Doc

Norah Jones

2024-04-07

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

```julia
using QuartoDocumenter, AstrodynamicalModels
QuartoDocumenter.autodoc(AstrodynamicalModels)
```

```
Precompiling QuartoDocumenter
    QuartoDocumenter
  1 dependency successfully precompiled in 1 seconds. 27 already precompiled.
```

Provides astrodynamical models as `AstrodynamicalModels.ODESystems`. Check out the `ModelingToolkit` docs to learn how to use these systems for orbit propagation with `DifferentialEquations`, or see `GeneralAstrodynamics` for some convenient orbit propagation wrappers.

Extended help

0.0.0.0.1 * License

0.0.0.0.2 * Exports

- `AttitudeFunction`
- `AttitudeParameters`
- `AttitudeState`
- `AttitudeSystem`
- `CR3BFunction`
- `CR3BOrbit`
- `CR3BParameters`
- `CR3BState`
- `CR3BSystem`
- `CartesianOrbit`
- `CartesianState`
- `KeplerianOrbit`
- `KeplerianParameters`
- `KeplerianState`
- `NBFunction`
- `NBSystem`
- `Orbit`
- `OrbitalElements`
- `PlanarEntryFunction`
- `PlanarEntryParameters`
- `PlanarEntryState`
- `PlanarEntrySystem`
- `R2BFunction`
- `R2BOrbit`
- `R2BParameters`
- `R2BState`
- `R2BSystem`
- `dynamics`
- `parameters`
- `state`
- `system`

0.0.0.0.3 * Imports

- `Base`
- `Core`
- `DocStringExtensions`

- `LinearAlgebra`
- `Memoize`
- `ModelingToolkit`
- `SciMLBase`
- `StaticArrays`
- `Symbolics`

```
AttitudeFunction(; stm, name, kwargs...)
```

Returns an `ODEFunction` for spacecraft attitude dynamics.

Extended Help

0.0.0.0.0.1 * Usage

The `stm` and `name` keyword arguments are passed to `Attitude`. All other keyword arguments are passed directly to `SciMLBase.ODEFunction`.

```
f = AttitudeFunction()
let u = randn(7), p = randn(15), t = NaN # time invariant
    f(u, p, t)
end
```

```
struct AttitudeParameters{F} <: AstrodynamicalModels.AstrodynamicalParameters{F, 15}
```

A parameter vector for attitude dynamics.

```
mutable struct AttitudeState{F} <: AstrodynamicalModels.AstrodynamicalState{F, 7}
```

A mutable state vector for attitude dynamics.

```
AttitudeSystem(; stm, name, defaults, kwargs...)
```

A `ModelingToolkit.ODESystem` for atmospheric entry. Currently, only exponential atmosphere models are provided! The output model is cached with `Memoize.jl`. Planet-specific parameters default to Earth values.
The order of the states follows: `[q , q , q , q ,  ,  , ]`.
The order of the parameters follows: `[]`

Extended Help

This model describes how an object moves through an exponential atmosphere, above a spherical planet.

0.0.0.0.1 * States

1. `q`: scalar-last attitude quaternion
2. : body rates (radians per second)

0.0.0.0.2 * Parameters

1. `J`: inertial matrix
2. `L`: lever arm where input torque is applied
3. `f`: torques on the vehicle body (Newton-meters)

0.0.0.0.2.1 * Usage

```
model = Attitude()
```

```
CR3BFunction(; stm, name, kwargs...)
```

Returns an `ODEFunction` for CR3B dynamics.
The order of the states follows: `[]`.
The order of the parameters follows: `[]`.

Extended Help

0.0.0.0.0.1 * Usage
The `stm`, and `name` keyword arguments are passed to `CR3B`. All other keyword arguments are passed directly to `SciMLBase.ODEFunction`.

```
f = CR3BFunction(; stm=false, jac=true)
let u = randn(6), p = randn(1), t = 0
    f(u, p, t)
end
```

```
struct Orbit{var"#s25"<:CartesianState, var"#s24"<:CR3BParameters} <: AstrodynamicalModels.
```

An `Orbit` which exists within CR3BP dynamics.

```
struct CR3BParameters{F} <: AstrodynamicalModels.AstrodynamicalParameters{F, 1}
```

A paremeter vector for CR3BP dynamics.

```
mutable struct CartesianState{F} <: AstrodynamicalModels.AstrodynamicalState{F, 6}
```

CartesianState

```
CR3BSystem(; stm, name, defaults, kwargs...)
```

A `ModelingToolkit.ODESystem` for the Circular Restricted Three-body Problem.
The order of the states follows: `[x, y, z, ẋ, ẏ, ż]`.
The order of the parameters follows: `[ ]`.

Extended Help

The Circular Restricted Three-body Problem is a simplified dynamical model describing one small body (spacecraft, etc.) and two celestial bodies moving in a circle about their common center of mass. This may seem like an arbitrary simplification, but this assumption holds reasonably well for the Earth-Moon, Sun-Earth, and many other systems in our solar system.

0.0.0.0.0.1 * Usage

```
model = CR3BSystem(; stm=true)
```

```
struct Orbit{var"#s25"<:CartesianState, P<:(AbstractVector)} <: AstrodynamicalModels.Astro
```

An `Orbit` which exists within R2BP dynamics.

```
mutable struct CartesianState{F} <: AstrodynamicalModels.AstrodynamicalState{F, 6}
```

A mutable vector, with labels, for 6DOF Cartesian states.

```
struct Orbit{var"#s25"<:OrbitalElements, var"#s24"<:KeplerianParameters} <: AstrodynamicalM
```

An `Orbit` which exists within Keplerian dynamics.

```
struct KeplerianParameters{F} <: AstrodynamicalModels.AstrodynamicalParameters{F, 1}
```

A parameter vector for Keplerian dynamics.

```
mutable struct OrbitalElements{F} <: AstrodynamicalModels.AstrodynamicalState{F, 6}
```

OrbitalElements

```
NBFunction(N; stm, name, kwargs...)
```

Returns an `ODEFunction` for NBP dynamics. The order of states and parameters in the `ODEFunction` arguments are equivalent to the order of states and parameters for the system produced with `NBP(N)`. As a general rule, the order of the states follows: `[x , y , z , ..., x , y , z , ẋ , ẏ , ż , ..., ẋ , ẏ , ż ]`.

Note

Unlike `R2BP` and `CR3BP`, `jac` is set to `false` by default. The number of states for `NBP` systems can be very large for relatively small numbers of bodies (`N`). Enabling `jac=true` by default would cause unnecessarily long waiting times for this (**memoize?**) function to return for `N  3` or so. If `N=2` and `stm=true`, setting `jac=true` could still result in several minutes of calculations, depending on the computer you're using.

Warning

Be careful about specifying `stm=true` for systems with `N  3`! If state transition matrix dynamics are enabled, you can calculate the total number of system states with `N*6 + (N*6)^2`. Note that this increases exponentially as `N` grows! For `N == 6`, unless you're using parallelization, your computer may run for several hours.

Extended Help

0.0.0.0.0.1 * Usage

The `stm`, and `name` keyword arguments are passed to `NBP`. All other keyword arguments are passed directly to `SciMLBase.ODEFunction`.

```julia
f = NBFunction(3; stm=false, name=:NBP, jac=false, sparse=false)
let u = randn(3*6), p = randn(1 + 3), t = 0
    f(u, p, t)
end
```

```
NBSystem(N; stm, name, defaults, kwargs...)
```

A `ModelingToolkit.ODESystem` for the Newtonian N-body Problem.
The order of the states follows: `[x , y , z , ..., x , y , z , ẋ , ẏ , ż , ..., ẋ , ẏ , ż ]`.
The order of the parameters follows: `[G, m , m , ..., m ]`.

Warning

Be careful about specifying `stm=true` for systems with `N  3`! If state transition matrix dynamics are enabled, you can calculate the total number of system states with `N*6 + (N*6)^2`. Note that this increases exponentially as `N` grows! For `N == 6`, unless you're using parallelization, your computer may run for several hours.

Extended Help

The N-body problem is a model which describes how `N` bodies will move with respect to a common origin. This problem typically involves many bodies which act due to one force: electromagentism, gravity, etc. This model applies most closely to many celestial bodies moving due to gravity. That's about right for a model in a package called `AstrodynamicalModels`!

0.0.0.0.0.1 * Usage

```
# One model for ALL the planets in our solar system
model = NBSystem(9)
```

```
struct Orbit{U<:(AbstractVector), P<:(AbstractVector)} <: AstrodynamicalModels.Astrodynamic
```

A full representation of an orbit, including a numerical state, and the parameters of the system.

```
mutable struct OrbitalElements{F} <: AstrodynamicalModels.AstrodynamicalState{F, 6}
```

A mutable vector, with labels, for 6DOF Keplerian states.

```
PlanarEntryFunction(; name, kwargs...)
```

Returns an `ODEFunction` for Planar Entry dynamics. Results are cached with `Memoize.jl`.
The order of the states follows: `[ , v, r, ]`.
The order of the parameters follows: `[R, P, H, m, A, C, ]`

Extended Help

0.0.0.0.0.1 * Usage
The `name` keyword argument is]passed to `PlanarEntry`. All other keyword arguments are passed directly to `SciMLBase.ODEFunction`.

```
f = PlanarEntryFunction()
let u = randn(4), p = randn(7), t = NaN # time invariant
    f(u, p, t)
end
```

```
struct PlanarEntryParameters{F} <: AstrodynamicalModels.AstrodynamicalParameters{F, 7}
```

A parameter vector for planar entry dynamics.

```
mutable struct PlanarEntryState{F} <: AstrodynamicalModels.AstrodynamicalState{F, 4}
```

A state vector for planar entry dynamics.

```julia
PlanarEntrySystem(; name, defaults, kwargs...)
```

A `ModelingToolkit.ODESystem` for atmospheric entry. Currently, only exponential atmosphere models are provided! The output model is cached with `Memoize.jl`. Planet-specific parameters default to Earth values.
The order of the states follows: `[ , v, r,  ]`.
The order of the parameters follows: `[R, P, H, m, A, C,  ]`

Extended Help

This model describes how an object moves through an exponential atmosphere, above a spherical planet.

0.0.0.0.0.1 * Usage

```julia
model = PlanarEntrySystem()
```

```
R2BFunction(; stm, name, kwargs...)
```

Returns an `ODEFunction` for R2B dynamics.
The order of the states follows: `[x, y, z, ẋ, ẏ, ż]`.
The order of the parameters follows: `[ ]`.

Extended Help

0.0.0.0.0.1 * Usage
The `stm`, and `name` keyword arguments are passed to `R2B`. All other keyword arguments are passed directly to `SciMLBase.ODEFunction`.

```
f = R2BFunction(; stm=false, name=:R2B, jac=true)
let u = randn(6), p = randn(1), t = 0
    f(u, p, t)
end
```

```
struct Orbit{var"#s25"<:CartesianState, var"#s24"<:R2BParameters} <: AstrodynamicalModels.A
```

An `Orbit` which exists within R2BP dynamics.

```
struct R2BParameters{F} <: AstrodynamicalModels.AstrodynamicalParameters{F, 1}
```

A parameter vector for R2BP dynamics.

```
mutable struct CartesianState{F} <: AstrodynamicalModels.AstrodynamicalState{F, 6}
```

CartesianState

```
R2BSystem(; stm, name, defaults, kwargs...)
```

A `ModelingToolkit.ODESystem` for the Restricted Two-body Problem.
The order of the states follows: `[x, y, z, ẋ, ẏ, ż]`.
The order of the parameters follows: `[ ]`.

Extended Help

The Restricted Two-body Problem is a simplified dynamical model describing one small body (spacecraft, etc.) and one celestial body. The gravity of the celestial body exhibits a force on the small body. This model is commonly used as a simplification to descibe our solar systems' planets orbiting our sun, or a spacecraft orbiting Earth.

0.0.0.0.0.1 * Usage

```
model = R2BSystem()
```

```
dynamics(orbit, args; kwargs...)
```

Return the underlying dynamics of the system in the form of a `ModelingToolkit.ODEFunction`.

```
parameters(orbit)
```

Return the parameter vector for an `Orbit`.

```
state(orbit)
```

Return the state vector for an `Orbit`.

```
system(orbit, args; kwargs...)
```

Return the underlying dynamics of the system in the form of a `ModelingToolkit.ODESystem`.

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

2 Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.