User Manual 3.4.1 SpacecraftState

De Wiki
Aller à : navigation, rechercher
User Manual 3.4.1 SpacecraftState

Sommaire

Introduction

Scope

This section describes the SpacecraftState object provided by the Orekit library.

Javadoc

The object SpacecraftState is available in the package `org.orekit.propagation` of OREKIT.

Links

Here is only described the SpacecraftState structure. Please refer to propagation chapter.

Useful Documents

None as of now.

Overview

The SpacecraftState is composed of :
- an orbit
- an attitude for forces computation
- an attitude for events computation
- a map of additional states (including mass states added from MassProvider).

Two attitudes are stored in order to apply (if needed) a different treatment to each attitude.

Features Description

One orbit

The SpacecraftState is composed of one Orbit. It is possible to simply declare a SpacecraftState with an orbit : `final SpacecraftState state = new SpacecraftState(orbit);`

The orbit could be updated using the following method : `final SpacecraftState newState = state.updateOrbit(newOrbit);` The attitude and additional states will remain the same and a new SpacecraftState will be created.

Two attitudes

The user can use one single Attitude or two different Attitude objects : one for forces computation and one for events computation. The following constructor can be used :

```
final SpacecraftState state = new SpacecraftState(orbit, attitudeForces,
attitudeEvents);
```

It is possible to get the attitude for forces or events computation using the following methods:

```
final Attitude attForces = state.getAttitudeForces();
final Attitude attEvents = state.getAttitudeEvents();
```

The user can deals with a single attitude in the SpacecraftState using the following constructor:

```
final SpacecraftState state = new SpacecraftState(orbit, attitude);
final Attitude att = state.getAttitude();
```

If the following constructor is used, both attitudes are set to null value: `final SpacecraftState state = new SpacecraftState(orbit);` Then calling `getAttitude` or `getAttitudeForces` or `getAttitudeEvents` will return null attitude.

Additional states

The additional states are stored in the SpacecraftState using a Map with the additional states names as keys. The additional states are in the type `double[]`. The additional states map could be given directly to the constructor as follow :

```
Map<String, double[]> addStates = new HashMap<String, double[]>();
addStates.put("name", new double[]{1.0});
SpacecraftState state = new SpacecraftState(orbit, attitudeForces,
attitudeEvents, addStates);
```

It is possible to add an additional state to a SpacecraftState using ***addAdditionalState***:

```
state2 = state.addAdditionalState("name2", new double[]{0.1, 0.1});
```

Note : ***addAdditionalState*** returns a new SpacecraftState with the added additional state. It is necessary to store the returned object. The additional state is not added to the current state.

Mass

A MassProvider can be provided to the SpacecraftState. In that case, the mass information are automaticaly stored as additional states. Be careful, the mass values can never be negative:

```
final SpacecraftState state = new SpacecraftState(orbit, massProvider);
final SpacecraftState state = new SpacecraftState(orbit, attitudeForces,
attitudeEvents, massProvider, addStates);
```

The mass provider can be added to the SpacecraftState after its initilisation with the method `addMassProvider` :

```
final SpacecraftState state = new SpacecraftState(orbit);
final SpacecraftState newState = state.addMassProvider(massProvider);
```

The mass parts from MassProvider are added to additional states map with the key : "MASS_<partName>".

The total mass of the SpacecraftState could not be obtained.

It is possible to obtain the mass of a given part : `state.getMass("part1");`

and to update the mass of a given part : `state.updateMass("part1", 1000.0);`

State vector

The SpacecraftState object could be created from a state vector : `final SpacecraftState state = new SpacecraftState(stateVector, OrbitType.CARTESIAN, PositionAngle.MEAN, date, mu, frame, addStatesInfo, attProviderForces, attProviderEvents);`

To build the SpacecraftState, it is necessary to know the size and the index of all additional states in the state vector. This information could be obtained from an old state using the following method : `final Map<String, AdditionalStateInfo> addStatesInfos = state.getAdditionalStatesInfos();`

The state vector could be obtained from a SpacecraftState :

```
final double[] stateVector = new double[]{};
state.mapStateToArray(OrbitType.CARTESIAN, PositionAngle.MEAN, stateVector);
```

Transform

The SpacecraftState class contains methods to compute the following transformations :
- `toTransform()` or `toTransformForces()` : Transform from orbit/attitude reference frame to spacecraft frame (attitude used for forces computation which is the default attitude).
- `toTransformEvents()` : Transform from orbit/attitude reference frame to spacecraft frame attitude used for events computation (same as toTransform if there is no specific attitude for Events).
- `toTransform(Frame)` or `toTransformForces(Frame)` : Transform from specified Frame to spacecraft frame (attitude used for forces computation which is the default attitude).
- `toTransformEvents(Frame)` : Transform from specified Frame to spacecraft frame attitude used for events computation (same as toTransform(Frame) if there is no attitude specific for Events). - `toTransform(LOFType)` : Transform from orbit/attitude reference frame to local orbital frame (LOFType).
- `toTransform(Frame, LOFType)` : Transform from specified Frame to local orbital frame (LOFType).

Here after is presented the computation of a station position in spacecraft frame:

```
// Station position defined in GCRF
final Vector3D station_InGCRF = new
Vector3D(Constants.EGM96_EARTH_EQUATORIAL_RADIUS, 0, 0);
final Frame gcrf = FramesFactory.getGCRF();
// Compute transform from GCRF to spacecraft frame
final Transform transform = state.toTransform(gcrf);
// Station position in spacecraft frame
final Vector3D station_InSpacecraftFrame =
transform.transformVector(station_InGCRF);
```

Here after is presented the computation of the Sun direction in spacecraft frame.

```
// Compute transform from orbit/attitude reference frame to local orbital
frame TNW
final Transform transform = state.toTransform(LOFType.TNW);
// Position of Sun in local orbital frame TNW
final Vector3D pos =
transform.transformPosition(sun.getPVCoordinates().getPosition());
```

If an Assembly is used, this is not necessary to use these methods because the conversion methods are included in Assembly functionalities (see dedicated User Manual).

Getting Started

Using Transform

Here after is presented the computation of a station position in spacecraft frame:

```
// Station position defined in GCRF
final Vector3D station_InGCRF = new
Vector3D(Constants.EGM96_EARTH_EQUATORIAL_RADIUS, 0, 0);
final Frame gcrf = FramesFactory.getGCRF();
// Compute transform from GCRF to spacecraft frame
final Transform transform = state.toTransform(gcrf);
// Station position in spacecraft frame
final Vector3D station_InSpacecraftFrame =
transform.transformVector(station_InGCRF);
```

Here after is presented the computation of the Sun direction in spacecraft frame.

```
// Compute transform from orbit/attitude reference frame to local orbital
frame TNW
final Transform transform = state.toTransform(LOFType.TNW);
// Position of Sun in local orbital frame TNW
final Vector3D pos =
transform.transformPosition(sun.getPVCoordinates().getPosition());
```

If an Assembly is used, this is not necessary to use these methods because the conversion methods are included in Assembly functionalities (see dedicated User Manual).

Contents

Interfaces

Interface	Summary	Javadoc
MassProvider	Interface providing the mass for spacecraft models.	...

Classes

Class	Summary	Javadoc
SpacecraftState	This class is the representation of a complete state holding orbit, attitude for forces and for events computation and additional states at a given date.	...
Attitude	Object representing the attitude of the spacecraft for a specific date and in a specific frame.	...

- Main differences between V4.9 and V4.8
- Main differences between V4.8 and V4.7
- Main differences between V4.7 and V4.6.1
- Main differences between V4.6.1 and V4.5.1
- Main differences between V4.5.1 and V4.4
- Main differences between V4.4 and V4.3
- Main differences between V4.3 and V4.2
- Main differences between V4.2 and V4.1.1
- Main differences between V4.1.1 and V4.1
- Main differences between V4.1 and V4.0
- Main differences between V4.0 and V3.4.1

User Manual

- User Manual 4.15
- User Manual 4.14
- User Manual 4.13
- User Manual 4.12
- User Manual 4.11
- User Manual 4.10
- User Manual 4.9
- User Manual 4.8
- User Manual 4.7
- User Manual 4.6.1
- User Manual 4.5.1
- User Manual 4.4
- User Manual 4.3
- User Manual 4.2
- User Manual 4.1
- User Manual 4.0
- User Manual 3.4.1
- User Manual 3.3

Tutorials

- Tutorials 4.15
- Tutorials 4.14
- Tutorials 4.13.5
- Tutorials 4.12.1
- Tutorials 4.8.1
- Tutorials 4.5.1
- Tutorials 4.4
- Tutorials 4.1
- Tutorials 4.0

Links

- CNES freeware server