Appendix A - Rigid Body Motion in 3D

Mikica B Kocic
The Physics of Virtual Environments, 2012-04-25

Definitions

```
Get[ "Quat.m", Path -> { NotebookDirectory [] } ];
```

Parameters

Frame of reference

The flag **frameOfRef** indicates whether angular velocity ω, angular momentum **L** and moment of inertia tensor $\mathcal{J}$ are given with coordinates either in **inertial** (also called **space** or **world**) frame of reference or in **non-inertial** (also called **body-fixed** or **rotating**) frame of reference.

```
frameOfRef := BodyFixed;
```

```
Inertial  =. (* inertial (or world) frame of reference *)
BodyFixed =. (* non-inertial (or rotating) frame of reference *)
```

Other physical quantities like position, velocity, forces or torque, are always given in inertial frame of reference.

Gyroscopic effects

The flag **gyroEffects** indicates whether to *include* or *ignore* gyroscopic effect when calculating the time derivative of the angular momentum.

```
gyroEffects := Include;
```

```
Include =.  (* include gyroscopic effects *)
Ignore  =. (* don't account gyroscopic effects *)
```

Physical constants

Gravitational acceleration $\mathbf{g}_n$, kg m s^{-2}

```
gn = { 0, 0, -9.81 };
```

Characteristic dimension ℓ of the system, m

```
ℓ = 10;
```

Rigid body parameters

Rigid body is defined by its mass *m*, a graphics complex `{body$v,body$i}` with centroid coordinates of the body vertices in the body-fixed frame of reference, and principal moment of inertia tensor $\mathcal{J}_0$ and its inverse $\mathcal{J}_{0,\text{inv}} == \mathcal{J}_0^{-1}$.
For more info about graphics complex, see: http://reference.wolfram.com/mathematica/ref/GraphicsComplex.html

```
setupBodyShape[ mass_, { width_, height_, depth_ }, type_ ] :=
 Module[
  { faces },

  (* Body mass, kg *)
  m = mass;

  (* Body dimensions, m *)
  { a, b, c } = { width, height, depth };

  (* Get graphic complex of the body *)
  { faces } = PolyhedronData[ type, "Faces"];

  (* Rescale graphic complex according to specified dimensions *)
  body$v = ( # {a, b, c}) & /@ faces[[1]];
  body$i = faces[[2]];

  (* Moment of inertia and its inverse *)
  𝒥0 = m/12 {{b^2 + c^2, 0, 0}, {0, a^2 + c^2, 0}, {0, 0, a^2 + b^2}} // N;
  𝒥0inv = Inverse[ 𝒥0 ];
 ]
```

Now define initial rigid body (a cube having 1 kg mass)

```
setupBodyShape[ 1, { 1, 1, 1}, "Cuboid" ];
```

Initial state variables: position, orientation, linear and angular velocity. All variables, except angular velocity, are given in inertial (world) frame of reference. Angular velocity frame of reference depends on the `frameOfRef` flag.

```
X0 = { 0, 0, 0 };
Q0 = ℚ[ 0, 0, 0, 0 ];
V0 = { 0, 0, 0 };
ω0 = { 0, 0, 0 };
```

Integration parameters

Integration method, either **rk4$stepper** or **semiImplicitEuler$stepper**

```
odeIntegrator := rk4$stepper
```

Time-step length, s

```
h = 0.01;
```

Final time, s

```
tf = 4;
```

- **Runge-Kutta 4th order method** (rk4$stepper)

  Classical implementation of the 4th order Runge-Kutta integrator. See http://mathworld.wolfram.com/Runge-KuttaMethod.html

```
rk4$stepper[ y_, h_, f_ ] := Module[ {k},

  k1 = h f[ y ];
  k2 = h f[ y + 1/2 k1 ];
  k3 = h f[ y + 1/2 k2 ];
  k4 = h f[ y + k3 ];

  y + 1/6 ( k1 + 2 k2 + 2 k3 + k4 )  // N
 ]
```

- **Semi-implicit Euler method** (semiImplicitEuler$stepper)

  The implementation bellow is demonstrative but very inefficient since it calls ODE function $\boldsymbol{f}$ twice. The implemented algorithm also requires that the state vector **y** has a special structure: 1) y_1 contains a time variable, 2) the time variable is followed by the ordinary state variables in the bottom-half and 3) the ordinary state variables are followed by their time derivatives in the top-half, i.e. the **y** should look like $\mathbf{y} = \{t, \mathbf{x}, \dot{\mathbf{x}}\}$ and its derivative (the ODE function $\boldsymbol{f}$) shoold return $\dot{\mathbf{y}} = \{1, \dot{\mathbf{x}}, \ddot{\mathbf{x}}\}$.

```
semiImplicitEuler$stepper[ y_, h_, f_ ] := Module[ { n, X, Xdot, y2 },

  n = 1 + ( Length[y] - 1 ) / 2; (* Split y into bottom- and top-halves *)

  Xdot = h f[ y ];  (* Solve velocities only *)
  Xdot[[1;;n]] = 0;  (* disregarding position and time solutions. *)
  y2 = y + Xdot;

  X = h f[ y2 ];     (* Now, solve position and time only *)
  X[[n+1;;]] = 0;    (* disregarding velocity solution. *)

  y + X + Xdot
 ]
```

Animation Functions

ςT: Transforms coordinates from body-fixed to inertial frame of reference

Transformations from non-inertial body-fixed (rotational) frame to intertial (world) frame of reference are based on global orientation quaternion q (variable Q) and position vector **x**: (variablle X)

```
ςT[ { x_, y_, z_ } ] := X + Im[ Q ** Q[0, x, y, z] ** Q* ]
```

```
ςT[ {} ] := {};
```

```
ςT[ points_ ?MatrixQ ] := ςT /@ points
```

```
ςT[ points__ ?VectorQ ] := ςT /@ { Sequence[ points ] }
```

getAnimationData: Gets coordinates of graphic primitives to be animated

Transforms body shape, orientation vectors and angular components into the inertial frame of reference

```
getAnimationData[] := {
  (* 1 *) ςT[body$v], (* body shape *)
  (* 2 *) body$i,
  (* 3 *) ςT[{0, 0, 0}, {a/2, 0, 0}], (* e1 axis *)
  (* 4 *) ςT[{0, 0, 0}, {0, b/2, 0}], (* e2 axis *)
  (* 5 *) ςT[{0, 0, 0}, {0, 0, c/2}], (* e3 axis *)
  (* 6 *) If[frameOfRef === Inertial, {{0, 0, 0}, ω/ωScale},
       (* else in body-fixed *) ςT[{0, 0, 0}, ω/ωScale]],
  (* 7 *) If[frameOfRef === Inertial, {{0, 0, 0}, L/LScale},
       (* else in body-fixed *) ςT[{0, 0, 0}, L/LScale]],
  (* 8 *) ωTrack (* angular velocity trajectory *)
 }
```

showAnimation: Renders 3D objects from animation data

Displays 3D graphics dynamically retrived by `getAnimationData` function.

```
showAnimation[] := Show[
  (* rigid body *)
  Graphics3D[{Yellow, Opacity[.2],
    GraphicsComplex[Dynamic[aData[[1]]], Dynamic[aData[[2]]]]}],
  (* ê1 axis *)
  Graphics3D[{Red, Thick, Line[Dynamic[aData[[3]]]]}],
  (* ê2 axis *)
  Graphics3D[{Green, Thick, Line[Dynamic[aData[[4]]]]}],
  (* ê3 axis *)
  Graphics3D[{Blue, Thick, Line[Dynamic[aData[[5]]]]}],
  (* Angular velocity ω *)
  Graphics3D[{Black, Thick, Line[Dynamic[aData[[6]]]]}],
  (* Angular momentum L *)
  Graphics3D[{Gray, Thick, Line[Dynamic[aData[[7]]]]}],
  (* Angular velocity ω trajectory *)
  Graphics3D[{Pink, Thick, Line[Dynamic[aData[[8]]]]}],
  (* Axes and plot range *)
  Boxed → True,
  (* Axes→True,AxesLabel→{ "x/m","y/m","z/m" },
  LabelStyle→Directive[ FontSize→12 ], *)
  ViewPoint → Front, ImageSize → Scaled[0.9],
  PlotRange → Dynamic@{{-ℓ, ℓ}, {-ℓ, ℓ}, {-ℓ, ℓ}}
 ]
```

▼ snapshotVars: Gets a verbose snapshot of state variables

Dumps all variables for debugging purposes. Usage: evaluate expression `Dynamic@snapshotVars[]` to get real-time update.

```
snapshotVars [] := TableForm[{
   {"Parameters", …, …},
   { "  Frame of Reference: " <> ToString@frameOfRef, …, …},
   { "  Gyroscopic Effects: " <> ToString@gyroEffects , …, …},
   {"Integrator", …, …},
   { Switch[ odeIntegrator,
     rk4$stepper, "  Runge-Kutta 4",
     semiImplicitEuler$stepper, "  Semi-Implicit Euler",
     _, "?" ] , …, …},
   { "  h", "=", h },
   { "Energy", …, …},
   { "  Eₖ", "=", Ek // N },
   { "  Eₚ", "=", Ep // N },
   { "  Eₜₒₜ", "=", Etot // N },
   { "Linear momentum, velocity and position", …, …},
   { "   |p⃗|", "=", Norm[P] // N },
   { "    p⃗", "=", P // N },
   { "   |v⃗|", "=", Norm[V] // N },
   { "    v⃗", "=", V // N },
   { "    x⃗", "=", X // N },
   { "Angular momentum, velocity and orientation", …, …},
   { "   |L⃗|", "=", Norm[L] // N },
   { "    L⃗", "=", L // N },
   { "   |ω⃗|", "=", Norm[ω] // N },
   { "    ω⃗", "=", ω // N },
   { "   |q|", "=", Abs[Q] // N },
   { "    q", "=", Q // QForm // N },
   { "Moment of inertia", …, …},
   { "  ‖𝒥‖", "=", Det[𝒥] // N }
  }, TableDepth → 2 ]
```

■ Equations of Motion

▼ rigidBodyEquations: Gets time derivatives of state variables

Equations of motion are depend on the chosen frame of reference and wheter gyroscopic effects are neglected or not. Thus, the moment of inertia $\mathcal{L}$, torque τ, angular momentum **L** and orientation time derivative $\dot{q}$ are given as piece-wise functions depending on the flags `frameOfRef` and `gyroEffects`.

```
rigidBodyEquations[{ t_, X_, Q_, P_, L_ }] := Module[
  { Pdot, Ldot, Xdot, Qdot, F, Fext, τ, R, 𝒥inv, ω },

  (* Calculate total force *)
  Fext = m gn; (* Sum up all external forces *)

  (* Calculate linear momentum time derivative *)
  Pdot = Fext + Fint;

  (* Moment of inertia *)
  R = { RotationMatrix[Q]   frameOfRef === Inertial ;

  𝒥inv = { R.𝒥0inv.Rᵀ   frameOfRef === Inertial
         { 𝒥0inv        frameOfRef === BodyFixed ;

  (* Derive angular velocity from angular momentum *)
  ω = 𝒥inv.L;

  (* Calculate total torque, depending on frame of reference *)
  τ = { τint                    frameOfRef === Inertial
      { Im[ Q* ** τint ** Q ]   frameOfRef === BodyFixed ;

  (* Calculate angular momentum time derivative *)
         { τ + ω × L   frameOfRef === Inertial ∧ gyroEffects === Ignore
  Ldot = { τ           frameOfRef === Inertial ∧ gyroEffects === Include
         { τ           frameOfRef === BodyFixed ∧ gyroEffects === Ignore
         { τ - ω × L   frameOfRef === BodyFixed ∧ gyroEffects === Include ;

  (* Calculate position time derivative (velocity) *)
  Xdot = m⁻¹ P;

  (* Calculate orientation time derivative *)
  Qdot = { 1/2 ω ** Q   frameOfRef === Inertial
         { 1/2 Q ** ω   frameOfRef === BodyFixed ;

  (* Return time derivatives of t, X, Q, P and L *)
  { 1, Xdot, Qdot, Pdot, Ldot }
]
```

ODE Solver

solverInit: Initializes state variables and computes derived quantities

```
solverInit [] := Module[
  { R, scalef },

  (* Stop any running simulations *)
  runSimulation = False;

  (* Initial time *)
  t = 0;

  (* Initial position and orientation, in inertial frame *)
  X = X0;
  Q = Sign[ Q0 ]; (* Normalize orientation to a versor *)

  (* Moment of inertia, frame dependent *)
  R = { RotationMatrix[ Q ]   frameOfRef === Inertial ;
  𝒥 = { R.𝒥0.Rᵀ   frameOfRef === Inertial
        𝒥0        frameOfRef === BodyFixed ;
  𝒥inv = { R.𝒥0inv.Rᵀ   frameOfRef === Inertial
           𝒥0inv        frameOfRef === BodyFixed ;

  (* Velocity and linear momentum, in inertial frame *)
  V = V0; (* Velocity *)
  P = m V; (* Linear momentum *)

  (* Initial angular velocity is always in body-fixed frame *)
  ω = { Im[ Q ** ω0 ** Q* ]   frameOfRef === Inertial
        ω0                    frameOfRef === BodyFixed ;
  (* Angular momentum, frame dependent *)
  L = 𝒥.ω; (* Angular momentum *)

  (* Internal forces and torque *)
  Fint = { 0, 0, 0 };
  τint = { 0, 0, 0 };

  (* Derived quantities *)
  Ek = 1/2 P.V + 1/2 L.ω; (* Kinetic energy *)
  Ep = -m gn.X; (* Potential energy *)
  Etot = Ek + Ep; (* Total energy *)

  (* Keep track of angular velocity *)
  ωTrack = {};

  (* Angular velocity and angular momentum scale factors *)
  scalef = 0.8 Max[ Abs[ body$v ], 0.9ℓ ];
  ωScale = Norm[ ω ] / scalef // N; If[ ωScale == 0, ωScale = 1 ];
  LScale = Norm[ L ] / scalef // N; If[ LScale == 0, LScale = 1 ];

  (* Update animation data *)
  aData = getAnimationData [];
 ]
```

▼ solverStep: Solves equations and recalculates derived quantities

Function calls repeatedly chosen ODE integrator, normalizes orientation qaternion to a versor and calculates derived quantities (like energy). (It saves also head of the angular velocity vector in an array to visualize precession and nutation.)

```
solverStep[ count_: 1 ] := Module[
  { R },

  Do[ If[ ¬ runSimulation, Break[] ];

      (* Solve equations using specified integrator *)
      { t, X, Q, P, L } = odeIntegrator[ { t, X, Q, P, L },
            h, rigidBodyEquations ];

      Q = Sign[ Q ];  (* Keep orientation as versor *)

      (* Update moment of inertia, but only if in inertial frame *)
      R = { RotationMatrix[ Q ]   frameOfRef === Inertial ;
      𝒥 = { R.𝒥0.Rᵀ   frameOfRef === Inertial
            𝒥0        frameOfRef === BodyFixed ;
      𝒥inv = { R.𝒥0inv.Rᵀ   frameOfRef === Inertial
               𝒥0inv        frameOfRef === BodyFixed ;

      (* Calculate derived quantities *)
      V = m^-1 P;    (* Linear velocity from linear momentum *)
      ω = 𝒥inv.L;  (* Angular velocity from angular momentum *)
      Ek = 1/2 P.V + 1/2 L.ω;  (* Kinetic energy *)
      Ep = -m gn.X;  (* Potential energy *)
      Etot = Ek + Ep;  (* Total energy *)

      (* Keep track of angular velocity *)
      AppendTo[ ωTrack,
           If[ frameOfRef === Inertial, ω, ςT[ ω ] ] / ωScale
       ],
    {count}
  ];

  (* Update animation data *)
  aData = getAnimationData[];
 ]
```

solverRun: Runs simulation until stopped

Creates the animation cell (if it does not exist) and evaluates `solverStep` function until `runSimulation` flag is reset to `False`.

```
solverRun[ locateCell_: False ] := Module[
  { nb = EvaluationNotebook[], noAnimationCell },

  (* Locate and evaulate cell containing solverRun[] *)
  If[ locateCell,
   NotebookFind[ nb, "RUNSIMULATION", Next, CellTags, AutoScroll → False ];
   SelectionEvaluateCreateCell[ nb ];
   Return []
  ];

  (* Recreate animation cell, if it does not exist *)
  If[ $Failed === NotebookFind[ nb, "ANIMATION", Next, CellTags, AutoScroll → False ],
      CellPrint[ ExpressionCell[ showAnimation [], CellTags → "ANIMATION" ] ];
      NotebookFind[ nb, "ANIMATION", Next, CellTags, AutoScroll → False ];
      SetOptions[ NotebookSelection[ nb ], CellAutoOverwrite → False ]
  ];

  NotebookFind[ nb, "RUNSIMULATION", Next, CellTags, AutoScroll → False ];
  SelectionMove[ nb, After, CellContents ];

  (* Run simulation, until cancelled *)
  runSimulation = True;
  While[ runSimulation ∧ t < tf, solverStep [] ];
  runSimulation = False;
 ]
```

Initialize Simulation

Default parameters are conviniently modified here before running simulation.

```
ℓ = 3.5; gn = { 0, 0, 0 }; tf = 4;

setupBodyShape[ 10, { 4, 5, 2}, "Cuboid" ];

X0 = { 0, 0, 0 }; (* in inertial frame *)
Q0 = N@ToQ$AngleAxis[ 0.1, { 1, 0, 0.5 } ]; (* in inertial frame *)
V0 = { 0, 0, 0 }; (* in inertial frame *)
ω0 = { -1, -3, 2 }; (* in body-fixed (!) frame *)

solverInit [];
```

Simulation

Frame: Body–fixed , Gyroscopic effects: Include

Integrator: Runge–Kutta 4 , h = 0.01 s

Initialize | Single Step | Run Simulation | Cancel

Stopped.

$t = 4.080$ s, $E_{tot} = 155.417$ J

$|\vec{\omega}_{world}| = \{-1.6, -1.6, 2.9\}\ s^{-1}$

$|\vec{L}_{world}| = \{-21.5, -57.7, 62.9\}\ kg\,m^2\,s^{-1}$

```
solverRun[]
```