Appendix A - Rigid Body Motion in 3D

Mikica Kocic, miko0008@student.umu.se
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/Graphics-Complex.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 $\mathbf{y}$ 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 $\mathbf{y}$ 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, χ, χdot, y2 },

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

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

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

  y + χ + χdot
 ]
```

■ 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 $\mathbf{x}$: (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 *) ϛT[ body$v ], (* body shape *)
  (* 2 *) body$i,
  (* 3 *) ϛT[ {0, 0, 0}, {a / 2, 0, 0} ], (* e1 axis *)
  (* 4 *) ϛT[ {0, 0, 0}, {0, b / 2, 0} ], (* e2 axis *)
  (* 5 *) ϛT[ {0, 0, 0}, {0, 0, c / 2} ], (* e3 axis *)
  (* 6 *) If[ frameOfRef === Inertial, { {0, 0, 0}, ω / ωScale },
       (* else in body-fixed *) ϛT[ {0, 0, 0}, ω / ωScale ] ],
  (* 7 *) If[ frameOfRef === Inertial, { {0, 0, 0}, L / LScale },
       (* else in body-fixed *) ϛT[ {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},
   {"  E_tot", "=", 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 $\mathbf{L}$ 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 g n; (* Sum up all external forces *)

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

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

  𝒥inv = { R.𝒥0inv.R^T   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
          { τ           frameOfRef === Inertial ∧ gyroEffects === Include
  Ldot = { 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 *)
          { 1/2 ω ** Q   frameOfRef === Inertial
  Qdot = {                                      ;
          { 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.R^T   frameOfRef === Inertial
       𝒥0         frameOfRef === BodyFixed ;
  𝒥inv = {R.𝒥0inv.R^T   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.R^T   frameOfRef === Inertial
          { 𝒥0         frameOfRef === BodyFixed ;

      𝒥inv = { R.𝒥0inv.R^T   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 g n.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

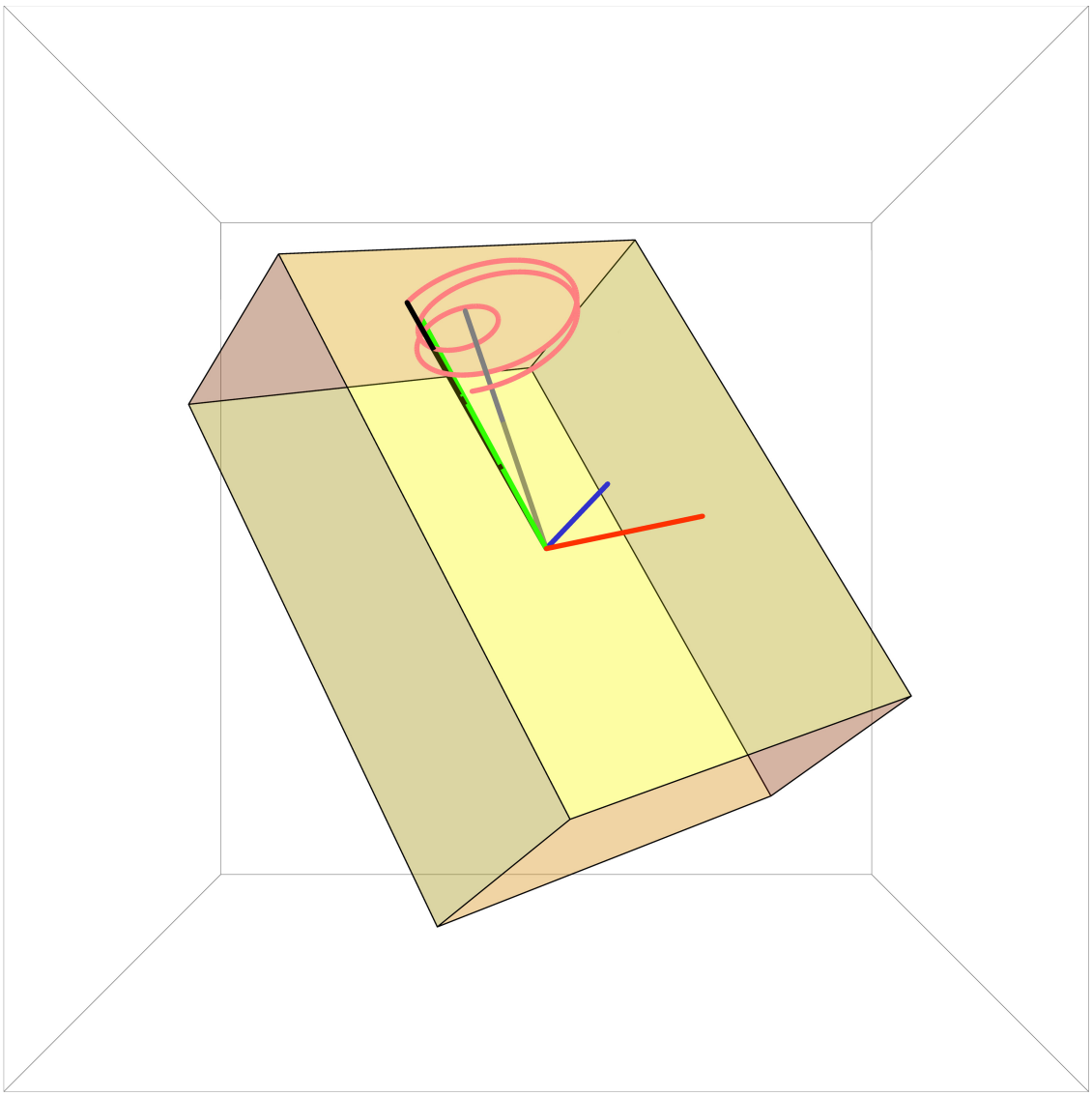
Stopped.

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 []
```



Appendix B - Quaternions

Package for symbolic calculations with quaternions

```
(*
   Quaternions package implements Hamilton's quaternion algebra.
   This package mimics original Mathematica Quaternions.m, however,
   it fixes various bugs and also allows symbolic transformations
   of quaternions to more extent than Mathematica's original.
   Author:  Mikica B Kocic
   Version: 0.4, 2012-04-22
*)
```

Begin Package

```
$Pre =.
```

```
BeginPackage[ "Quat`" ];
```

```
Unprotect[ Quat`Q ];
```

```
ClearAll[ "Quat`Private`*" ];
ClearAll[ "Quat`*" ];
```

Exported Symbols

```
Q::usage = "Q[w,x,y,z] represents the quaternion with real part w \
(also called scalar part) and imaginary part {x,y,z} (also called vector part)";
```

```
QQ::usage = "QQ[q] gives True if q is a quaternion, \
and False otherwise.";
```

```
ToQ::usage = "ToQ[expr] transforms expr into a quaternion object if at all possible.";
```

```
ScalarQ::usage = "ScalarQ[q] gives True if q is a scalar, and False otherwise.";
```

```
ToList::usage = "ToList[q] gives components of q in a list";
```

```
ToVector::usage = "ToVector[q] gives q as column vector matrix";
```

```
ToMatrix::usage = "ToMatrix[q] gives matrix representation of quaternion q";
```

```
Abs$Im::usage = "Abs$Im[q] gives the absolute value of the vector quaternion part of q.";
```

```
AdjustedSign$Im::usage = "AdjustedSign$Im[q] gives the Sign of the vector part of q, \
adjusted so its first non-zero part is positive.";
```

```
NonCommutativeMultiply::usage = "Implements non-commuatative quaternion multiplication.";
```

```
ToQ$AngleAxis::usage = "ToQ$AngleAxis[θ,{ux,uy,uz}] transforms axis u and angle θ into quat
```

```
ToQ::invargs = "ToQ: failed to construct Q from argument:\n args = `1`"
```

```
RotationMatrix4::usage = "RotationMatrix4[q] gives the 4×4 rotation matrix for a \
counterclockwise 3D rotation around the quaterion q"
```

```
QForm::usage = "QForm[q] prints with the elements of matrix or quaternion \
arranged in a regular array."
```

```
Assert::usage = "QForm[q] prints with the elements of matrix or quaternion \
arranged in a regular array."
```

Private Section

```
Begin[ "Quat`Private`" ];
```

Operators

```
Subscript[ q_ ?QQ, n_Integer /; 1 ≤ n ≤ 4 ] := q[[n]]
```

```
AngleBracket[ q_ ?QQ ] := Re[q]
```

```
OverVector[ q_ ?QQ ] := q[[2;;4]] /. Q → List
```

```
OverHat[ q_ ?QQ ] := q / Abs[ q ]
```

```
SuperStar[ q_ ?QQ ] := Conjugate[ q ]
```

```
BracketingBar[ q_ ?QQ ] := Abs[ q ]
```

Utility Functions

Strip Output Label of Form Info

```
SetAttributes[ TimeIt, HoldAll ];
```

```
TimeIt[ expr_ ] :=
Module[
  { result = expr, out, form },
  If[ TrueQ[ MemberQ[ $OutputForms, Head[result] ] ],
    (* Then *) out = First[result]; form = "//" <> ToString[ Head[result] ],
    (* Else *) out = result; form = ""
  ];
  If[ out =!= Null,
    CellPrint[
      ExpressionCell[ result, "Output",
        CellLabelAutoDelete → True,
        CellLabel → StringJoin[ "Out[", ToString[$Line], "]:="]
      ]
    ];
    Unprotect[ Out ];
    Out[ $Line ] = out;
    Protect[ Out ];
    out (* needed for % *)
  ];
];
```

▼ Assert truth of a logical statement (i.e. prove a theorem)

```
Assert[ statement_, params__: Null ] :=
Module[
  { result },
  (* Try to proove the statement... *)
  result = statement;
  If[ result,
    (* is true *)  Return@ Style[ "⊢ True ∎", Blue ],
    (* is false *) Return@ Style[ "False ☹", Red, Large, Bold ],
    (* is neither true or false *) Null (* continue... *)
  ];
  (* ... now, try even harder to proove the statement *)
  result = Simplify[ statement,params ];
  If[ result,
    (* is true *)  Return@ Style[ "⊢ True (after Simplify) ∎", Blue ],
    (* is false *) Return@ Style[ "False (after Simplify) ☹", Red, Large, Bold ],
    (* is neither true or false *) Null (* continue... *)
  ];
  (* ... now, try even harder to proove the statement *)
  result = FullSimplify[ statement, params ];
  If[ result,
    (* is true *)  Style[ "⊢ True (after FullSimplify) ∎", Blue ],
    (* is false *) Style[ "False (after FullSimplify) ☹", Red, Large, Bold ],
    (* is neither true or false *)
        Style[ "Indeterminate ☹", Darker[Red], Large, Bold ]
  ]
]
```

■ Transformation Rules

▼ Scalar test

```
ScalarQ[ x_ ] := AtomQ[ x ] ⋁ Length[ x ] === 0 ⋁
  ( Length[ x ] =!= 0 ⋀ ¬ListQ[ x ] ⋀ Head[ x ] =!= Q ⋀ Head[ x ] =!= Complex )
```

▼ Quaternion test

```
QQ[ q_ ] :=
  Head[ q ] === Q ⋀ Length[ q ] === 4 ⋀
  And @@ ( ScalarQ /@ q )
```

▼ Transform expressions into Q objects

```
ToQ[ w_ ?ScalarQ, { x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ] :=
    Q[ w, x, y, z ]
```

```
ToQ[ q_ ] := q /. {
    Q[ w_, x_, y_, z_ ] ↦ Q[ w, x, y, z ],
    { x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ }  ↦ Q[ 0, x, y, z ],
    { w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ↦ Q[ w, x, y, z ],
    { {w_ ?ScalarQ}, {x_ ?ScalarQ}, {y_ ?ScalarQ}, {z_ ?ScalarQ} } ↦ Q[ w, x, y, z ],
    Complex[ x_, y_ ] ↦ Q[ x, y, 0, 0 ],
    Plus[ x_, Times[ Complex[ 0, 1 ], y_ ] ] ↦ Q[ x, y, 0, 0 ],
    Times[ Complex[ 0, 1 ], x_ ] ↦ Q[ 0, x, 0,  0 ],
    Times[ Complex[ 0, x_ ], y_ ] ↦ Q[ 0, x y, 0, 0 ],
    x_ ?ScalarQ ↦ Q[ x, 0, 0, 0],
    (* unknown list *)
    list_ ↦ Module[ {}, Message[ ToQ::invargs, list ]; Abort[] ]
} // QSimplify
```

Axis and angle into Q object

```
ToQ$AngleAxis[ θ_, { ux_, uy_, uz_ } ] := Q[
    Cos[θ/2], ux Sin[θ/2], uy Sin[θ/2], uz Sin[θ/2]
]
```

To list

```
ToList[ q_ ?QQ ] := ( q /. Q→ List )
```

Vector

```
ToVector[ q_ ?QQ ] := Transpose@{ q /. Q→ List }
```

To matrix representation

```
ToMatrix[ Q[ w_, x_, y_, z_ ] ] := {
   {  w,  x,  y,  z },
   { -x,  w, -z,  y },
   { -y,  z,  w, -x },
   { -z, -y,  x,  w }
}
```

To matrix form

```
Q /: MatrixForm[ q: Q[ __ ?ScalarQ ] ] := MatrixForm@ ToList@ q
```

To paritioned matrix form (with divider lines)

```
QForm[ Q_ ?MatrixQ ] := DisplayForm@
  RowBox[{
    "(", "",
    GridBox[ Release[ Q ],
      RowSpacings → 1, ColumnSpacings → 1,
      RowAlignments → Baseline,
      ColumnAlignments → Center,
      GridBoxDividers → {
        "Columns" → { False, GrayLevel[0.84] },
        "Rows" → { False, GrayLevel[0.84] }
      }
   ],
   "", ")"
 }]
```

```
QForm[ q_ ?QQ ] := QForm[ { Release[ ToList@ q ] } ]
```

To rotation matrix, Shoemake's form

```
Q /: RotationMatrix[ Q[ w_, x_, y_, z_ ] ] :=
{
   {  1 - 2( y^2 + z^2 ),    2 x y - 2 w z,      2 x z + 2 w y  },
   {  2 x y + 2 w z,      1 - 2( x^2 + z^2 ),  2 y z - 2 w x  },
   {  2 x z - 2 w y ,     2 w x+2 y z,         1 - 2( x^2 + y^2 ) }
}
```

```
Q /: RotationMatrix4[ Q[ w_, x_, y_, z_ ] ] :=
{
   {  1,  0,                  0,                  0                 },
   {  0,  1 - 2( y^2 + z^2 ),  2 x y - 2 w z,     2 x z + 2 w y     },
   {  0,  2 x y + 2 w z,      1 - 2( x^2 + z^2 ),  2 y z - 2 w x     },
   {  0,  2 x z - 2 w y ,     2 w x + 2 y z,      1 - 2( x^2 + y^2 ) }
}
```

▼ Conjugate

```
Q /: Conjugate[ Q[ w_, x_, y_, z_ ] ]:=
    Q[ w, -x, -y, -z ]
```

▼ Squared Norm

```
Q /: SqNorm[ q:  Q[ __ ?ScalarQ ] ] :=
    Plus @@ ( ( List @@ q )^2 )
```

▼ Norm

```
Q /: Norm[ q: Q[ __ ?ScalarQ ] ]:=
    Sqrt[ SqNorm[ q ] ]
```

▼ Abs

```
Q /: Abs[ q:  Q[ __ ?ScalarQ ] ] :=
    Sqrt[ SqNorm[ q ] ]
```

▼ Round

```
Q /: Round[ q: Q[ __ ?ScalarQ ] ]:=
Module[
  {
    cent = Round /@ q;
    mid = ( Floor /@ q ) + Q[ 1/2, 1/2, 1/2, 1/2 ];
  },
  If[ SqNorm[ q - cent ] <= SqNorm[ q - mid ], cent, mid ]
]
```

▼ Real Part

```
Q /: Re[ q: Q[ __ ?ScalarQ ] ] :=
    q[[1]]
```

▼ Sign (returns Versor)

```
Q /: Sign[ q: Q[ __ ?ScalarQ ] ] :=
    Q[ 1, 0, 0, 0 ] /; Abs[ q ] == 0
```

```
Q /: Sign[ q: Q[ __ ?ScalarQ ] ] :=
    q / Abs[ q ]
```

▼ Imaginary Part

```
Q /: Im[ q: Q[ __ ?ScalarQ ] ] :=
    { q[[2]], q[[3]], q[[4]] }
```

```
AdjustedSign$Im[ q_Complex | q_ ?ScalarQ ] := ⅈ
```

```
AdjustedSign$Im[ Q[ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ ] ] :=
Which[
    x != 0,
        Sign[ x ] * Sign[ Q[ 0, x, y, z ] ],
    y != 0,
        Sign[ y ] * Sign[ Q[ 0, x, y, z ] ],
    z != 0,
        Sign[ z ] * Sign[ Q[ 0, x, y, z ] ],
    True,
        i (* ?abort? *)
]
```

```
Abs$Im[ Q[ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ ] ]:=
    Sqrt[x^2 + y^2 + z^2]
```

```
Abs$Im[ x_ ? NumericQ ] :=
    Im[ x ]
```

```
Sign$Im[ Q[ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ ] ] :=
    Sign[ Q[ 0, x, y, z ] ]
```

▼ Addition

```
Q /: Q[ w1_, x1_, y1_, z1_ ] + Q[ w2_, x2_, y2_, z2_ ] :=
    Q[ w1 + w2, x1 + x2, y1 + y2, z1 + z2 ] // QSimplify
```

```
Q /: Complex[ re_, im_ ] + Q[ w_, x_, y_, z_ ] :=
    Q[ w + re, x + im, y, z ] // QSimplify
```

```
Q /: λ_ ?ScalarQ + Q[ w_, x_, y_, z_ ] :=
    Q[ w + λ, x, y, z ]
```

▼ Multiplication

```
Q /: λ_ ?ScalarQ * Q[ w_, x_, y_, z_ ]:=
    Q[ λ w, λ x, λ y, λ z ]
```

▼ Non-commutative multiplication

```
Q /: Q[ w1_, x1_, y1_, z1_ ] ** Q[ w2_, x2_, y2_, z2_ ]:=
    Q[
        w1 w2 - x1 x2 - y1 y2 - z1 z2,
        w1 x2 + x1 w2 + y1 z2 - z1 y2,
        w1 y2 - x1 z2 + y1 w2 + z1 x2,
        w1 z2 + x1 y2 - y1 x2 + z1 w2
    ] // QSimplify
```

▼ Non-commutative multiplication with complex numbers

```
Unprotect[ NonCommutativeMultiply ]
(*SetAttributes[ NonCommutativeMultiply, Listable ]*)
```

```
a_ ?ScalarQ ** b_ ?QQ := a * b
```

```
a_ ?QQ ** b_ ?ScalarQ := a * b
```

```
{ x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ** q_ ?QQ :=
    Q[ 0, x, y, z ] ** q
```

```
q_ ?QQ ** { x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } :=
    q ** Q[ 0, x, y, z ]
```

```
{ w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } ** q_ ?QQ :=
    Q[ w, x, y, z ] ** q
```

```
q_ ?QQ ** { w_ ?ScalarQ, x_ ?ScalarQ, y_ ?ScalarQ, z_ ?ScalarQ } :=
    q ** Q[ w, x, y, z ]
```

```
{ {w_ ?ScalarQ}, {x_ ?ScalarQ}, {y_ ?ScalarQ}, {z_ ?ScalarQ} } ** q_ ?QQ :=
    Q[ w, x, y, z ] ** q
```

```
q_ ?QQ ** { {w_ ?ScalarQ}, {x_ ?ScalarQ}, {y_ ?ScalarQ}, {z_ ?ScalarQ} } :=
    q ** Q[ w, x, y, z ]
```

```
( ( a_: 1 ) * Complex[ b_, c_ ] ) ** ( ( x_: 1 ) * Complex[ y_, z_ ] ) :=
    a b x y - a c x z + ( a c x y + a b x z ) i
```

```
( x_ + y_ ) ** a_ := ( x ** a ) + ( y ** a )
a_ ** (x_ + y_) := ( a ** x ) + ( a ** y )
```

```
Protect[ NonCommutativeMultiply ]
```

▼ Math Functions

```
extfunc[ func_, a_, b_ ] :=
    Re[b] + Abs$Im[b] * AdjustedSign$Im[a]
```

```
Block[
    { extend, $Output = {} },

    extend[ foo_ ]:= (
        Unprotect[ foo ];
        Q /: foo[ a: Q[ __ ?ScalarQ ] ] :=
           extfunc[ foo, a, foo[ Re[a] + Abs$Im[a] * i ] ];
        Protect[ foo ];
    );

    extend /@ {
        Log,
        Cos, Sin, Tan, Sec, Csc, Cot,
        ArcCos, ArcSin, ArcTan, ArcSec, ArcCsc, ArcCot,
        Cosh, Sinh, Tanh, Sech, Csch, Coth,
        ArcCosh, ArcSinh, ArcTanh, ArcSech, ArcCsch, ArcCoth
    };
]
```

▼ Exp e^q

```
Q /: Exp[ q: Q[ __ ?ScalarQ ] ] :=
    Exp[ Re[ q] ] *
    ( Cos[ Abs$Im[ q ] ] + Sin[ Abs$Im[ q ] ] * Sign$Im[ q] ) // QSimplify
```

▼ Power

```
Q /: Power[ q: Q[ __ ?ScalarQ ], 0 ] :=
    1
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], 1 ] :=
    q
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], -1 ] :=
    Conjugate[q] ** ( 1 / SqNorm[q] )
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], n_ ] :=
    q ** Power[ q, n - 1 ] /; n > 1 ⋀ n ∈ Integers
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], n_ ] :=
    Power[ 1/q, -n ] /; n < 0 ⋀ n != -1
```

```
Q /: Power[ q: Q[ __ ?ScalarQ ], n_ ] :=
Module[
  {
    μ = Abs[ q ], (* modulus of {q1,q2,q3,q4} *)
    re = Re[ q ], (* scalar part re = {q1} *)
    im = Abs$Im[ q ], (* modulus of vector part im = {q2,q3,q4} *)
    θ, (* Angle between vector im and scalar re *)
    ϕ = 0, (* Angle between q2 and vector part {q2,q3,q4} *)
    γ = 0 (* Angle between q3 and subvector {q3,q4} *)
  },

  θ = If[ re =!= 0,
        (* Then *) ArcTan[ im / re ],
        (* Else *) π/2
      ];

  If[ im =!= 0,
  (* Then *)
     ϕ = ArcCos[ q[[2]] / im ];
     γ = If[ Sin[ϕ] =!= 0,
         (* Then *)
            ArcCos[ q[[3]] / ( im Sin[ϕ] ) ] Sign[ q[[4]] ],
            ( π/2 ) Sign[ q[[4]] ]
         ]
  ];

  Q[
     μ^n Cos[ n θ ] ,
     μ^n Sin[ n θ ] Cos[ϕ] ,
     μ^n Sin[ n θ ] Sin[ϕ] Cos[γ],
     μ^n Sin[ n θ ] Sin[ϕ] Sin[γ]
  ]  // QSimplify

] /; ScalarQ[n] ⋀ n > 0
```

▼ Power e^q

```
Q /: Power[ E, q: Q[ __ ?ScalarQ ] ] :=
    Exp[ q ]
```

▼ Sqrt

```
Q /: Sqrt[ q: Q[ __ ?ScalarQ ] ] :=
    Power[ q, 1/2 ]
```

▼ Right Divide

```
Q /: Divide[ left: Q[ __ ?ScalarQ ], right : Q[ __ ?ScalarQ ] ] :=
    left ** ( Conjugate[right] ** 1/SqNorm[right] )
```

▼ Simplification

```
Q /: QSimplify[ q: Q[ __ ?ScalarQ ] ] :=
  Simplify[ TrigExpand /@ q ]
```

```
QSimplify[ q_ ] := q
```

End Package

```
$Pre = TimeIt;
```

```
End[];
```

```
EndPackage[];
```

Appendix C - Rotation of Moment of Inertia Tensor using Quaternions

Mikica Kocic, miko0008@student.umu.se
The Physics of Virtual Environments, 2012-04-22

There is a wonderful connection between complex numbers (and quaternions as their extension) and geometry in which, translations correspond to additions, rotations and scaling to multiplications and reflections to conjugations.

However, the beauty of using quaternions for spatial rotations is shadowed by an expression like

$$\begin{pmatrix} 1-2\left(y^2+z^2\right) & 2xy-2wz & 2wy+2xz \\ 2xy+2wz & 1-2\left(x^2+z^2\right) & 2yz-2wx \\ 2xz-2wy & 2wx+2yz & 1-2\left(x^2+y^2\right) \end{pmatrix}\cdot$$

$$\mathcal{J}\cdot\begin{pmatrix} 1-2\left(y^2+z^2\right) & 2xy-2wz & 2wy+2xz \\ 2xy+2wz & 1-2\left(x^2+z^2\right) & 2yz-2wx \\ 2xz-2wy & 2wx+2yz & 1-2\left(x^2+y^2\right) \end{pmatrix}^{\mathrm{T}}$$

which actually represents such simple transformation as rotation of moment of inertia tensor using quaternions [2].

The purpose of this document is to express spatial rotations of a moment of inertia tensor in pure quaternionic form and to highlight an algebraical meaning behind such rotations (read: to expell the evil of linear algebra from rotation equations :)

■ Conventions and Definitions

Load quaternions package (see `Quat.nb` for annotated source code)

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

▼ Conventions

In this document, matrices, vectors and matrix representation of quaternions are shown in uppercase font (roman-bold font in text) and quaternions and other numbers are shown in lowercase font (italic font in text). Since the symbol `I` in *Mathematica* is used for imaginary unit, the selected symbol for moment of inertia is uppercase script J i.e. $\mathcal{J}$. The identity matrix is suffixed with number of dimensions, i.e. `I4` represents identity matrix in $\mathbb{R}^4$.

Matrices 4×4 originated from quaternions may be geometrically partitioned into scalar (real), vector (pure imaginary) and cross-product parts [3]. Such partitions are visualized using row and column divider lines throughout this document:

$$\left(\begin{array}{c|ccc} \textit{scalar} & \square & \textit{vector} & \square \\ \hline \square & \square & \square & \square \\ \textit{vector}^{\mathrm{T}} & \square & \textit{cross product} & \square \\ \square & \square & \square & \square \end{array}\right)$$

▼ General symbols and variables used in theorems

Hamilton's quaternions q, q_1 and q_2 with real number components (but not with complex number components as biquaternions; see definition of default `$Assumptions` bellow)

```
q = Q[ w, x, y, z ];
```

```
q1 = Q[ w1, x1, y1, z1 ];
```

```
q2 = Q[ w2, x2, y2, z2 ];
```

Identity matrix `I4` in $\mathbb{R}^4$

```
I4 = IdentityMatrix[4];
```

Common $\mathbb{R}^4$ matrices **A** and **B**

```
A = ( Aww | Awx  Awy  Awz
      Axw | Axx  Axy  Axz
      Ayw | Ayx  Ayy  Ayz
      Azw | Azx  Azy  Azz );
```

```
B = ( Bww | Bwx  Bwy  Bwz
      Bxw | Bxx  Bxy  Bxz
      Byw | Byx  Byy  Byz
      Bzw | Bzx  Bzy  Bzz );
```

Symmetrical $\mathbb{R}^4$ matrix representing moment of inertia tensor $\mathcal{J}$

```
𝒥 = ( 1 |  0    0    0
      0 | Ixx  Ixy  Ixz
      0 | Ixy  Iyy  Iyz
      0 | Ixz  Iyz  Izz );
```

Default assumption used for all asserted statements and expressions in this notebook is that previously defined entities q, q_1, q_2, **A**, **B** and $\mathcal{J}$ are on field of $\mathbb{R}$, i.e. that their components are elements in $\mathbb{R}$

```
$Assumptions = Flatten@{ ToList /@ { q, q1, q2 }, A, B, 𝒥} ∈ Reals;
```

■ Multiplications

The ordirary notation for multiplication in *Mathematica* is

A . **B**	multiplication between matrices ("dot" multiplication)
a × **b**	cross-product between 3D vectors
p ** q	quaternion multiplication (both in `Quat.m` and *Mathematica* Quaternions package)

Beside the ordinary notation for multiplication, this document also introduces the following notation based on the center dot (·) operator.

A · **B**	:= multiplication between matrices
$p \cdot q$	:= quaternion multiplication
$q \cdot$ **A**	:= column-wise quaternion-matrix multiplication

Note that the center dot operator in *Mathematica* is without built-in meaning.

▼ Matrix multiplication

Define center dot operator **A** · **B** as matrix multiplication (otherwise, *Mathematica* uses plain dot "." for matrix multiplications).

```
CenterDot[ m__ ?MatrixQ ] := Dot[ m ]
```

▼ Quaternion multiplication

Mathematica uses non-commutative multiply (**) operator as symbol for quaternion multiplication. The `Quat.m` package and this document follow this convention. Here we define also the center dot operator as symbol for quaternion multiplication, just for convinience and esthetics

```
CenterDot[ q__ ?QQ ] := NonCommutativeMultiply[ q ]
```

▼ Column-wise quaternion-matrix multiplication

Let us now define multiplication between quaternion q and matrix $\mathbf{M}$, where it is assumed that matrix is a row vector of quaternions in columns, i.e. $\mathbf{M} = (\, q_1 \quad q_2 \quad \ldots \quad q_n \,)$, so that quaternion multiplication is done between q and q_j in every column j.

$$q \cdot \left(\begin{array}{c|ccc} q_{11} & q_{12} & & q_{1n} \\ \hline q_{21} & q_{22} & & q_{2n} \\ q_{31} & q_{32} & \cdots & q_{3n} \\ q_{41} & q_{42} & & q_{4n} \end{array}\right) := \overset{\begin{array}{ccccc} q\,q_1 & & q\,q_j & & q\,q_n \\ \downarrow & & \downarrow & & \downarrow \end{array}}{\left(\begin{array}{c|ccc} q_{11} & q_{12} & & q_{1n} \\ \hline q_{21} & q_{22} & & q_{2n} \\ q_{31} & q_{32} & \cdots & q_{3n} \\ q_{41} & q_{42} & & q_{4n} \end{array}\right)}$$

Since the quaternion multiplication is not commutative, there are several cases to distinguish:

a) Left multiplication $q \cdot \mathbf{M}$, where quaternion in each column in $\mathbf{M}$ is multiplied by quaternion q from the left:

```
CenterDot[ q_?QQ, mat_?MatrixQ ] := Transpose@
  Table[
   ToList[ q · ToQ@mat⟦All,j⟧ ], {j, 1, Dimensions[mat]⟦2⟧}
  ]
```

b) Right multiplication $\mathbf{M} \cdot q$, where quaternion in each column in $\mathbf{M}$ is multiplied by quaternion q from the right:

```
CenterDot[ mat_?MatrixQ, q_?QQ ] := Transpose@
  Table[
   ToList[ ToQ@mat⟦All,j⟧ · q ], {j, 1, Dimensions[mat]⟦2⟧}
  ]
```

c) Compound multiplication from both sides $p \cdot \mathbf{M} \cdot q$, where multiplication of each column in M by quaternion p from the left and q from the right:

```
CenterDot[ q1_?QQ, mat_?MatrixQ, q2_?QQ ] := Transpose@
  Table[
   ToList[ q1 · ToQ@mat⟦All,j⟧ · q2 ], {j, 1, Dimensions[mat]⟦2⟧}
  ]
```

The multiplication between quaternion and matrix is associative, i.e. it holds

$$q_1 \cdot \mathbf{M} \cdot q_2 == q_1 \cdot (\mathbf{M} \cdot q_2) == (q_1 \cdot \mathbf{M}) \cdot q_2$$

which is proved as theorem in the following sections.

Note that for a single column matrix $\mathbf{M}$, the multiplication between quaternion and matrix falls back to an ordinary quaternion multiplication.

▼ Row-wise quaternion-matrix multiplication

Row-wise quaternion-matrix multiplication can be defined transposing matrices and using column-wise multiplication

$$\left(q_1 \cdot \mathbf{M}^{\mathsf{T}} \cdot q_2\right)^{\mathsf{T}}$$

▼ Conj[] matrix operator

Conj[] operator conjugates only the vector part of the matrix, but not the scalar and the cross product parts:

```
Conj[ q_?MatrixQ ] :=
```

$$\left(\begin{array}{c|ccc} q_{[\![1,1]\!]} & -q_{[\![1,2]\!]} & -q_{[\![1,3]\!]} & -q_{[\![1,4]\!]} \\ \hline -q_{[\![2,1]\!]} & q_{[\![2,2]\!]} & q_{[\![2,3]\!]} & q_{[\![2,4]\!]} \\ -q_{[\![3,1]\!]} & q_{[\![3,2]\!]} & q_{[\![3,3]\!]} & q_{[\![3,4]\!]} \\ -q_{[\![4,1]\!]} & q_{[\![4,2]\!]} & q_{[\![4,3]\!]} & q_{[\![4,4]\!]} \end{array}\right)$$

Note that Conj[] is **not** a conjugate of a quaternion! The conjugate of a quaternion corresponds to the plain transpose of the matrix.

TODO: compare Conj[] to extracting vector part extraction operations.

Background

Basic idea behind quaternion-matrix multiplication comes from an expression

```
Assert[
 Q[1, 0, 0, 0] == q · Q[1, 0, 0, 0] · q*,
 Assumptions → |q|^2 == 1
]
```

```
⊢ True (after Simplify) ∎
```

for which it is assumed to be also valid when transforming identity matrix I4 that is equvalent representation of quaternion *Q*[1, 0, 0, 0] in matrix form, i.e. that there should be kind of multiplication where

```
Assert[
 (1  0  0  0)^T == q · (1  0  0  0)^T · q*,
 Assumptions → |q|^2 == 1
]
```

```
⊢ True (after Simplify) ∎
```

Theorems

Quaternion-matrix multiplication

Quaternion multiplication associativity (sanity-check)

```
q1 · q · q2 == q1 · (q · q2) == (q1 · q) · q2 // Assert
```

```
⊢ True ∎
```

Quaternion-matrix multiplication associativity

```
q1 · A · q2 == q1 · (A · q2) == (q1 · A) · q2 // Assert
```

```
⊢ True ∎
```

```
q · A · q* == (q · A) · q* // Assert
```

```
⊢ True ∎
```

```
q · A · q* == q · (A · q*) // Assert
```

```
⊢ True ∎
```

```
(q · I4)^T == q* · I4 // Assert
```

```
⊢ True ∎
```

```
(I4 · q)^T == I4 · q* // Assert
```

```
⊢ True ∎
```

```
(q · I4) · A == q · A // Assert
```

```
⊢ True ∎
```

```
(I4 · q) · A == A · q // Assert
```

```
⊢ True ∎
```

```
(q1 · I4) · (I4 · q2) == q1 · I4 · q2 // Assert
```

⊢ True ∎

```
(I4 · q2) · ( (q1 · I4) · A ) == (q1 · A) · q2 == q1 · A · q2 // Assert
```

⊢ True (after Simplify) ∎

```
(I4 · q2) · (q1 · I4) · A == q1 · A · q2 ⋀
  (I4 · q2) · (q1 · I4) · A == q1 · A · q2 ⋀
  (q1 · I4) · (I4 · q2) · A == q1 · A · q2 ⋀
  (q1 · I4 · q2) · A == q1 · A · q2  // Assert
```

⊢ True (after Simplify) ∎

Conjugations of quaternion-matrix multiplications

```
Conj[ q · A ] == Conj[A] · q*  // Assert
```

⊢ True ∎

```
Conj[ A · q ] == q* · Conj[A] // Assert
```

⊢ True ∎

```
Conj[ q1 · A · q2] == Conj[ q1 · (A · q2)] == Conj[A · q2] · q1* == q2* · Conj[A] · q1* // Assert
```

⊢ True (after Simplify) ∎

```
Conj[ q · A · q*] == q · Conj[A] · q* // Assert
```

⊢ True (after Simplify) ∎

▼ Left and right rotation matrices

Because quaternion multiplication is bilinear, it can be expressed in matrix form, and in two different ways ([1]). Multiplication on the left qp gives $\mathbf{L}_q \cdot \mathbf{p}$ where p is now treated as a 4-dimensional column vector $\mathbf{p}$ and multiplication on the right pq gives $\mathbf{p} \cdot \mathbf{R}_q$.

Let us define $\mathbf{L}_q$ and $\mathbf{R}_q$ as quaternion-multiplication with identity matrix.

```
ClearAll[ L, R, Q ]
```

```
L_q_ := q · I4
R_q_ := I4 · q
Q_q_ := L_q · R_q*
```

Proof that quaternion multiplication can be decomposed into L&R matrix parts and that $\mathbf{L}_q \, \mathbf{R}_{q^*} \, \mathbf{p}$ corresponds to $q\,p\,q^*$

```
Assert[
 L_q · R_q* · ToVector[q1] == ToVector[ q · q1 · q* ],
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

▼ Properties of L, R and Q matrices

```
L_q* == (L_q)^T  // Assert
```

⊢ True ∎

```
R_q* == (R_q)^T  // Assert
```

⊢ True ∎

```
(q · Rq*) · A == q · (Rq* · A) == (Lq · A) · q* // Assert
```

⊢ True (after Simplify) ∎

```
Qq == Lq · Rq* == (Rq*ᵀ · Lqᵀ)ᵀ == (Rq · Lq*)ᵀ == Rq* · Lq // Assert
```

⊢ True ∎

```
Qq* == Rq · Lq* == Lq* · Rq // Assert
```

⊢ True ∎

```
Qq · Qq* == |q|^4 I4 // Assert
```

⊢ True (after Simplify) ∎

```
Conj[Lq · A] · q* == Conj[q · (Lq · A)] // Assert
```

⊢ True (after Simplify) ∎

```
Conj[Lq · A] · q* == Conj[Lq · A] · q* // Assert
```

⊢ True ∎

▼ L & R matrices components

```
Grid@{
  {"Lq", "Lq*", "Rq", "Rq*"},
  {Lq // QForm, Lq* // QForm, Rq // QForm, Rq* // QForm}
 }
```

L_q

$$\begin{pmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{pmatrix}$$

L_{q^*}

$$\begin{pmatrix} w & x & y & z \\ -x & w & z & -y \\ -y & -z & w & x \\ -z & y & -x & w \end{pmatrix}$$

R_q

$$\begin{pmatrix} w & -x & -y & -z \\ x & w & z & -y \\ y & -z & w & x \\ z & y & -x & w \end{pmatrix}$$

R_{q^*}

$$\begin{pmatrix} w & x & y & z \\ -x & w & -z & y \\ -y & z & w & -x \\ -z & -y & x & w \end{pmatrix}$$

▼ Rotation matrix Q components

```
Grid@{
  {"Qq == Lq·Rq* == Rq*·Lq", "Qq* == Lq*·Rq == Rq·Lq*"},
  {Simplify[Qq, Assumptions → |q|^2 == 1] // QForm,
   Simplify[Qq*, Assumptions → |q|^2 == 1] // QForm}
 }
```

$Q_q == L_q \cdot R_{q^*} == R_{q^*} \cdot L_q$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-2y^2-2z^2 & 2xy-2wz & 2(wy+xz) \\ 0 & 2(xy+wz) & 1-2x^2-2z^2 & -2wx+2yz \\ 0 & -2wy+2xz & 2(wx+yz) & 1-2x^2-2y^2 \end{pmatrix}$$

$Q_{q^*} == L_{q^*} \cdot R_q == R_q \cdot L_{q^*}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-2y^2-2z^2 & 2(xy+wz) & -2wy+2xz \\ 0 & 2xy-2wz & 1-2x^2-2z^2 & 2(wx+yz) \\ 0 & 2(wy+xz) & -2wx+2yz & 1-2x^2-2y^2 \end{pmatrix}$$

▼ Shoemake's rotation matrix from quaternion

Classical Shoemake's form of the rotation matrix from a quaternion is given as (see [2])

```
RotationMatrix4[q] // QForm
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-2(y^2+z^2) & 2xy-2wz & 2wy+2xz \\ 0 & 2xy+2wz & 1-2(x^2+z^2) & -2wx+2yz \\ 0 & -2wy+2xz & 2wx+2yz & 1-2(x^2+y^2) \end{pmatrix}$$

Scalar, vector and cross-product parts of L&R matrices

Scalar part

$$\frac{1}{2}\left(L_q + L_{q^*}\right) == \frac{1}{2}\left(R_q + R_{q^*}\right) == \left(\begin{array}{c|ccc} w & 0 & 0 & 0 \\ \hline 0 & w & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & w \end{array}\right) \text{// Assert}$$

⊢ True ∎

Vector part

$$\frac{1}{2}\left(L_{q^*} - R_q\right) == \frac{1}{2}\left(R_{q^*} - L_q\right) == \left(\begin{array}{c|ccc} 0 & x & y & z \\ \hline -x & 0 & 0 & 0 \\ -y & 0 & 0 & 0 \\ -z & 0 & 0 & 0 \end{array}\right) \text{// Assert}$$

⊢ True ∎

$$\frac{1}{2}\left(L_q - R_{q^*}\right) == \frac{1}{2}\left(R_q - L_{q^*}\right) == \left(\begin{array}{c|ccc} 0 & x & y & z \\ \hline -x & 0 & 0 & 0 \\ -y & 0 & 0 & 0 \\ -z & 0 & 0 & 0 \end{array}\right)^{\mathsf{T}} \text{// Assert}$$

⊢ True ∎

Cross product part

$$\frac{1}{2}\left(L_q - R_q\right) == \left(\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -z & y \\ 0 & z & 0 & -x \\ 0 & -y & x & 0 \end{array}\right) \text{// Assert}$$

⊢ True ∎

$$\frac{1}{2}\left(L_{q^*} - R_{q^*}\right) == \left(\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -z & y \\ 0 & z & 0 & -x \\ 0 & -y & x & 0 \end{array}\right)^{\mathsf{T}} \text{// Assert}$$

⊢ True ∎

$\mathbf{Q}\cdot\mathbf{Q} + 2\,\mathbf{Q}\cdot$(vector part) form.
Note that vector part is pure imaginary part of a quaternion.

```
ToQ@0 == q^2 + 2 q · ToQ@Im[q*] - |q|^2 // Assert
```

⊢ True ∎

$$\text{Assert}\Big[\; Q_q == L_q \cdot L_q + 2\,L_q \cdot \left(\begin{array}{c|ccc} 0 & x & y & z \\ \hline -x & 0 & 0 & 0 \\ -y & 0 & 0 & 0 \\ -z & 0 & 0 & 0 \end{array}\right)$$

$$== L_q \cdot L_q + 2\,L_q \cdot \left(\frac{1}{2}\left(R_{q^*} - L_q\right)\right)$$

$$== L_q \cdot R_{q^*}$$

$$\Big]$$

⊢ True (after Simplify) ∎

$$\frac{1}{2}\,L_{q^{-1}} \cdot \left(L_q \cdot L_q - Q_q\right) == \frac{1}{2}\left(L_q - R_{q^*}\right) \text{// Assert}$$

⊢ True (after Simplify) ∎

Equality of rotation matrix from quaternion and Euler-Rodrigues' (tensor) formula

Euler-Rodrigues' formula for the rotation matrix corresponding to a rotation by an angle θ about a fixed axis specified by the unit vector **u**.

Reference: http://mathworld.wolfram.com/RodriguesRotationFormula.html

```
EulerRodrigues[θ_, {x_, y_, z_}] :=
 Cos[θ] {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}} + (1 - Cos[θ]) {{x x, y x, z x}, {x y, y y, z y}, {x z, y z, z z}} + Sin[θ] {{0, -z, y}, {z, 0, -x}, {-y, x, 0}}
```

Prove that rotation matrix from quaternion (as defined in Quat.m package) is equivalent to Euler-Rodrigues' formula.

```
Block[{x, y, z, θ, q},
 q = ToQ$AngleAxis[θ, {x, y, z}];
 Assert[
  RotationMatrix[q] == EulerRodrigues[θ, {x, y, z}],
  Assumptions →
   -π ≤ θ ≤ π ⋀ x^2 + y^2 + z^2 == 1 ⋀ {x, y, z, θ} ∈ Reals
 ]
]
```

⊢ True (after Simplify) ∎

▼ Rotation matrix theorems

```
RM = RotationMatrix4[q];
RM // QForm
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1-2\left(y^2+z^2\right) & 2xy-2wz & 2wy+2xz \\ 0 & 2xy+2wz & 1-2\left(x^2+z^2\right) & -2wx+2yz \\ 0 & -2wy+2xz & 2wx+2yz & 1-2\left(x^2+y^2\right) \end{pmatrix}$$

```
Assert[
 RM == Qq == Lq · Rq* ⋀
  RM^T == Qq* == Rq*^T · Lq^T,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

```
Assert[
 RM · ToVector[q1]
       == Lq · Rq* · ToVector[q1]
       == ToVector[q · q1 · q*],
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

```
Assert[
 RM · A · RM^T
       == Q_q · A · Q_q*
       == L_q · R_q* · A · L_q* · R_q
       == ( q · ( q · A · q* )^T · q* )^T,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

```
Assert[
 I4
       == (q · I4 · q*) · I4 · (q · I4 · q*)^T
       == (L_q · R_q*) · I4 · (L_q · R_q*)^T
       == L_q · R_q* · I4 · R_q*^T · L_q^T
       == L_q · (R_q* · I4 · R_q*^T) · L_q^T,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

```
Assert[
 RM · A · RM^T
       == (q · I4 · q*) · A · (q · I4 · q*)^T
       == (L_q · R_q*) · A · (L_q · R_q*)^T
       == L_q · R_q* · A · R_q*^T · L_q^T
       == L_q · (R_q* · A · R_q*^T) · L_q^T,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

Proof that both $\mathbf{Q}_q$ and $\mathbf{R}_M$ are orthogonal matrices.

```
Assert[
 Q_q · Q_q^T
       == (L_q · R_q*) · (L_q · R_q*)^T
       == L_q · R_q* · (R_q*)^T · (L_q)^T
       == L_q · R_q* · R_q · L_q*
       == I4,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

```
Assert[
 Q_q · Q_q^T
       == Q_q · Q_q*
       == RM · RM^T
       == I4,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

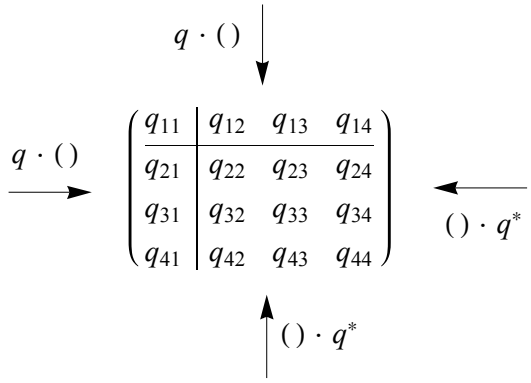
▼ Quaternionic meaning of R(*q*)·A·R(*q*)^T

The hint how quaternion multplication affects (moment of inertia or any other) tensor **A** can be seen from the earlier proven theorem

```
Assert[
 RM · A · RM^T == ( q · ( q · A · q* )^T · q* )^T,
 Assumptions → |q|^2 == 1
]
```

⊢ True (after Simplify) ∎

which shows that tensor **A** is rotated, at first, by rotating quaternions across each column $\mathbf{A}_c = q \cdot \mathbf{A} \cdot q^*$, and at second, by rotating quaternions across each row $\mathbf{A}_{rc} = \left(q \cdot (\mathbf{A}_c)^T \cdot q^*\right)^T$.



The meaning of the compound multiplications across columns and rows is extracting and affecting individual components of quaternions embedded from the previous rotations of the tensor. The moment of inertia tensor can be always reduced to its principal axes in diagonal form. In matrix representation of a quaternion, the diagonal of the matrix is constant and contains the scalar (real) part of the quaternion. (For any quaternion q there exist quaternion p such that qp is scalar.) The principal moment of inertia, on the other hand, has on its diagonal three different scalars in general case, which gives origin to three different quaternions that are embedded and shuffled across-columns/rows during rotations.

■ References

[1] Eberly, David H. with a contrib. by Shoemake, Ken - *Game Physics*, Morgan Kaufmann, 2004, Chapter 10 "Quaternions", pp. 507-544, http://books.google.se/books?id=a9SzFHPJ0mwC

[2] Shoemake, Ken - *Uniform random rotations*, in: D. Kirk (Ed.), *Graphics Gems III*, Academic Press, London, 1992, pp. 124-132, http://books.google.se/books?id=xmW_u3mQLmQC

[3] Shoemake, Ken - *Quaternions*, Department of Computer and Information Science, University of Pennsylvania, 2005-10-07, downloaded from C+MS course CS171 at Caltech: http://courses.cms.caltech.edu/cs171/quatut.pdf