In the case of zero bodies, there is nothing to be solved. The number of bodies can not be negative of course.

The solutions of the two-body problem have some interesting properties. The *orbit* of one of the bodies around the other one has the shape of a conical section: a circle, an ellipse, a parabola or a hyperbola. The shape of an orbit is described by its *eccentricity* e: $e = 0$ gives a circle, $0 \leq e < 1$ gives an ellipse, $e = 1$ gives a parabola and $e > 1$ gives a hyperbola. Note that circles and ellipses are closed shapes: an object in a circular or elliptical orbit will continue repeating the same trajectory over and over again. Parabola and hyperbola, on the other hand, are 'open'. An object in such an orbit will sooner or later depart from the other object, to never return again. This difference is also related to the energy in the system: when an object's velocity exceeds its *escape velocity*, it has enough kinetic energy to escape the other object's gravity field: it has a hyperbolic orbit. The case where the velocity exactly equals escape velocity corresponds to a parabolic orbit.

Numerical simulation

In reality, there are always more than two bodies. When looking at the motion of a spacecraft in Low Earth Orbit, there may be two important bodies (Earth and the spacecraft), but there are many other bodies, like the sun, the moon and Jupiter, whose gravity will disturb the two-body motion. Often, these disturbances are so small that they can be ignored as long as no high accurracy is required. In cases where these disturbances become relevant, the only way to predict the motion accurately is through numerical simulation. In numerical simulation, the differential equations are solved in a step-by-step appoach by a computer program. Typically, this is done by gradually increasing the time by taking small *time steps*, starting at the initial situation. In each time step, the next state of all bodies is calculated from the previous state, by using some simple approximations. These approximations make it easy to do the time step, but they are only valid for small time steps. As a result, numerical simulation is only accurate when the time steps are small. Small time steps means you need a lot of them, and many time steps means you need to spend a lot of computing power on the simulation.

Like the two-body approximation, numerical simulation can only give an approximate prediction of the motion in more-than-two body systems. Both approaches have advantages and disadvantages. In situations where there are large disturbances from the two-body model, numerical simulation is superior. However, when doing very long-term calculations, the more naïve numerical simulations tend to build up increasingly large errors. Besides accurracy arguments, numerical simulation tends to be computationally expensive, and a two-body approximation tends to give a more clear and simple view on what is going on.

KOST

KOST focuses on using the two-body approximation. The reason for this is that, while it is actually quite easy to make a simple numerical simulation program, the equations describing the exact two-body solution are quite complicated. Therefore, it is more useful to have a toolkit do that for you.

How to use KOST

Currently, there is no complicated build system supporting KOST. You can simply add the KOST source files to your project, and make sure they are compiled and linked together with the rest of your program. In your own source files, you can simply include kost.h, which gives you access to all KOST functions. Please make sure you know the licensing conditions of KOST before you release any software that uses KOST. In particular, if you statically link with KOST, as described here, you are

limited in your choice of licensing terms for your software.

Configuring KOST

In the header file kost_types.h, you can change which data types are used by KOST. KOST has its own data structures for various things, like vectors or orbital elements, but if you want it to use different data structures, you can change that here. This is useful, for instance, for better inter-operability with the data types of another part of the application. As long as the data types and structs are interchangeable on source code level (e.g. structs contain all the required elements), KOST should recompile without problems.

The structs of KOST are deliberately designed to be as compatible as possible with the structs of the Orbiter space flight simulator.

Header files

The following header files are available. If you don't want to worry about header files, just inlcude kost.h, which will simply include everything.

- **kost.h**: Includes all other KOST headers. Only includes kostmm.h when compiled in C++.
- **kost_constants.h**: Some mathematical and physical constants. All are in SI units.
- **kost_elements.h**: Functions for dealing with orbital elements.
- **kost_linalg.h**: Linear algebra functions.
- **kost_math.h**: Includes math.h, and adds some extra math functions.
- **kost_shape.h**: Functions for getting the shape of an orbit.
- **kost_types.h**: Type definitions.
- **kostmm.h**: C++ utility functions, mostly for linear algebra.

Constants

kost_constants.h

kost_constants.h includes math.h, so all constants in math.h are also defined by including kost_constants.h. Besides this, the following constants are defined. All physical constants are in SI units.

- **M_PI**: π = 3.14159265....; you can find it on any calculator
- **M_TWOPI**: 2* π
- **ASTRONOMICAL_UNIT**: The size of an astronomical unit (AU)
- **GRAVITATIONAL_CONSTANT**: The gravitational constant G

Types

kost_types.h

The following types are used in the KOST API:

```
typedef double kostReal;
```

The type used for real numbers. Typically, float or double is used. KOST uses the double type of math.h functions internally, so a higher accurracy than double is not guaranteed.

```
typedef struct
{

      kostReal x, y, z;

} kostVector3;
```

Threedimensional vector type.

```
typedef struct
{

      kostReal
            m11, m12, m13,
            m21, m22, m23,
            m31, m32, m33;

} kostMatrix3;
```

3x3 matrix type. Such matrices are often used for transforming 3D vectors.

```
typedef struct
{
      kostReal a;       /*Semi-major axis*/
      kostReal e;       /*Eccentricity*/
      kostReal i;       /*Inclination*/
      kostReal theta;   /*Longitude of ascending node*/
      kostReal omegab; /*Longitude of periapsis*/
      kostReal L;       /*Mean longitude at epoch*/
} kostElements;
```

Keplerian orbital elements.

```
typedef struct
{
	/*Same as ORBITPARAM*/
	kostReal SMi;  /*semi-minor axis*/
	kostReal PeD;  /*periapsis distance*/
	kostReal ApD;  /*apoapsis distance*/
	kostReal MnA;  /*mean anomaly*/
	kostReal TrA;  /*true anomaly*/
	kostReal MnL;  /*mean longitude*/
	kostReal TrL;  /*true longitude*/
	kostReal EcA;  /*eccentric anomaly*/
	kostReal Lec;  /*linear eccentricity*/
	kostReal T;    /*orbit period*/
	kostReal PeT;  /*time to next periapsis passage*/
	kostReal ApT;  /*time to next apoapsis passage*/

	/*Additional*/
	kostReal AgP;  /*argument of periapsis*/
} kostOrbitParam;
```

Additional orbital parameters, which are not present in the kostElements struct.

```
typedef struct
{
	kostVector3 pos;
	kostVector3 vel;
} kostStateVector;
```

State vector of an object, consisting of its position and its velocity.

```
typedef struct
{
	kostVector3 pe, ap, dn, an;

	kostVector3 *points;
	unsigned int numPoints;

} kostOrbitShape;
```

Shape of an orbit. **pe**, **ap**, **dn** and **an** are the positions of the periapsis, apoapsis, descending node and ascending node. The array **points** contains a number of points that are on the orbit. **numPoints** is the number of elements in **points**.

Math functions

kost_math.h

kost_math.h includes math.h, so all functions in math.h are also defined by including kost_math.h. Besides this, the following functions are defined:

```
double acosh(double x);
double asinh(double x);
```

Inverse hyperbolic cosine and inverse hyperbolic sine functions. They do exist in the C89 standard, but don't seem to be present in VC6's math.h.

Linear Algebra functions

kost_linalg.h

The following linear algebra functions are available. Note the suffix in the function name: this relates to the types of the arguments (v = vector, m = matrix, r = real number). This is necessary for having many different 'overloads' of the same operator in C.

```
kostVector3 kostConstructv(kostReal x, kostReal y, kostReal z);
```

Constructs a vector, by filling in the cartesian coordinates x, y and z.

```
kostVector3 kostAddvv(const kostVector3 *v1, const kostVector3 *v2);
kostVector3 kostSubvv(const kostVector3 *v1, const kostVector3 *v2);
```

Vector addition and subtraction.

```
kostVector3 kostMulrv(kostReal r, const kostVector3 *v);
```

Multiplication of a vector and a scalar.

```
kostReal    kostDotProductvv(const kostVector3 *v1, const kostVector3 *v2);
kostVector3 kostCrossProductvv(const kostVector3 *v1, const kostVector3 *v2);
```

Dot product (v1 · v2) and cross product (v1 × v2) of two vectors.

```
kostReal    kostAbsv(const kostVector3 *v);
kostReal    kostAbs2v(const kostVector3 *v);
```

kostAbsv returns the absolute value (the length) of the vector.

kostAbs2v returns the square of the length. **kostAbs2v** is slightly faster than **kostAbsv**, so if you need the square, or if it makes no difference, use **kostAbs2v**.

```
kostVector3 kostNormalv(const kostVector3 *v);
```

Returns a normalized vector (length = 1), pointing in the same direction as **v**.

```
void kostMakeUnitm(kostMatrix3 *m);
```

Fills the elements of **m** with values that make it the unit matrix. A unit matrix has values 1 on the diagonal, and 0 in all other elements. The interesting property of the unit matrix is that multiplying a vector or a matrix with it doesn't change that vector or matrix. So, when a matrix is used for a transformation, the unit matrix is the “don't change anything” transformation.

```
void kostMakeXRotm(kostMatrix3 *m, kostReal angle);
void kostMakeYRotm(kostMatrix3 *m, kostReal angle);
void kostMakeZRotm(kostMatrix3 *m, kostReal angle);
```

Fills the elements of **m** with values that make it a rotation matrix, for rotating **angle** radians counterclockwise around one of the coordinate axes. A composed rotation can be made by multiplying several matrices with each other.

```
void kostMakeTransposem(kostMatrix3 *m);
```

Turns the matrix **m** into its transpose (swapping rows and columns). For rotation matrices, the transpose is also the inverse: the transpose will transform into the opposite direction.

kostVector3 kostMulmv(const kostMatrix3 *m, const kostVector3 *v);

Matrix-vector-multiplication. This can be used for transforming vectors: for instance, when **m** is a rotation matrix, the return value is a rotated version of **v**.

kostMatrix3 kostMulmm(const kostMatrix3 *m1, const kostMatrix3 *m2);

Matrix-matrix-multiplication. When using matrices for transformations, this makes a composed transformation out of two transformations. Be aware that the order is important here: doing one tranformation first, and then the other, is not always the same as the other way around. So, A*B is not always equal to B*A.

C++ utility functions

kostmm.h

kostVector3 operator+(const kostVector3 &v1, const kostVector3 &v2);
kostVector3 operator-(const kostVector3 &v1, const kostVector3 &v2);

Vector addition and subtraction.

kostVector3 operator*(const kostVector3 &v, kostReal r);

Multiplication of a vector and a scalar.

kostReal dotProduct(const kostVector3 &v1, const kostVector3 &v2);
kostVector3 crossProduct(const kostVector3 &v1, const kostVector3 &v2);

Dot product (v1 · v2) and cross product (v1 × v2) of two vectors.

kostReal abs(const kostVector3 &v);
kostReal abs2(const kostVector3 &v);

Absolute value and square of absolute value. See **kostAbsv** and **kostAbs2v**.

kostMatrix3 operator*(const kostMatrix3 &m1, const kostMatrix3 &m2);
kostVector3 operator*(const kostMatrix3 &m, kostVector3 &v);

Matrix-matrix and matrix-vector multiplication. See **kostMulmm** and **kostMulmv**.

kostVector3 &operator*=(kostVector3 &v, const kostMatrix3 &m);

Matrix-vector multiplication that stores the result in the original vector. Useful when you want compact code, and don't need to keep the original vector.

void makeTranspose(kostMatrix3 &m);

Turns the matrix **m** into its transpose. See **kostMakeTransposem**.

Orbital elements functions

kost_elements.h

These functions help you in conversion between state vectors (position and velocity), and Keplerian orbital elements. Both describe the state of one of the bodies with respect to the other one, but for some things you need one form, and for other things you need the other form.

The state vector is usually how you get the initial conditions, and it is usually also the form how you would like to have predictions about the future state of the system. However, *making* predictions is actually easier in the orbital elements form. In a true two-body system, most of the orbital elements don't change in time, and the only one that does (the mean longitude at epoch), does so in a relatively simple way. Also, the orbital elements that don't change give some useful information about the kind of orbit the body is in.

One of the arguments of the following functions is **mu**. **mu** is a measure for the amount of gravity between the two bodies. You can calculate its value as mu = GRAVITATIONAL_CONSTANT * (m1 + m2), where m1 and m2 are the masses of the two bodies.

These conversions assume a right-handed coordinate system, with the XY-plane being the reference plane, and the X-axis being the reference direction.

```
void kostElements2StateVector(
  kostReal mu,
  const kostElements *elements,
  kostStateVector *state);
```

Converts orbital elements to a state vector. At the time of writing, this function is **Not Yet Implemented**

```
void kostStateVector2Elements(
  kostReal mu,
  const kostStateVector *state,
  kostElements *elements,
  kostOrbitParam *params);
```

Converts a state vector to orbital elements. As a bonus, this function also gives additional information about the orbit in **params**. You can optionally set either **elements** or **params** to NULL if you don't need it. Note that some elements / parameters don't make sense for hyperbolic orbits: their value will be undefined.

Orbital shape functions

kost_shape.h

These functions deal with the shape of the orbit. Their typical purpose is for visualization of an orbit.

These functions assume a right-handed coordinate system, with the XY-plane being the reference plane of the orbital elements, and the X-axis being the reference direction.

`void kostElements2Shape(const kostElements *elements, kostOrbitShape *shape);`

Determines the shape, based on the orbital elements.

shape->points should already be allocated before this function is called. It should cointain at least **shape->numPoints** elements, as this function will write this number of points into the array.

After this function is called, **shape->points** contains a polyline approximation of the shape of the orbit. In the case of parabolic and hyperbolic orbits, the actual shape continues to infinity. The first and last point of the polyline are not at infinity. Their actual length depends on the number of points: with more points on an otherwise equal orbit, they will be longer.