Geometric Discrete Variational Dynamics — Part 1: Free Bodies

Brian Beckman

Dec 2023

Abstract

We report work-in-progress with conservative, geometric, discrete, variational integrators on Lie groups — modern techniques for simulation and control software. The current notebook successfully applies geometric integration to a benchmarking example — A non-isotropic top that spontaneously exhibits the intermediate-axis phenomenon pursuant to the *tennis-racket theorem*.[8] The geometric integrator **CGDVIE3** (conservative, geometric, discrete, variational integrator on SE(3)) conserves energy for this benchmark at least as well as Runge-Kutta-4 (**RK4**) over quaternions, but with a time step 3 times larger.

The long-term objective of this series of notebooks is to integrate the 4-degree-of-freedom inverted spherical pendulum (**4DISP**) with CGDVIE3. 4DISP is surprisingly difficult given that the 2-DOF inverted circular pendulum (2DICP) is an undergraduate exercise. Numerically, 2DICP is easily solved using any of the methods below that failed on 4DISP.[15] The final objective — 4DISP on CGDVIE3 — awaits further research outlined below. In this notebook, we exhibit intermediate results:

- RK4 on quaternions fails spectacularly 4DISP, even with painfully small, sub-millisecond time steps. The integrator is conclusively shown at fault; the dynamical model is correct.
- Rewriting 4DISP in pitch-cone-roll coordinates avoids singularity at the North pole, as with quaternions. We apply *Mathematica*'s `NDSolve` to the standard Euler-Lagrange equations. This works well, but does not export out of *Mathematica*. For our purposes, it furnishes ground truth for later developments.
- Next stop is a simple discrete Lagrangian integrator on pitch-cone-roll coordinates — Discrete Euler-Lagrange Equations (**DELE)**.[10] This works plausibly but disappoints. Like RK4, it requires fine time steps, making it too slow.
- Next stop is CGDVIE3. Coding it requires a deep swim in manifold theory, topic of other notebooks of mine. We get it working on the non-isotropic top. It is most promising for 4DISP, but we must formulate an adjoint representation for force in the cotangent bundle, and that work is not done yet.

Prior Failures

Documenting failed approaches is valuable because it can save time for other researchers.

Equations of Motion with Euler Angles

We implemented a full integrator, not shown here, for Guangyu Liu's 2007 dissertation.[7] We checked his equations and corrected several mistakes. We used *Mathematica*'s built-in numerical integrator `NDSolve`. In the initial test, we dropped the inverted pole from a shallow co-elevation angle: 5° from vertical. The integration proceeds plausibly for a few seconds, and then gains energy and angular momentum. It performs better than RK4 on our Tiny Dynamical Quaternion Library (**TDLQ**), but fails in the same way. Our RK4/TDLQ is easier to explain.

We also found that *Mathematica*'s `NDSolve` could not start at the North pole, even with pitch-cone-roll coordinates, which are not singular at the North pole. This latter failure is unexplained.

Due to the following factors, we abandoned Guangyu Liu's approach:

- Diagnosing the numerical and singularity issues above would take too long.
- An opaque integrator like `NDSolve` is not satisfactory in the long run because it cannot be ported out of *Mathematica*.
- Liu's equations of motion are large and complex — several full pages each for altitude-azimuth, pitch-cone-roll, and projected *x*-*y* Cartesian coordinates. It is time-consuming to check them, yet we must because we found mistakes. After the failure of the most intuitively promising of the three coordinate systems, pitch-cone-roll, it did not seem worth the effort to work through the other two, less-promising coordinate systems.
- It would have been worth pursuing if it had yielded a quick success for 4DISP. However, CGDVIE3 will most likely be superior for all problems, so we might as well implement it now rather than sink more time into an integration scheme that seems to have no long-term future.

The Future: Discretized Lagrangians on Lie Groups

Contemporary integrators embody the following features:

- The Lagrangian is discretized *before* applying the variational principle, resulting in discrete, reduced Euler-Poisson equations (**DREPEs**). Contrast discretized Euler-Lagrange equations, wherein the variational principle is applied *before* discretizing. Discretizing first is the contemporary approach, supported by many dissertations and papers from the shop of the late Jerrold E. Marsden and his students at Caltech (citations inline). A similar approach pertains to Hamiltonian formulations.
- The intrinsic configuration manifold of a rigid body is SE(3). Typical textbooks present rigid-body mechanics in Euler angles or quaternions. Euler angles inhabit the surface of a 3-toroid and

rotation quaternions inhabit the interior of a 3-ball or the surface of the unit 4-sphere. Neither manifold has a similar structure to SE(3). Integrating Euler angles invites gimbal lock. Integrating either Euler angles or quaternions invites non-conservation. Integrating DREPEs on the correct manifold automatically — in theory — conserves energy, momentum, and angular momentum pursuant to the discrete Noether's theorem.[9][10] We find difficulties in the concrete implementations below.

The theoretical arguments in favor of integrating DREPEs on the appropriate manifold are compelling. Integrators that do so are called “discrete variational integrators on Lie groups” or “geometric discrete variational integrators.” The particular integrator we need is **CGDVIE3** — conservative, geometric, discrete, variational integrator on SE(3).

References

1. Jack B Kuipers, *Quaternions and Rotation Sequences*, Princeton University Press, 1999.
2. Jeongseok Lee, Karen Liu, Frank C. Park, Siddartha S. Srinivasa, *A Linear-Time Variational Integrator for Multibody Systems*, 2018, https://arxiv.org/abs/1609.02898
3. Ari Stern, *Discrete Geometric Mechanics and Variational Integrators*, 2006, http://ddg.cs.columbia.edu/SIGGRAPH06/stern-siggraph-talk.pdf.
4. Ethan Eade, *Lie Groups for 2D and 3D Transformations*, 2017, http://ethaneade.com/lie.pdf.
5. NIST: *Digital Library of Mathematical Functions*, https://dlmf.nist.gov.
6. Brian C. Hall, *Lie Groups, Lie Algebras, and Representations*, Second Edition, 2015, Springer.
7. Guangyu Liu, *Modeling, stabilizing control and trajectory tracking of a spherical inverted pendulum*, PhD thesis, 2007, University of Melbourne, https://minerva-access.unimelb.edu.au/handle/11343/37225.
8. Wikipedia, *Tennis-Racket Theorem*, https://en.wikipedia.org/wiki/Tennis_racket_theorem.
9. Marin Kobilarov, Keenan Crane, Mathieu Desbrun, *Lie Group Integrators for Animation and Control of Vehicles*, 2009 (?).
10. Ari Stern, Mathieu Desbrun, *Discrete Geometric Mechanics for Variational Time Integrators*, 2006 (http://www.geometry.caltech.edu/pubs/SD06.pdf).
11. Kenth Engø, *On The BCH Formula in* $\mathfrak{so}(3)$, https://www.researchgate.net/profile/Kenth_Engo-Monsen2/publication/233591614_On _the _BCH-formula_in _so3/links/004635199177f69467000000/On-the-BCH-formula-in-so3.pdf.
12. Alexander Van-Brunt, Max Visser, *Explicit Baker-Campbell-Hausdorff formulae for some specific Lie algebras*, https://arxiv.org/pdf/1505.04505.pdf.
13. Wikipedia, *Baker-Campbell-Hausdorff Formula*, https://en.wikipedia.org/wiki/Baker% E2 %80 %93 Campbell % E2 %80 %93 Hausdorff_formula.
14. Jerrold E. Marsden, Tudor S. Ratiu, *Introduction to Mechanics and Symmetry*, Second Edition, 2002

15. Moylan, Andrew, *Stabilized Inverted Pendulum*, https://blog.wolfram.com/2011/01/19/stabilized-inverted-pendulum/, and *Stabilized n-Link Pendulum*, https://blog.wolfram.com/2011/03/01/stabilized-n-link-pendulum/
16. Paul R. Halmos, *Naive Set Theory*, 2015.
17. David Lovelock and Hanno Rund, *Tensors, Differential Forms, and Variational Principles*, 1975.
18. Ralph Abraham and Jerrold E. Marsden, *Foundations of Mechanics: 2nd Edition*, 1980.
19. Taeyoung Lee, Melvin Leok, and N. Harris McClamroch, *Discrete Control Systems*, https://arxiv.org/abs/0705.3868.
20. Taeyoung Lee, Melvin Leok, N. Harris McClamroch, *Global Formulations of Lagrangian and Hamiltonian Mechanics on Manifolds*, Springer, https://a.co/d/0lTbL9t.
21. Tristan Needham, *Visual Differential Geometry and Forms*, https://a.co/d/9hoKekz.
22. *xAct, Efficient tensor computer algebra for the Wolfram Language*, http://xact.es/index.html.
23. Blanco, Jose-Luis, *A tutorial on SE(3) transformation parameterizations and on-manifold optimization*, Technical Report #012010, Universidad de Malaga (https://jinyongjeong.github.io/Download/SE3/jlblanco2010geometry3d_techrep.pdf).
24. Marsden and West has exhaustive treatment of Noether's Theorem (http://www.cds.caltech.edu/~marsden/bib/2001/09-MaWe2001/MaWe2001.pdf).
25. Maxime Tournier, *Notes on Lie Groups*, https://maxime-tournier.github.io/notes/lie-groups.html.
26. John Baez, Javier P. Muniain, *Gauge Fields, Knots and Gravity*, (https://a.co/d/jdRnOU0)

Quaternions and RK4

- Section Abstract: This method promises to avoid gimbal lock inherent to Euler angles. It produces plausible results on several benchmarks, running quickly enough for interactive animation; and spontaneously exhibits the intermediate-axis effect, precession, and nutation. However, it fails dramatically on 4DISP with 10-msec time step, requiring microsecond granularity to conserve energy and momentum. Such failure is not unexpected, reading the references, but the drama is surprising — despite doing a good job on other problems, it's not even close for 4DISP. We are forced to find other integrators.

Tiny Quaternionic Dynamics Library (TQDL)

For those who know quaternions well, this section is straightforward code. For those who don't, I immodestly propose it as a good way to learn about them.

The most interesting thing, here, is a general-purpose RK4 integrator (highlighted below). It takes a dynamical variable, *v*, a function `vprime` that computes the time derivative of *v*, and a timestep `dt`;

and returns an updated value of *v*. It is suitable for streaming applications, that is, for producing animations on-the-fly, without storing intermediate states of *v*.

In our application, RK4 updates orientation quaternions.

Primitives

Most things in here have two names: one long and one short. The long names only remind me of the short ones, so this library could be squeezed to half a page.

Remember for the distant future that `rqb2sn` and `ωbn` call `rk4`. We’ll use them to propose roots for the discretized Lagrangian.

In[212]:=

```
<< Quaternions`
ClearAll[rq, rq0, ranv, ranθ, ranrq, rqw, vq, wrq, θrq,
  vrq, θvrq, qv, rv, rf, versor, random3Vector, randomAngleRad,
  randomVersor, versorFromTwistVector, realPart, twistVectorFromVersor,
  twistAngleFromVersor, twistAngleAndRealPartFromVersor,
  pureQuaternion, rotatedVector, vectorInRotatedFrame, normalize];

(* Be aware that rq_ is often used as a parameter (pattern variable),
so the same symbol, rq, can mean different things *)

versor := rq;

(* rq CAN take a zero vector as input. See documentation
 for "Normalize." Results are not valid rotation quaternions,
though this library allows them (more below). The error
 message "too few arguments given for Quaternion" is incorrect
 because "Sequence" expands into three arguments. *)

rq[θ_?NumberQ, v_List] :=
  Quaternion[Cos[θ / 2], Sequence @@ (Sin[θ / 2] Normalize[v])];
(* overload for four numeric arguments *)
rq[θ_?NumberQ, x_?NumberQ, y_?NumberQ, z_?NumberQ] := rq[θ, {x, y, z}];

(* It's best to have a canonical 0-rotation quaternion about a non-zero,
but arbitrary, axis. *)

rq0 = rq[0, {1, 0, 0}];

random3Vector := ranv;
ranv[] := RandomReal[{-1, 1}, 3];
```

```
randomAngleRad := ranθ;
ranθ[] := RandomReal[{0, 2 π}];

randomVersor := ranrq;
ranrq[] := rq[RandomReal[2 π], ranv[]];

versorFromTwistVector := rqw;
rqw[w_List] := rq[Norm[w], w];

realPart := vq;
vrq := vq;
vq[q_Quaternion] := List @@ q⟦2 ;; 4⟧;

twistVectorFromVersor := wrq;
wrq[rq_Quaternion] := 2 ArcCos[rq⟦1⟧] Normalize @ vq @ rq;

twistAngleFromVersor := θrq;
θrq[rq_Quaternion] := 2 ArcCos[rq⟦1⟧];

twistAngleAndRealPartFromVersor := θvrq;
θvrq[rq_Quaternion] := {θrq[rq], vrq[rq]};

pureQuaternion := qv;
(* ... another incorrect error message
  because "Sequence" expands to three arguments ... *)
qv[v_List] := Quaternion[0, Sequence @@ v];

rotatedVector := rv;
rv[rq_Quaternion, v_List] := vq[rq ** qv[v] ** rq*];

vectorInRotatedFrame := rf;
rf[rq_Quaternion, v_List] := vq[rq* ** qv[v] ** rq];

normalize[q_Quaternion] :=
 With[{n = Chop[Abs[q]]},
  If[n ⩵ 0.0, Quaternion[0, 0, 0, 0], (* ? should be "rq0" ? *)
   q / Abs[q]]]

ClearAll[drqb2sdt, dωbdt, rk4, ωbn, dVersorDt,
  dBodyAngVelDt, stepVersorFromBodyAngVel, stepAngVelBodyFrame];

(*The usual definition of drq/dt is
  qv[ω]**q/2. My angular velocity is in the body b-frame;
ω_s=qω_bq*is angvel in the inertial s-frame and dq/dt=1/2 ω_s q=1/2 qω_bq*q=1/2 q ω_b
   because q*q=1 for a versor by definition.*)

dVersorDt := drqb2sdt;
drqb2sdt[qb2s_Quaternion, ωb_List] := (qb2s / 2) ** qv[ωb];

dBodyAngVelDt := dωbdt;
dωbdt[ωb_List, mi_List, mii_List, τs_, rq_] :=
   mii.(rf[rq, τs] - ωb × (mi.ωb));
(* this integrator is very general, not specialized to TQDL *)
ClearAll[rk4];
rk4[v_, vprime_, dt_?NumberQ, args___] :=
   With[{k1 = dt * vprime[v, Sequence @@ {args}]},
    With[{k2 = dt * vprime[v + k1 / 2, Sequence @@ {args}]},
     With[{k3 = dt * vprime[v + k2 / 2, Sequence @@ {args}]},
      With[{k4 = dt * vprime[v + k3, Sequence @@ {args}]},
       v + (k1 + 2 k2 + 2 k3 + k4) / 6]]]];

(* I haven't seen the need for the call of '
 normalize' below in practice. It is a bit of paranoia. *)

stepVersorFromBodyAngVel := rqb2sn;
rqb2sn[rqb2snm1_Quaternion, ωbnm1_List, dt_?NumberQ] :=
   normalize@rk4[rqb2snm1, drqb2sdt, dt, ωbnm1];

stepAngVelBodyFrame := ωbn;
ωbn[ωbnm1_List, mib_List, miib_List, dt_?NumberQ, τs_, rq_] :=
   rk4[ωbnm1, dωbdt, dt, mib, miib, τs, rq];
```

Q: Quaternion From *ψ*, *θ*, *ϕ*

Mnemonics: *θ* looks like pitch, *ϕ* looks like roll, *ψ* looks like ‘y’ in ‘yaw.’

Page 206 of Kuipers.[1]

In[253]:=

```
yawQ[ψ_] := With[{α = ψ/2}, Quaternion[Cos[α], 0, 0, Sin[α]]];

pitchQ[θ_] := With[{β = θ/2}, Quaternion[Cos[β], 0, Sin[β], 0]];

rollQ[ϕ_] := With[{γ = ϕ/2}, Quaternion[Cos[γ], Sin[γ], 0, 0]];
```

For OBJECT ROTATIONS, compose these in the forward order (right-to-left). Reverse the order, left-to-right, for FRAME ROTATIONS:

In[256]:=

```
Q[ψ_, θ_, ϕ_] := rollQ[ϕ] ** pitchQ[θ] ** yawQ[ψ];
```

Yaw and roll should be within the ranges $-\pi$ to π. Pitch should in the range $-\pi/2$ to $\pi/2$. Problems occur near the endpoints of those ranges, especially for pitch.

Forced Motion for TQDL & RK4

The following few sections contain unit tests for TQDL & RK4. Note that we have a force model for TQDL & RK4, though we do not yet have one for CGDVIE3.

Global Constants

Used freely everywhere in this notebook.

Rectangular basis vectors, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{e}_3$, origin $\boldsymbol{o}$, acceleration of Earth’s gravity $\boldsymbol{g}$ in m/sec^2, and a default plot range that leaves some padding.

In[257]:=

```
ClearAll[e1, e2, e3, o, g, plotRg];
e1 = {1, 0, 0}; e2 = {0, 1, 0}; e3 = {0, 0, 1};
o = {0, 0, 0};
g = -9.81;
plotRg = {-1.25, 1.25};
```

Show Apparatus

Apparatus is a lightweight data type: a symbol with `DownValues` or an `Association` with some known string lookup keys.

`showApparatus` is for graphics only, embodying no interesting physics or numerics. Its code is repeated in several places in the notebook with minor variations. We did not take the effort to properly abstract it.

In[260]:=

```
ClearAll[myFont];
myFont[color_, size_ : 18] :=
  Style[#, color, Bold, size, FontFamily → "Courier New"] &;

ClearAll[showApparatus];
showApparatus[t_, ωb_, Lb_, ωs_, Ls_, Ib_, rq_, apparatus_] :=
(Module[{placeZ = 1}, With[{arrowDiameter = 0.02, sq = Sequence @@ θvrq[rq],
     displaceZ = -0.15},
    Show[{
      Graphics3D[{
        Text[myFont[Blue][
          "t = " <> ToString[NumberForm[t, {10, 2}]]], {-1, -1, placeZ}],

        placeZ += displaceZ;
        Text[myFont[Blue][
          "|L_s| = " <> ToString[NumberForm[Sqrt[Ls.Ls], {10, 4}]]],
         {-.975, -1, placeZ}],

        placeZ += displaceZ;
        Text[myFont[Blue][
          "|L_b| = " <> ToString[NumberForm[Sqrt[Lb.Lb], {10, 4}]]],
         {-.975, -1, placeZ}],

        placeZ += displaceZ;
        Text[myFont[Blue][
          "ω_b^T I_b ω_b/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Ib.ωb], {10, 4}]]],
         {-.95, -1, placeZ}],

        placeZ += displaceZ;
        Text[myFont[Blue][
          "ω_b^T L_b/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Lb], {10, 4}]]],
         {-.95, -1, placeZ}],

        placeZ += displaceZ;
        Text[myFont[Blue][
          "ω_s^T L_s/2 = " <> ToString[NumberForm[Sqrt[1/2 ωs.Ls], {10, 4}]]],
         {-.95, -1, placeZ}],

        Magenta, Arrow[Tube[{o, Ls}, arrowDiameter]],
        Text[myFont[Black, 12]["Inertial-Frame Ang Mom"],
         {-1.9, +0.2, 1}, Background → Magenta],

        Red, Arrow[Tube[{o, Lb}, arrowDiameter]],
        Text[myFont[Black, 12]["Body-Frame Ang Mom"],
         {-1.7, -0.1, 1}, Background → Red],

        Cyan, Arrow[Tube[{o, ωs / 10}, arrowDiameter * 1.10]],
        Text[myFont[Black, 12]["Inertial-Frame Ang Vel"],
         {-1.4, -0.4, 1}, Background → Cyan],

        Blue, Arrow[Tube[{o, ωb / 10}, arrowDiameter * 1.10]],
        Text[myFont[White, 12]["Body-Frame Ang Vel"],
         {-1.3, -0.7, 1}, Background → Blue],

        White, Rotate[apparatus["graphics primitives"], sq]}
    ]},
   Axes → True,
   PlotRange → ConstantArray[plotRg, 3],
   AxesLabel → myFont[Red] /@ {"X", "Y", "Z"},
   ImageSize → Large]]]);
```

Mathematica MomentOfInertia Is Not Moment of Inertia

Beware that *Mathematica*'s `MomentOfInertia` built-in returns quantities of fifth degree in the dimension of length. These quantities are actually the product of moment of inertia and volume. The following shows that the ratio of *Mathematica*'s `MomentOfInertia` and *Mathematica*'s `Volume` for a `Cylinder` object gives the expected result for moment of inertia. Factors of h and $\sqrt{h^2}$ cancel, of course, when $h > 0$, as stipulated. I don't know why *Mathematica* won't cancel them.

In[264]:=

```
With[{body = Cylinder[{{0, 0, -h / 2}, {0, 0, h / 2}}, r]},
 With[{ℐ = MomentOfInertia[body, Assumptions → {h > 0 ∧ r > 0}], 𝒱 = Volume[body]},
  <|"ℐ" → (ℐ // MatrixForm), "𝒱" → 𝒱,
   "ℐ/𝒱" → (ℐ / 𝒱 // FullSimplify // MatrixForm) |>]]
```

Out[264]=

$$\langle\left|\, \mathcal{I} \to \begin{pmatrix} \frac{\pi r^2 (h^4+3 h^2 r^2)}{12 \sqrt{h^2}} & 0 & 0 \\ 0 & \frac{\pi r^2 (h^4+3 h^2 r^2)}{12 \sqrt{h^2}} & 0 \\ 0 & 0 & \frac{1}{2} \sqrt{h^2}\, \pi r^4 \end{pmatrix},\right.$$

$$\left. \mathcal{V} \to \sqrt{h^2}\, \pi r^2,\ \mathcal{I}/\mathcal{V} \to \begin{pmatrix} \frac{1}{12}\left(h^2+3 r^2\right) & 0 & 0 \\ 0 & \frac{1}{12}\left(h^2+3 r^2\right) & 0 \\ 0 & 0 & \frac{r^2}{2} \end{pmatrix} \right|\rangle$$

Run Sim Forced Motion (TQDL & RK4)

The following simulates only the rotational component of rigid-body motion. It depends on the TQDL above. SE(3) will add the translational component in CGDVIE3 later. Initial conditions for angular velocity ω_b in the body frame b, for the initial orientation quaternion **rq**, and external forces and torques are chosen ad-hoc to produce a pleasing display.

In[265]:=

```
ClearAll[oneStepForcedRotationalMotion];
oneStepForcedRotationalMotion[ωb_, rq_, Ib_, Ibi_, dt_, τs_] :=
  With[{
    ωbNew = ωbn[ωb, Ib, Ibi, dt, τs, rq],
    rqNew = rqb2sn[rq, ωb, dt]},
   With[{LbNew = Ib.ωbNew},
    {ωbNew, rqNew, rv[rqNew, ωbNew], LbNew, rv[rqNew, LbNew]}]];

ClearAll[runSimForcedRotationalMotion];
runSimForcedRotationalMotion[
   apparatus_,
   ωbIn_ : {6., .01, 0},
   rqIn_ : rq[π / 4.0, {0, 1., 0}],
   fs_ : {0, 0, 0},
   τs_ : {0, 0, 0}] :=
  With[{ibibi = apparatus["moment of inertia"]},
   With[{Ib = ibibi⟦1⟧, Ibi = ibibi⟦2⟧},
    DynamicModule[
     {t = 0, dt = 0.03, ωb = ωbIn, ωs, Lb, Ls, rq = rqIn},
     Dynamic[t += dt;
      {ωb, rq, ωs, Lb, Ls} =
       oneStepForcedRotationalMotion[ωb, rq, Ib, Ibi, dt, τs];
      showApparatus[t, ωb, Lb, ωs, Ls, Ib, rq, apparatus]]]]];
```

Unit test

In[269]:=

```
oneStepForcedRotationalMotion[{6., .01, 0}, rq[π / 4.0, {0, 1., 0}],
 Sequence @@ dzhanybekhov["moment of inertia"], 0.03, {0, 0, 0}]
```

Out[269]=

```
{{6., 0.0100899, -0.00108325},
 Quaternion[0.920083, 0.0830369, 0.381273, -0.034395],
 {4.2419, 0.0102823, -4.24338}, {0.480952, 0.000832814, -4.29862 × 10^-6},
 {0.340086, 0.000833079, -0.340084}}
```

Demos of TQDL & RK4

Dzhanybekhov

A benchmark and standard test for any rigid-body simulator (search the web for “dzhanibekov effect youtube,” taking care to spell with their (correct) ‘k’ rather than our (incorrect) ‘kh.’).

Apparatus

The code here defines an apparatus, a symbol with three `DownValues`: *graphics primitives*, *mass*, and *moment of inertia in the body frame*.

In[270]:=

```
ClearAll[dzhanybekhov];
dzhanybekhov["graphics primitives"] :=
  With[{r1 = 0.125, r2 = 0.25, r3 = 0.0625},
   {Lighter[Red, 0.50], Sphere[e1, r1],
    Lighter[Purple, 0.50], Sphere[-1 e1, r1],
    Lighter[Green, 0.50], Sphere[e2, r2],
    Lighter[Yellow, 0.50], Sphere[-1 e2, r2],
    RGBColor[1, .71, 0], Opacity[0.125],
    Cylinder[{-e3, e3} / 10 000]}];
dzhanybekhov["mass"] = 1; (* not needed for this demo *)
dzhanybekhov["moment of inertia"] :=
  With[{
    M = 0.0801587 / 2,
    m = 0.0825397 / 2,
    mm = 0.00396825},
   With[{Ib = DiagonalMatrix[{2 M, 2 m, mm}]},
    With[{Ibi = Inverse@Ib},
     {Ib, Ibi}]]];
```

Free Motion Demo (TQDL & RK4)

With a time step of 10 milliseconds, this slowly leaks energy and magnitude of angular momentum over the course of hours. We shall see RK4 perform much worse on the inverted spherical pendulum.

Energy is the same in the body frame *b* and the inertial or space frame *s* because there is no translational motion, only rotation. Ditto for magnitudes of angular momentum.

Magenta is the angular momentum L_s in the inertial frame *s*. Red is the angular momentum L_b in the body frame *b*. Cyan is angular velocity ω_s in the inertial frame *s*. Blue is angular velocity ω_b in the body frame *b*. Three equivalent computations of kinetic energy are shown.

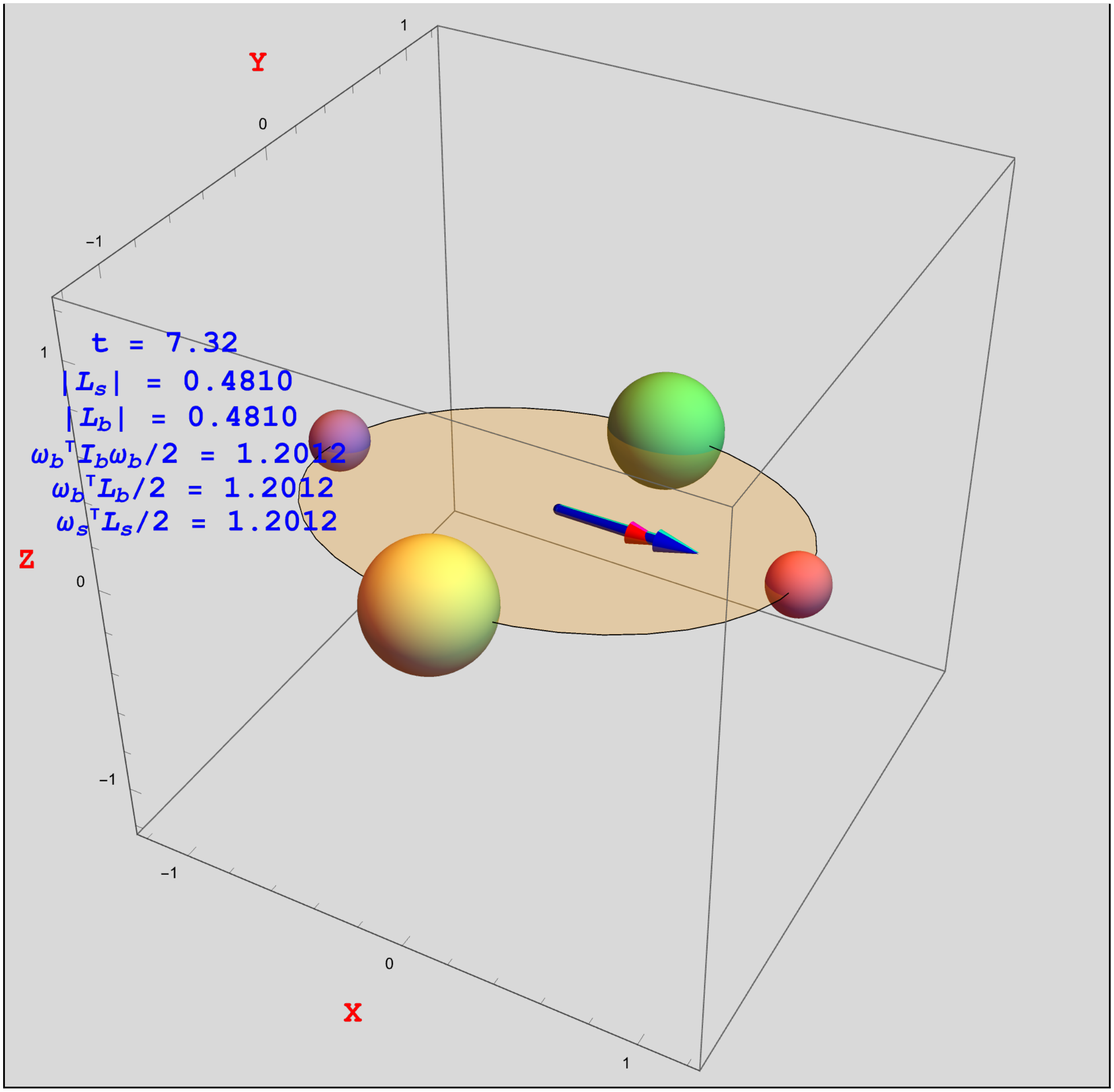
We can see the magenta arrow wiggle a little bit when the apparatus flips. This is a bad sign. It means that the computation of angular momentum in the inertial frame has some numerical trouble. It seems to spontaneously correct itself, but this trouble could be due for deeper investigation.

The arguments of `runSimForcedRotationalMotion` are the apparatus to display, the initial angular velocity in the body frame, and the initial orientation quaternion.

In[274]:=

```
runSimForcedRotationalMotion[dzhanybekhov, {6., 0., 0.01}, Q[0, 0, 0]]
```

Out[274]=

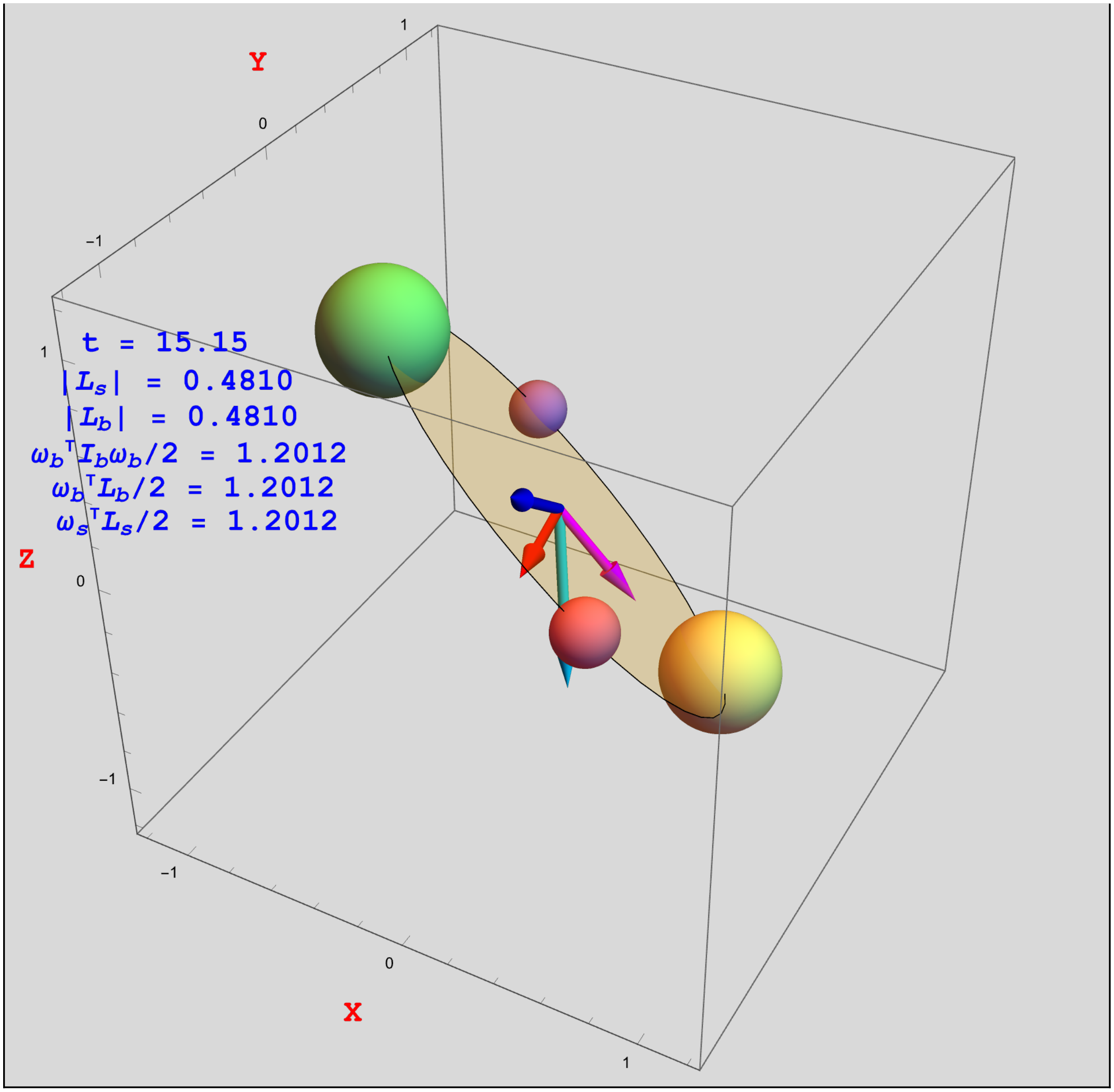

1
Y
0
-1
1
t = 7.32
$|L_s|$ = 0.4810
$|L_b|$ = 0.4810
$\omega_b{}^T I_b \omega_b/2$ = 1.2012
$\omega_b{}^T L_b/2$ = 1.2012
$\omega_s{}^T L_s/2$ = 1.2012
Z
0
-1
-1
0
X
1

Notice that the values of kinetic energy and magnitude of angular momentum do not depend on initial orientation.

In[275]:=

```
runSimForcedRotationalMotion[
 dzhanybekhov, {6., 0., 0.01}, rq[π / 4., {0., 1.0, 0.}]]
```

Out[275]=



$t = 15.15$

$|L_s| = 0.4810$

$|L_b| = 0.4810$

$\omega_b{}^T I_b \omega_b / 2 = 1.2012$

$\omega_b{}^T L_b / 2 = 1.2012$

$\omega_s{}^T L_s / 2 = 1.2012$

Forced Motion Demos (TQDL & RK4)

aCyl (apparatus)

In[276]:=

```
ClearAll[cyl];
With[{h = 0.5, r = 0.5, pill = 0.03},
  cyl["graphics primitives"] =
   {Opacity[0.750],
    Cylinder[{{0, 0, -h / 2}, {0, 0, h / 2}}, r],
    Black, Sphere[{r - 2 pill, 0, h / 2}, pill]};
  cyl["moment of inertia"] :=
   With[{m = 5}, With[{Ib = m * (MomentOfInertia[cyl["graphics primitives"]⟦2⟧] /
          Volume[cyl["graphics primitives"]⟦2⟧])},
     With[{Ibi = Inverse@Ib}, {Ib, Ibi}]]]];
```

Rod1 (apparatus)

In[278]:=

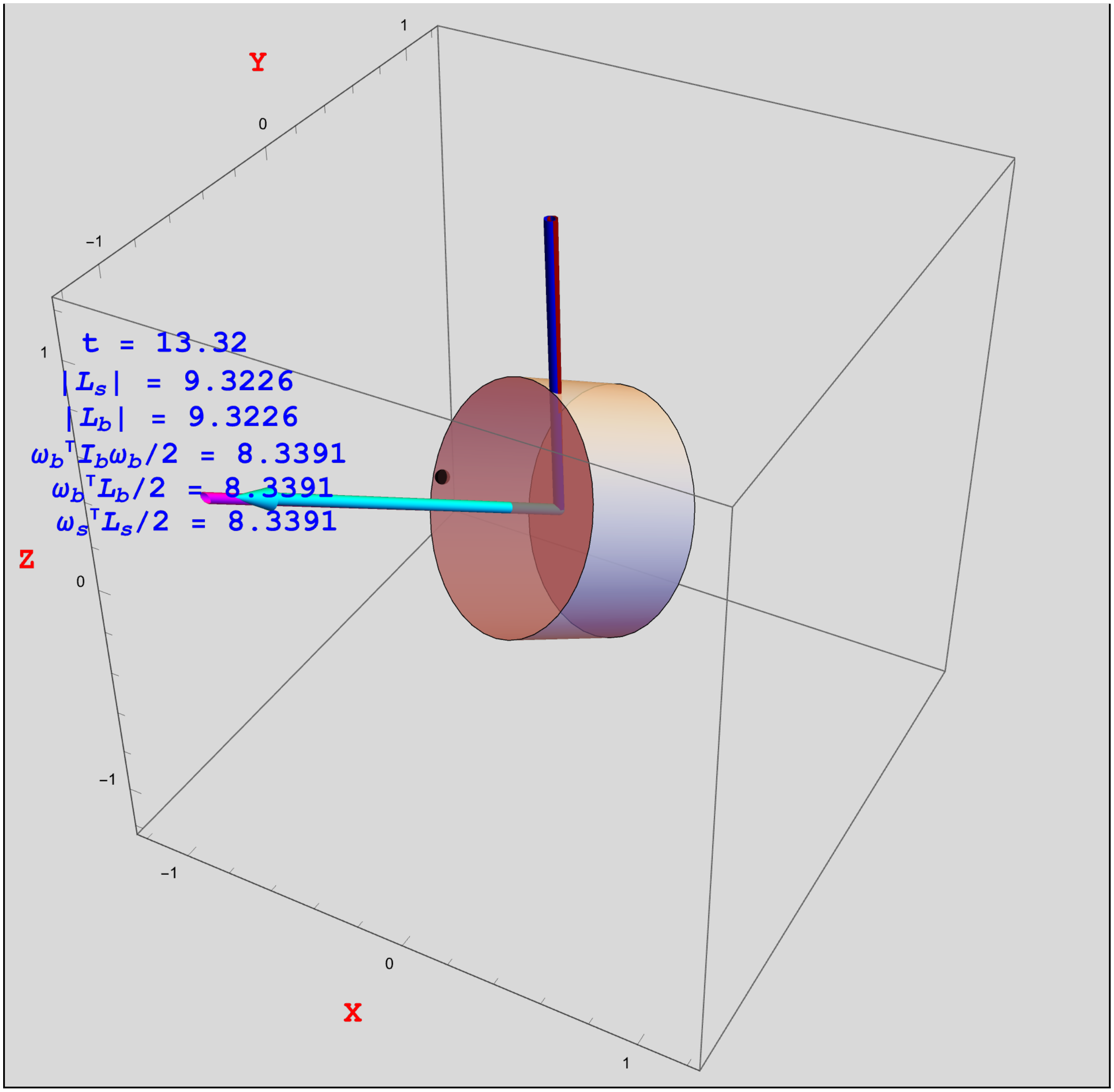
```
ClearAll[rod1];
With[{h = 2.4, r = 0.02},
  rod1["graphics primitives"] =
   {Opacity[0.750], Cylinder[{{0, 0, -h / 2}, {0, 0, h / 2}}, r]};
  rod1["moment of inertia"] :=
   With[{m = 5}, With[{Ib = m * (MomentOfInertia[cyl["graphics primitives"]⟦2⟧] /
          Volume[cyl["graphics primitives"]⟦2⟧])},
     With[{Ibi = Inverse@Ib}, {Ib, Ibi}]]]];
```

If our dynamics library is correct, this must show nutation and precession. It does not conserve energy because a constant torque is applied in the space frame. It speeds up intentionally. When the kinetic energy approaches 37, the precession and nutation become imperceptible.

In[280]:=

```
runSimForcedRotationalMotion[cyl,
 (* ωb(0) *){0, 0, 2 π},
 rq[0, {1, 1, 1}],
 (* f *){0, 0, 0},
 (* τs *){-0.4, -0.50, 0}]
```

Out[280]=

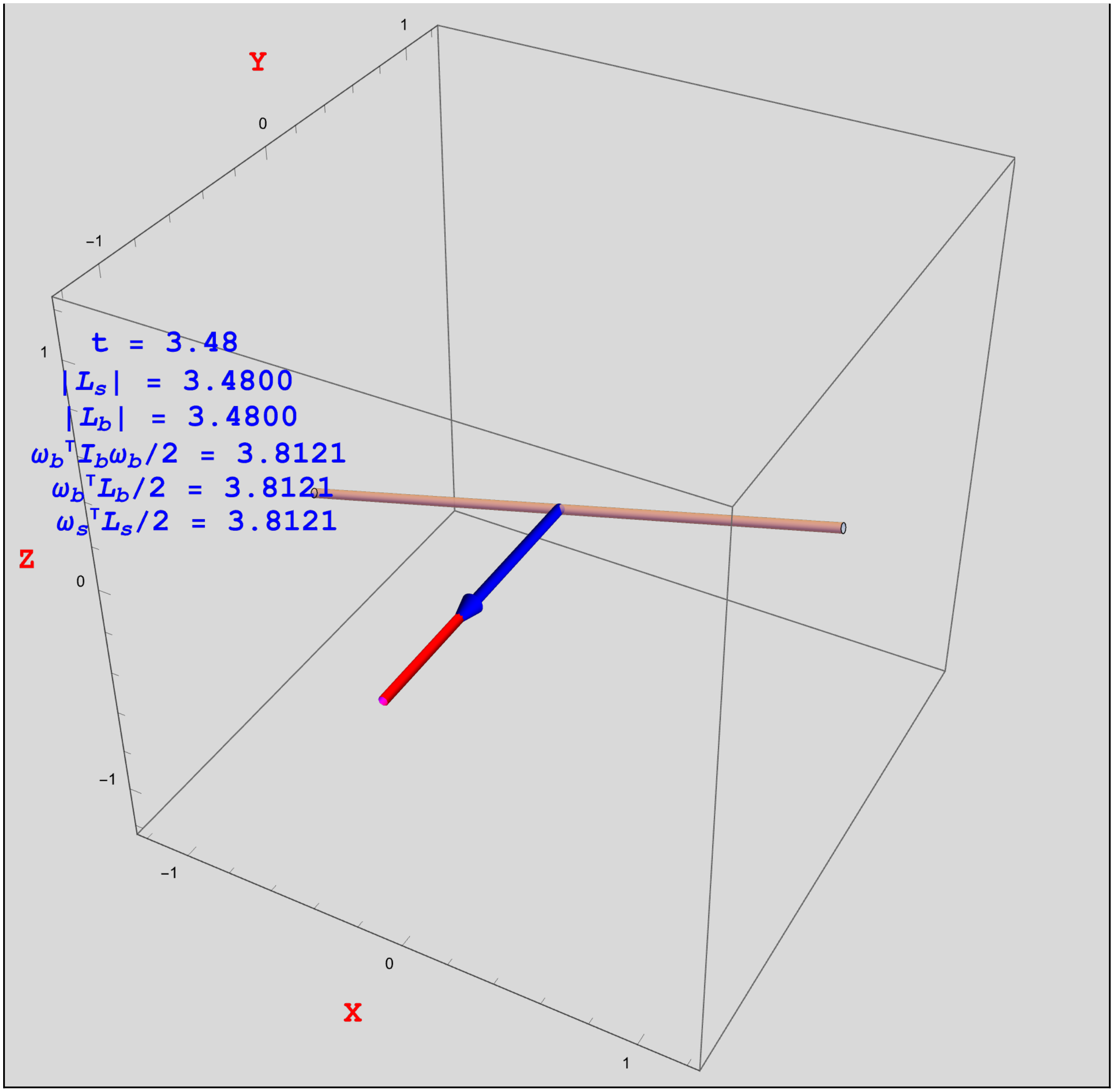

Y
1
0
−1
t = 13.32
$|L_s|$ = 9.3226
$|L_b|$ = 9.3226
$\omega_b^T I_b \omega_b/2$ = 8.3391
$\omega_b^T L_b/2$ = 8.3391
$\omega_s^T L_s/2$ = 8.3391
1
Z
0
−1
−1
0
X
1

Next is a forced rotating rod. If our dynamics library is correct, it must not exhibit precession and nutation.

In[281]:=

```
runSimForcedRotationalMotion[rod1, {0, 0, 0}, rq[0, {0., 1.0, 0}],
 {0, 0, 0}, {0, -1, 0}]
```

Out[281]=



Spherical Pendulum

We now build the graphics and apparatus for the inverted spherical pendulum on a massless cart, and integrate its motion with TQDL & RK4.

Rig (apparatus)

In[282]:=

```
ClearAll[rig];
With[{l = 1.2, mass = 1.0, d = 2.4, r = 0.02},
  With[{baton = Cylinder[{{0, 0, -l}, {0, 0, +l}}, r],
    batonCenterBodyFrame = {0, 0, l}},
   (* Immutable constant properties *)
   rig["graphics primitives"] = baton;
   rig["moment of inertia"] =
    With[{Ib = mass * (MomentOfInertia[baton] / Volume[baton])},
     With[{Ibi = Inverse@Ib}, {Ib, Ibi}]];
   rig["cb"] := batonCenterBodyFrame;
   rig["half length"] := l;
   rig["mass"] := mass;
  ]];
```

Axis Jack for Visualizing Orientation

Several overloads

In[284]:=

```
ClearAll[jack];
jack[0, opacity_ : 0.1, diameter_ : 0.01] := {
   Opacity[opacity],
   {Red, Arrow[Tube[{o, e1}], diameter]},
   {Darker[Green], Arrow[Tube[{o, e2}], diameter]},
   {Blue, Arrow[Tube[{o, e3}], diameter]},
   {Lighter[Cyan], Arrow[Tube[{o, -e1}], diameter]},
   {Magenta, Arrow[Tube[{o, -e2}], diameter]},
   {Yellow, Arrow[Tube[{o, -e3}], diameter]}};
jack[rq_Quaternion, opacity_ : 0.1, diameter_ : 0.01] := {
   Opacity[opacity],
   {Red, Arrow[Tube[{o, rv[rq, e1]}], diameter]},
   {Darker[Green], Arrow[Tube[{o, rv[rq, e2]}], diameter]},
   {Blue, Arrow[Tube[{o, rv[rq, e3]}], diameter]},
   {Lighter[Cyan], Arrow[Tube[{o, rv[rq, -e1]}], diameter]},
   {Magenta, Arrow[Tube[{o, rv[rq, -e2]}], diameter]},
   {Yellow, Arrow[Tube[{o, rv[rq, -e3]}], diameter]}};
jack[ωHat_List, opacity_ : 0.1, diameter_ : 0.01] := {
   Opacity[opacity],
   {Red, Arrow[Tube[{o, ωHat.e1}], diameter]},
   {Darker[Green], Arrow[Tube[{o, ωHat.e2}], diameter]},
   {Blue, Arrow[Tube[{o, ωHat.e3}], diameter]},
   {Lighter[Cyan], Arrow[Tube[{o, ωHat.-e1}], diameter]},
   {Magenta, Arrow[Tube[{o, ωHat.-e2}], diameter]},
   {Yellow, Arrow[Tube[{o, ωHat.-e3}], diameter]}};
```

nfm, vfm, qfm, font: Number-Formatting

`nfm` For numbers, `vfm` for vectors, and `qfm` for quaternions

In[288]:=

```
ClearAll[nfm, vfm, qfm, font];
nfm[num_, dig_ : 10, dec_ : 4] := ToString[NumberForm[Chop@num, {dig, dec}]];
vfm[vec_] := ToString[NumberForm[#, {7, 4},
      NumberSigns → {"-", " "}, NumberPadding → {" ", "0"}] & /@ Chop@vec];
qfm[q_Quaternion] := ToString[Map[NumberForm[#, {7, 4},
      NumberSigns → {"-", " "}, NumberPadding → {" ", "0"}] &, Chop@q, {1}]];
font = myFont[Blue, 12];
```

ISSUE: Accumulating Energy and Angular Momentum

In the body frame, gravitation points upward with a point of application at the bottom of the baton. The vector from the center of gravity to the point of application is $-c_s$, the torque is

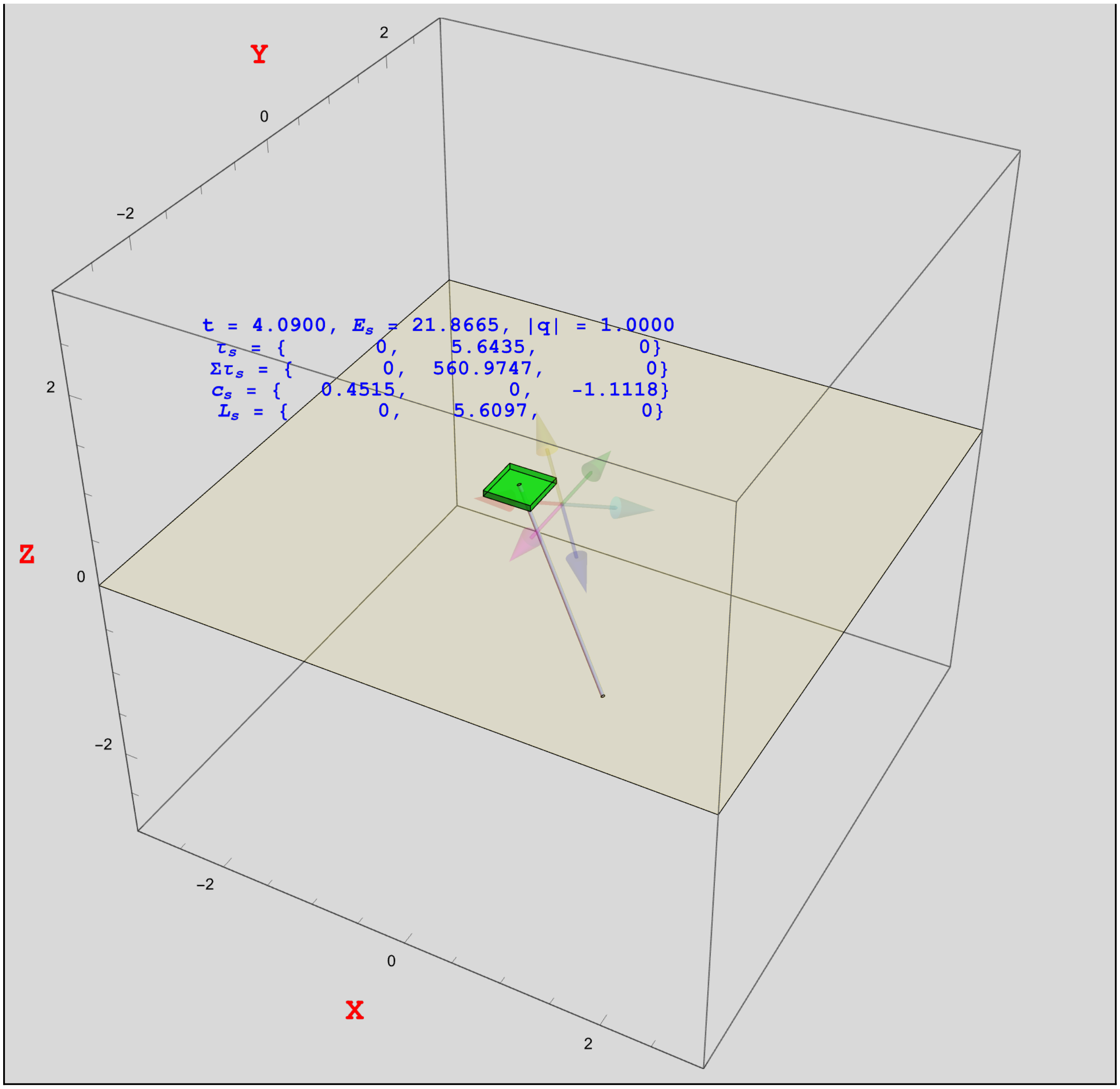
$(-c_s \times m\,|g|\,e_3) = (m\,|g|\,e_3 \times c_s)$, with × the 3D vector cross product. The gravitational force vector is applied at the same point with the same signs. The integration step size is 10 milliseconds, as before.

This rapidly accumulates kinetic energy and average angular momentum. In fact, the dropped pendulum immediately goes all the way around with a time step of 10 msec, an unphysical result. The time step must be reduced by a factor of 100 to prevent such swinging around. At that time step, the animation is intolerably slow (try it by changing `dt`, if you have 10 minutes or so to devote to watching it). In any event, a real pendulum with a frictionless mount would not accumulate energy or angular momentum.

In[549]:=

```
With[{epsilon = 0.0001, dt = 0.01, h = 1 / 15., w = 1 / 4., vu = 3.},
 With[{ztxt = 3, xtxt = 0, ytxt = -2, znl = 0.2},
  With[{kart = Cuboid[{-w, -w, epsilon}, {w, w, epsilon + h}],
    floor = Polygon[{{-vu, -vu, 0}, {vu, -vu, 0}, {vu, vu, 0}, {-vu, vu, 0}}],
    axisLabelStyle =
     text ↦ Style[text, Red, Bold, 18, FontFamily → "Courier New"],
    Ib = rig["moment of inertia"]⟦1⟧, Ibi = rig["moment of inertia"]⟦2⟧,
    cb = rig["cb"], rq0 = Q[0, 10 °, 0 * -0.5 °]},
   DynamicModule[
    {rq = rq0, ωb = o, ωs = o, Lb = o, Ls = o, pos = o, ps = o, force = o, τs = o, t = 0,
     cs = rv[rq0, cb], θv = θvrq[rq0],
     renderPos = o, rotarg1 = 0, rotarg2 = o, Στs = o},
    Dynamic[
     t += dt;
     τs = ((rig["mass"] Abs[g] e3 + force) × (cs));
     Στs += τs;
     cs = rv[rq, cb];
     renderPos = -cs⟦1⟧ e1 - cs⟦2⟧ e2;
     (* TODO: pos? Risk z drift. *)
     θv = θvrq[rq];
     rotarg1 = θv⟦1⟧;
     rotarg2 = If[o === Chop[θv⟦2⟧], e1, θv⟦2⟧];
     {ωb, rq, ωs, Lb, Ls} =
      oneStepForcedRotationalMotion[ωb, rq, Ib, Ibi, dt, τs];
     Graphics3D[{
        Text[font["t = " <> nfm@t <> ", E_s = " <> nfm@(ωs^T.Ls / 2 - g cs⟦3⟧) <>
           ", |q| = " <> nfm@Abs[rq]], {xtxt, ytxt, ztxt}],
        Text[font["τ_s = " <> vfm@τs], {xtxt, ytxt, ztxt - znl}],
        Text[font["Στ_s = " <> vfm@Στs], {xtxt, ytxt, ztxt - 2 znl}],
        Text[font["c_s = " <> vfm@cs], {xtxt, ytxt, ztxt - 3 znl}],
        Text[font["L_s = " <> vfm@Ls], {xtxt, ytxt, ztxt - 4 znl}],
        jack[rq], {Yellow, Opacity[.3 / 4], floor},
        {Green, Opacity[.6], Translate[kart, renderPos]},
        {White, Opacity[.75], Translate[Translate[
           Rotate[rig["graphics primitives"], rotarg1, rotarg2],
           cs], renderPos + (epsilon + h) e3]}},
      ImageSize → Large, Axes → True,
      AxesLabel → axisLabelStyle /@ {"X", "Y", "Z"},
      PlotRange → {{-vu, vu}, {-vu, vu}, {-vu, vu}}]  ]  ]  ]]]
```

Out[549]=



```
t = 4.0900, E_s = 21.8665, |q| = 1.0000
 τ_s = {         0,    5.6435,         0}
Στ_s = {         0,  560.9747,         0}
 c_s = {    0.4515,         0,   -1.1118}
 L_s = {         0,    5.6097,         0}
```

Addressing the Mystery

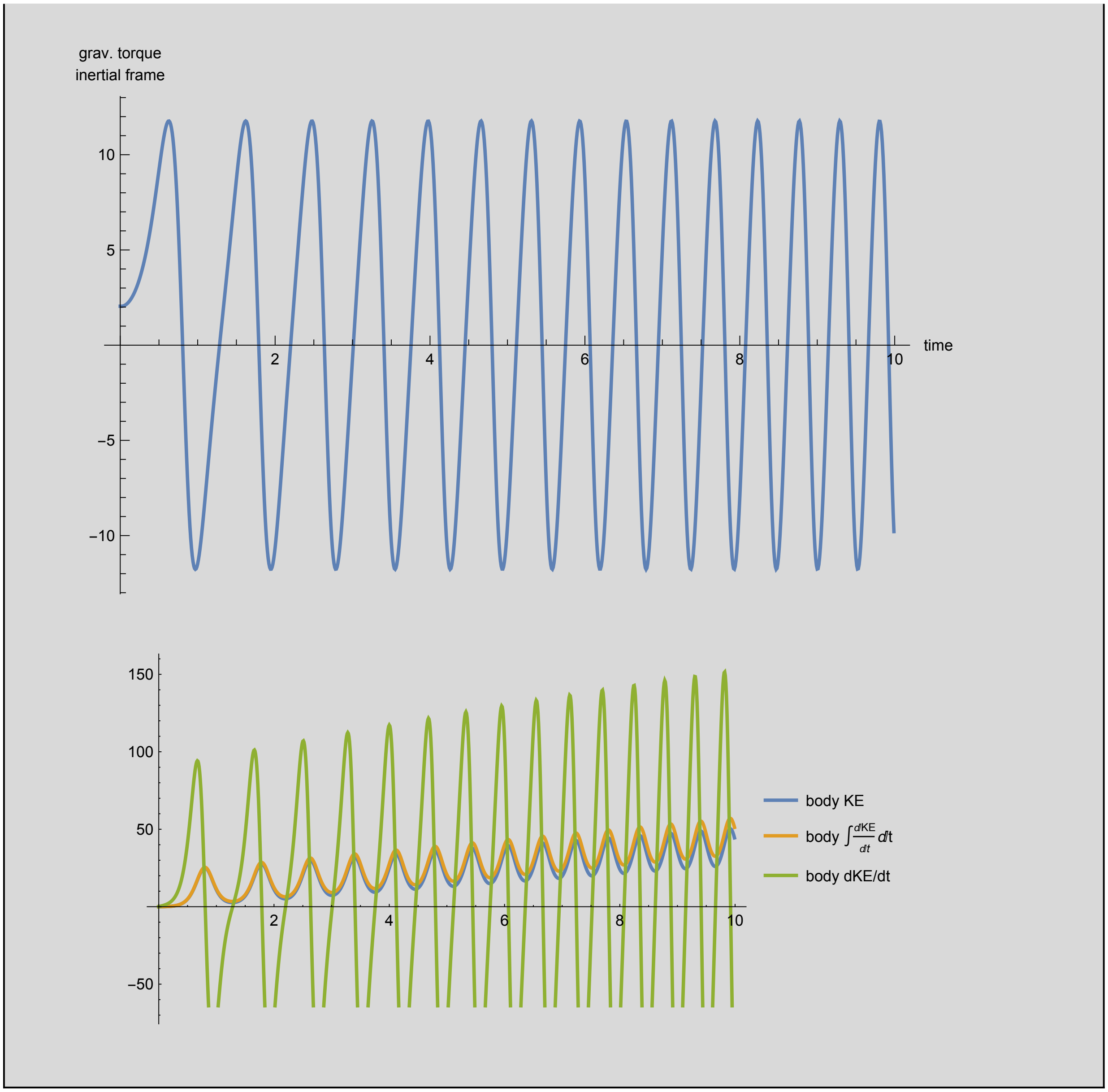
Plot $(d\omega_b/dt)\cdot L_b$, rotational “work.” It should integrate up to rotational kinetic energy, versus time.

In[294]:=

```
With[{epsilon = 0.0001,
  (* Adjust here ~~> *)dt = 0.01, nMax = 1000,
  h = 1 / 15., w = 1 / 4., vu = 3.},
 With[{ztxt = 6, xtxt = 0, ytxt = -2, znl = 0.2},
  With[{kart = Cuboid[{-w, -w, epsilon}, {w, w, epsilon + h}],
    floor = Polygon[{{-vu, -vu, 0}, {vu, -vu, 0}, {vu, vu, 0}, {-vu, vu, 0}}],
    axisLabelStyle =
     text ↦ Style[text, Red, Bold, 18, FontFamily → "Courier New"],
    Ib = rig["moment of inertia"]⟦1⟧, Ibi = rig["moment of inertia"]⟦2⟧,
    cb = rig["cb"], rq0 = Q[0, 10 °, 0 * -0.5 °]},
   Module[
    {rq = rq0, ωb = o, ωs = o, Lb = o, Ls = o, pos = o, ps = o, force = o, τs = o, t = 0,
     cs = rv[rq0, cb], θv = θvrq[rq0], Στs = o},
    Module[{
      n = 0, tn = ConstantArray[0, nMax],
      τn = ConstantArray[0, nMax], Στsn = ConstantArray[0, nMax],

      ωbLast = ωb, dωbdt = o, dEbrotdt = 0, dEbrotdtn = ConstantArray[0, nMax],
      Ebrotn = ConstantArray[0, nMax],
      ΣdtdEbrotdt = 0, ΣdtdEbrotdtn = ConstantArray[0, nMax],

      ωsLast = ωs, dωsdt = o, dEsrotdt = 0, dEsrotdtn = ConstantArray[0, nMax],
      Esrotn = ConstantArray[0, nMax],
      ΣdtdEsrotdt = 0, ΣdtdEsrotdtn = ConstantArray[0, nMax]},

     For[(n = 1; t = 0), (n ≤ nMax), (n++),
      τs = ((rig["mass"] Abs[g] e3 + force) × (cs));
      tn⟦n⟧ = t; τn⟦n⟧ = τs;
      Στs += τs * dt;
      Στsn⟦n⟧ = Στs;
      cs = rv[rq, cb];
      {ωb, rq, ωs, Lb, Ls} =
       oneStepForcedRotationalMotion[ωb, rq, Ib, Ibi, dt, τs];
      (* ---------------- energy stats, body frame ---------------- *)
      dωbdt = (ωb - ωbLast) / dt;
      dEbrotdt = dωbdt.Lb;
      dEbrotdtn⟦n⟧ = dEbrotdt;
      Ebrotn⟦n⟧ = ωb.Lb / 2;
      ΣdtdEbrotdt += (ωb - ωbLast).Lb;
      ΣdtdEbrotdtn⟦n⟧ = ΣdtdEbrotdt;
      ωbLast = ωb;
      (* ---------------- energy stats, space frame ---------------- *)
      dωsdt = (ωs - ωsLast) / dt;
      dEsrotdt = dωsdt.Lb;
      dEsrotdtn⟦n⟧ = dEsrotdt;
      Esrotn⟦n⟧ = ωs.Ls / 2;
      ΣdtdEsrotdt += (ωs - ωsLast).Ls;
      ΣdtdEsrotdtn⟦n⟧ = ΣdtdEsrotdt;
      ωsLast = ωs;
```

```
      (* -------------------------------------------------- *)
      t += dt];
     GraphicsColumn[{
       ListLinePlot[{{tn, #[[2]] & /@ τn}ᵀ}, AxesLabel →
         {"time", "grav. torque\ninertial frame"}, ImageSize → Large],
       ListLinePlot[{{tn, Ebrotn}ᵀ, {tn, ΣdtdEbrotdtn}ᵀ, {tn, dEbrotdtn}ᵀ},
        PlotLegends → {"body KE", "body ∫ⅆKE/ⅆt ⅆt", "body dKE/dt"},
        ImageSize → Medium](*,
       ListLinePlot[{{tn,Esrotn}ᵀ,{tn,ΣdtdEsrotdtn}ᵀ,{tn,dEsrotdtn}ᵀ}]*)
      }]
     (*Quiet@ListLinePlot[{{tn,#[[2]]&/@Στsn}ᵀ,{tn,#[[2]]&/@τn}ᵀ}]*)
    ]]  ]]]
```

Out[294]=

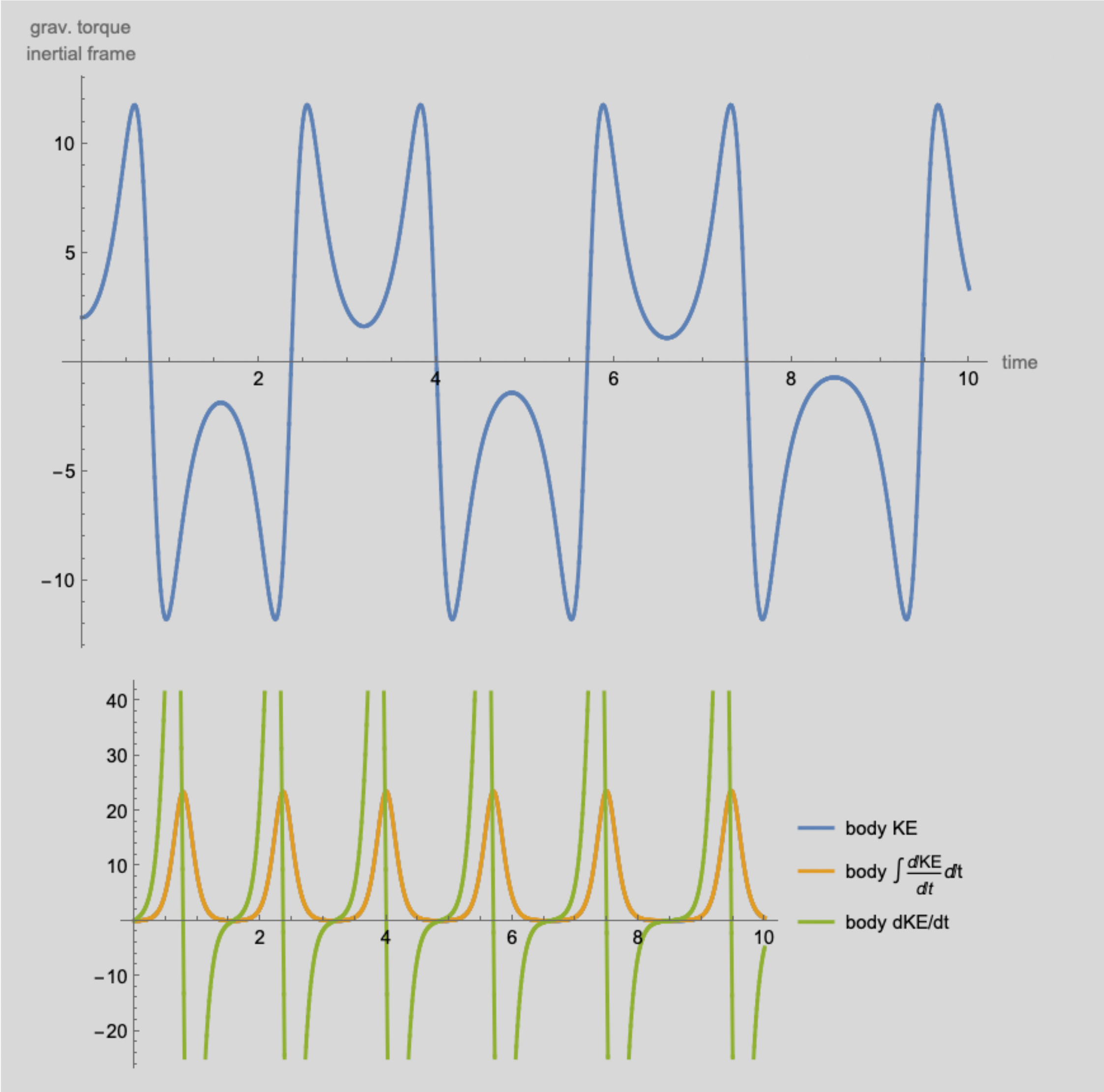


Notice the magnitude of the gravitational torque is not qualitatively what we'd expect. It should be zero when the pole is upright or dangling. On a swing-up arc, the torque should go through a "bounce" or "whiplash" as the sliding cart takes some of the lever arm away, thus have two maxima at zenith before falling.

The plots above show steadily increasing angular velocity and kinetic energy, plus a mismatch between kinetic energy and the integral of its derivative, which should always be equal.

But if I decrease d*t* by a factor of 100 to 100 *μ*sec, I get something much more sensible. It's visibly not conservative over ten simulated seconds, but it shows the expected torque whiplash, plus the body KE and the integral of its derivative coincide within the resolution of the plot. This takes several minutes to run, so I save only a screenshot.

In[295]:=



;

Conclusion:

The integrator is guilty.

Because decreasing the time increment results in physically plausible behavior and in better energy conservation, we deem the dynamics correct.

Stern-Desbrun Symplectic

Integrator

- Section Abstract: This integrator produces plausible results for a small time step, but, in general, is too slow and unstable. We leave further investigation to the future, preferring to invest time in CGDVIE3. In particular, we have not profiled the code and we have not explored its poor energy conservation, which is supposed to be automatic via the discrete Noether's Theorem. In the offing, for ground truth, we produce a good numerical integrator via *Mathematica*'s `NDSolve`. This integrator is not a primary objective of this work, but serves only to check others.

Recast the problem in Lagrangian form. What are good generalized coordinates and velocities of the pendulum? The obvious choice, spherical coordinates, has a singularity at the North pole, right where we want the baton to linger when get to controlling it. At the North pole, the longitude can be any number. A better choice of coordinates are pitch and an aerobatic, coordinated-cone roll (not axis roll!). These angles are co-latitude δ and an angular displacement η at a clockwise right angle to the great-circle arc of co-latitude. For any non-zero δ, spinning η makes a non-great circle on the surface around a pair of poles on the Equator. The demonstration below makes this visually clear. These poles are not as catastrophic as the North and South poles. *Mathematica*'s integrator produce interpolation functions that fly right through them. Both δ and η are zero at the North pole. η has two singularities on the Equator, when δ is 90° or 270°. But that situation is better than a singularity at the North pole because the baton will fly through these poles at the Equator, not linger at them.

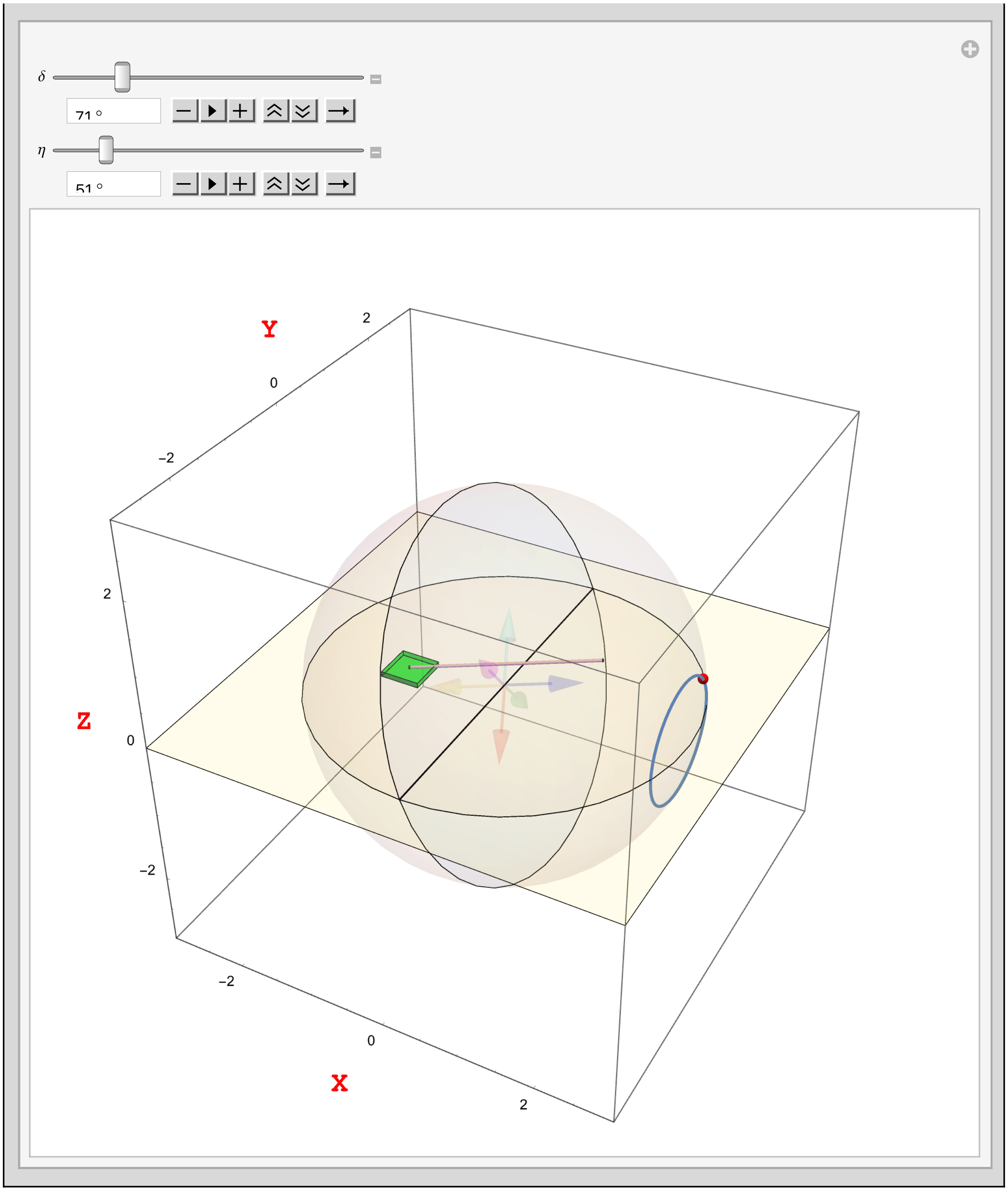
The actual physical system is represented by the red ball at the center of mass of the baton, swinging around the origin. The sliding cart is a visual fiction, useful later when we control the baton. The Lagrangian, here, does not account for translational energy imparted by the cart, only rotational energy. We fix that later.

Demonstration of the Coordinate System

In[296]:=

```
With[{epsilon = 0.0001, h = 1 / 15., w = 1 / 4.,
   vu = 3., o = {0, 0, 0}, thin = {0, 0, .0001}, cb = rig["cb"]},
 With[{kart = Cuboid[{-w, -w, epsilon}, {w, w, epsilon + h}], s = 2,
    floor = Polygon[{{-vu, -vu, 0}, {vu, -vu, 0}, {vu, vu, 0}, {-vu, vu, 0}}]},
  With[{
     c3 = Cylinder[{o, thin}, s cb⟦3⟧], b3 = Sphere[o, s cb⟦3⟧],
     t3x = Tube[{-s cb⟦3⟧ e2, s cb⟦3⟧ e2}, .01],
     axisLabelStyle =
      text ↦ Style[text, Red, Bold, 18, FontFamily → "Courier New"]},
    Manipulate[
     DynamicModule[{renderPos = o, Στs = o, cs, rotfn},
      cs = RotationMatrix[-η, e1].RotationMatrix[δ, e2].cb;
      renderPos = -cs⟦1⟧ e1 - cs⟦2⟧ e2;
      rotfn = RotationTransform[-η, e1] @* RotationTransform[δ, e2];
      Show[{
        Graphics3D[{
          GeometricTransformation[jack[0], rotfn],
          {Black, t3x},
          (*{White,Opacity[1./2],Cone[{RotationMatrix[δ,e2].cb,o},1]},*)
          {White, Opacity[.7 / 8], c3, b3,
           GeometricTransformation[c3, RotationTransform[δ, e2]]},
          {Red, Sphere[s cs, 1 / 16.]},
          {Yellow, Opacity[.3 / 4], floor},
          {Green, Opacity[.6], Translate[kart, renderPos]},
          {White, Opacity[.75],
           Translate[
            Translate[
             GeometricTransformation[rig["graphics primitives"], rotfn],
             cs], renderPos + (epsilon + h) e3]}},
         ImageSize → Large,
         Axes → True, AxesLabel → axisLabelStyle /@ {"X", "Y", "Z"},
         PlotRange → {{-vu, vu}, {-vu, vu}, {-vu, vu}}],
        ParametricPlot3D[
         s RotationMatrix[-ha, e1].RotationMatrix[δ, e2].cb, {ha, 0, 2 π}]}]],
     {{δ, 10 °}, 0 °, 360 °, 1 °, Appearance → {"Open"}},
     {{η, 0 °}, 0 °, 360 °, 1 °, Appearance → {"Open"}}]]]]
```

Out[296]=



δ 71 °

η 51 °

Dynamical Set-Up

Center of mass

In[297]:=

```
ClearAll[cm];
(cm[{δ_, η_}] := RotationMatrix[-η, e1].RotationMatrix[δ, e2].{0, 0, c_z});
cm[{δ, η}] // MatrixForm
```

Out[299]//MatrixForm=

```
⎛      Sin[δ] c_z      ⎞
⎜ Cos[δ] Sin[η] c_z    ⎟
⎝ Cos[δ] Cos[η] c_z    ⎠
```

z-Height

The potential energy depends only on the z-height of the CM:

In[300]:=

```
ClearAll[zh];
zh[{δ_, η_}] := -cm[{δ, η}]⟦3⟧;
zh[{δ, η}]
```

Out[302]=

```
-Cos[δ] Cos[η] c_z
```

Potential Energy

We’ll need symbolical and numerical variations on the theme of energy. The symbolical variations support *Mathematica*’s magic calculus with parametric functions like *δ*[t], *η*[t], *δ*’[t], and *η*’[t]. That magic yields the equations of motion almost effortlessly. I learned this from Andrew Moylan’s Reference 15. The numerical forms support graphics and discrete calculations later. For the numerical forms, we simply replace parametric functions with unbound symbols via numRules.

numRules

In[303]:=

```
ClearAll[numRules];
numRules = {δ[t] → δ, η[t] → η, δ'[t] → Dδ, η'[t] → Dη, m → 1, c_z → 1.2};
```

In[305]:=

```
ClearAll[Vnum, V];
V[{δ_, η_}] := m g zh[{δ, η}]; V[{δ, η}]
(* Notice not :=, SetDelayed, but =, Set *)
Vnum[{δ_, η_}] = ((V[{δ, η}]) /. numRules);
Vnum[{δ[t], η[t]}]
```

Out[306]=

$$9.81\, m\, \text{Cos}[\delta]\, \text{Cos}[\eta]\, c_z$$

Out[308]=

$$11.772\, \text{Cos}[\delta[t]]\, \text{Cos}[\eta[t]]$$

Velocity Vector

The kinetic energy depends on the velocity vector, squared. We begin some symbolical magic, here, with *Mathematica*’s `D` operator.

In[309]:=

```
ClearAll[v];
(v[{δ_, η_}] := D[cm[{δ, η}], t]);
v[{δ[t], η[t]}] // FullSimplify // MatrixForm
```

Out[311]//MatrixForm=

$$\begin{pmatrix} \text{Cos}[\delta[t]]\, c_z\, \delta'[t] \\ c_z\, (-\text{Sin}[\delta[t]]\, \text{Sin}[\eta[t]]\, \delta'[t] + \text{Cos}[\delta[t]]\, \text{Cos}[\eta[t]]\, \eta'[t]) \\ -c_z\, (\text{Cos}[\eta[t]]\, \text{Sin}[\delta[t]]\, \delta'[t] + \text{Cos}[\delta[t]]\, \text{Sin}[\eta[t]]\, \eta'[t]) \end{pmatrix}$$

Kinetic Energy

We’ll need symbolical and numerical variations on this theme.

In[312]:=

```
ClearAll[Tnum, T];
(T[{δ_, η_}] := 1/2 m v[{δ, η}].v[{δ, η}]);
T[{δ[t], η[t]}] // FullSimplify
(* hand-constructed, for checking *)
Tnum[{δ_, η_}, {Dδ_, Dη_}] := 1/2 (1.2)^2 (Dδ^2 + Cos[δ]^2 Dη^2);
Tnum[{δ_, η_}, {Dδ_, Dη_}] := 1/2 (1.2)^2 (Dδ^2 + (1/2 (1 + Cos[2 δ])) Dη^2);
(* Set =, not SetDelayed := *)
Tnum[{δ_, η_}, {Dδ_, Dη_}] = (T[{δ[t], η[t]}] /. numRules // FullSimplify);
Tnum[{δ, η}, {Dδ, Dη}]
```

Out[314]=

```
1/2 m c_z^2 (δ′[t]^2 + Cos[δ[t]]^2 η′[t]^2)
```

Out[318]=

```
0.72 Dδ^2 + 0.36 Dη^2 + 0.36 Dη^2 Cos[2 δ]
```

Lagrangian

We don’t need `Lnum` for `NDSolve` and the equations of motion in this section, but we do need it for the discretized Lagrangian, L_d, later.

In[319]:=

```
ClearAll[Lnum, L];
(L[{δ_, η_}] := T[{δ, η}] - V[{δ, η}]);
L[{δ[t], η[t]}] // FullSimplify
Lnum[{δ_, η_}, {Dδ_, Dη_}] := Tnum[{δ, η}, {Dδ, Dη}] - Vnum[{δ, η}];
Lnum[{δ, η}, {Dδ, Dη}]
```

Out[320]=

```
0.25 m c_z (-39.24 Cos[δ[t]] Cos[η[t]] + c_z (2. δ′[t]^2 + (1. + 1. Cos[2 δ[t]]) η′[t]^2))
```

Out[322]=

```
0.72 Dδ^2 + 0.36 Dη^2 + 0.36 Dη^2 Cos[2 δ] - 11.772 Cos[δ] Cos[η]
```

Leqns

Mathematica has symbolical magic for the Euler-Lagrange equations. I’ve solved a lot of problems like this.

In[323]:=

```
With[{Lcrds = {δ[t], η[t]}},
  With[{Lvels = D[Lcrds, t]},
   With[{Lncnf = {0, 0}}, (* non-conservative forces *)
    Leqns = Flatten@MapThread[
       {q, v, f} ↦ D[D[L[Lcrds], v], t] - D[L[Lcrds], q] == f,
       {Lcrds, Lvels, Lncnf}]]]] // FullSimplify
```

Out[323]=

```
{m c_z (-9.81 Cos[η[t]] Sin[δ[t]] + c_z (Cos[δ[t]] Sin[δ[t]] η′[t]^2 + δ″[t])) == 0,
 m Cos[δ[t]] c_z
   (-9.81 Sin[η[t]] + c_z (-2. Sin[δ[t]] δ′[t] η′[t] + Cos[δ[t]] η″[t])) == 0}
```

State-Space Form

In[324]:=

```
ClearAll[stEqns];
stEqns =
 Equal @@@ (Solve[Leqns, {δ''[t], η''[t]}] // FullSimplify // Chop // Part[#, 1] &)
```

Out[325]=

```
                     9.81 Cos[η[t]]
{δ″[t] == Sin[δ[t]] (―――――――――――――― - 1. Cos[δ[t]] η′[t]^2),
                          c_z
          9.81 Sec[δ[t]] Sin[η[t]]
 η″[t] == ―――――――――――――――――――――――― + 2. Tan[δ[t]] δ′[t] η′[t]}
                    c_z
```

Ground-Truth Solution with Initial Conditions

So we can check the Stern-Desbrun integrator later, let’s let *Mathematica* do its magical solution. We’ll regard that as ground truth. Even though it ends up pretty good, it’s not the end of the story because it’s opaque. We can’t easily see inside, so we can’t easily write a version in C or Python that does a good job. The Stern-Desbrun integrator, like our old RK4, is something we could code up in another programming language. That’s how we like to use *Mathematica*: as an executable design language for algorithms that we later code up in other programming languages for wider deployment.

Solve the system for tLim = 25 simulated seconds. Feel free to change this parameter. The solver is fast.

In[326]:=

```
ClearAll[tLim, initialConditions, stSolns];
tLim = 25;
initialConditions = {δ[0] == 10 °, η[0] == 5 °, δ'[0] == 0, η'[0] == 0};
stSolns = NDSolve[Append[stEqns, initialConditions] /. {cz → 1.2},
  {δ[t], η[t], δ'[t], η'[t]}, {t, 0, tLim}]
```

Out[329]=

```
{{δ[t] → InterpolatingFunction[ ⊞ Domain: {{0., 25. }}
                                    Output: scalar ][t],

  η[t] → InterpolatingFunction[ ⊞ Domain: {{0., 25. }}
                                   Output: scalar ][t],

  δ′[t] → InterpolatingFunction[ ⊞ Domain: {{0., 25. }}
                                    Output: scalar ][t],

  η′[t] → InterpolatingFunction[ ⊞ Domain: {{0., 25. }}
                                    Output: scalar ][t]}}
```

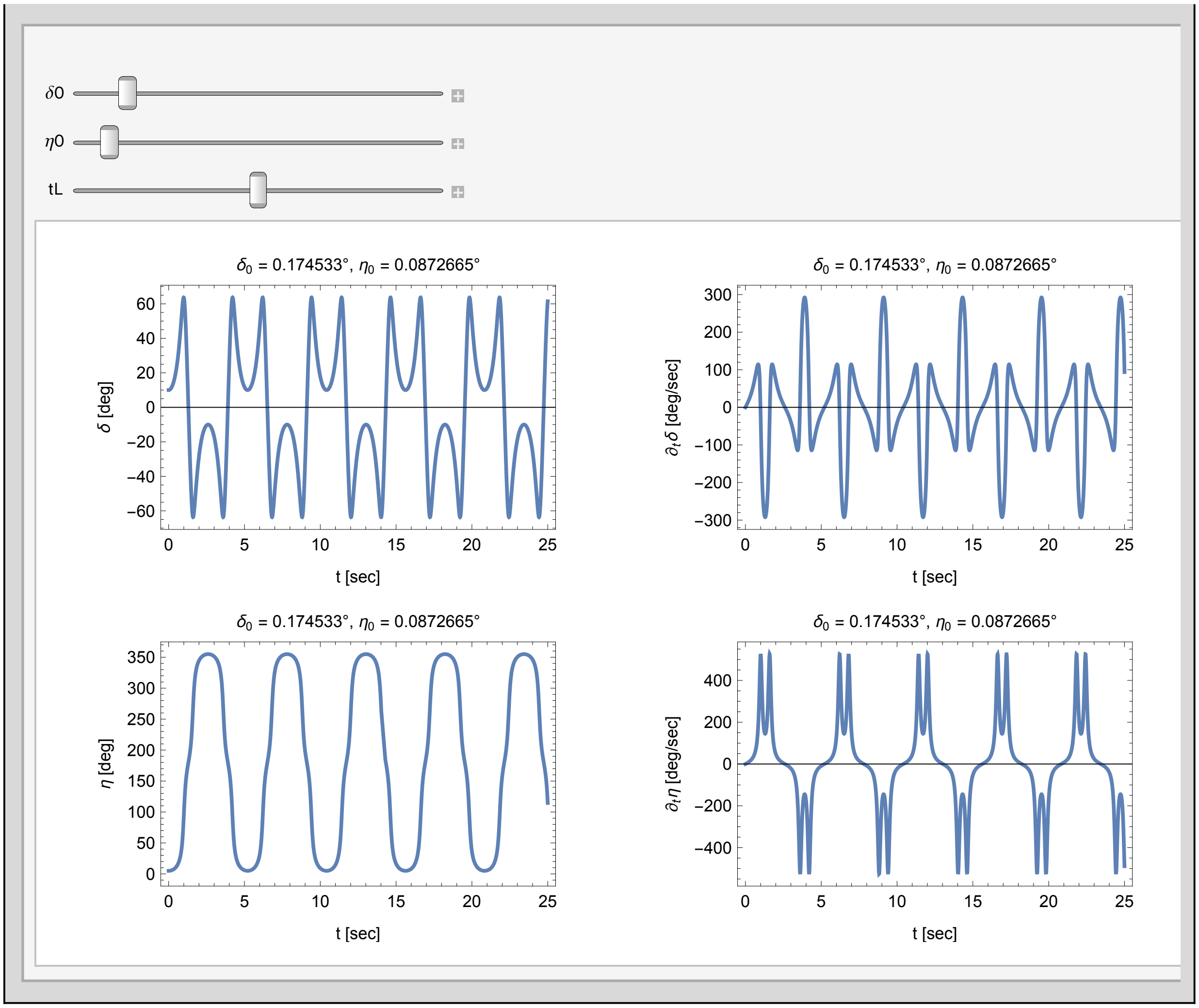
Plot the Solutions

The solutions look pretty reasonable. I won’t go over all the physical intuitions in my nodding head. I consider the animation to be the most valuable validation that we’ve got this mostly right. More discussion follows the animation.

In[330]:=

```
Manipulate[DynamicModule[{
    ics = {δ[0] == δ0, η[0] == η0, δ'[0] == 0, η'[0] == 0},
    solns, δx, δDotx, ηx, ηDotx, flt},
  solns = NDSolve[Append[stEqns, ics] /. {c_z → 1.2},
    {δ[t], η[t], δ'[t], η'[t]}, {t, 0, tL}];
  δx = solns⟦1, 1, 2⟧;
  δDotx = solns⟦1, 3, 2⟧;
  ηx = solns⟦1, 2, 2⟧;
  ηDotx = solns⟦1, 4, 2⟧;
  flt = "δ_0 = " <> ToString[δ0] <> "°, η_0 = " <> ToString[η0] <> "°";
  GraphicsGrid[{
    {Plot[δx / °, {t, 0, tL},
      FrameLabel → {{"δ [deg]", ""}, {"t [sec]", flt}}, Frame → True],
     Plot[δDotx / °, {t, 0, tL},
      FrameLabel → {{"∂_tδ [deg/sec]", ""}, {"t [sec]", flt}}, Frame → True]},
    {Plot[ηx / °, {t, 0, tL},
      FrameLabel → {{"η [deg]", ""}, {"t [sec]", flt}}, Frame → True],
     Plot[ηDotx / °, {t, 0, tL},
      FrameLabel → {{"∂_tη [deg/sec]", ""}, {"t [sec]", flt}}, Frame → True]}}]],
 {{δ0, 10. °}, 0., 90. °}, {{η0, 5. °}, 0, 90. °}, {{tL, 25}, 0, 50}]
```

Out[330]=



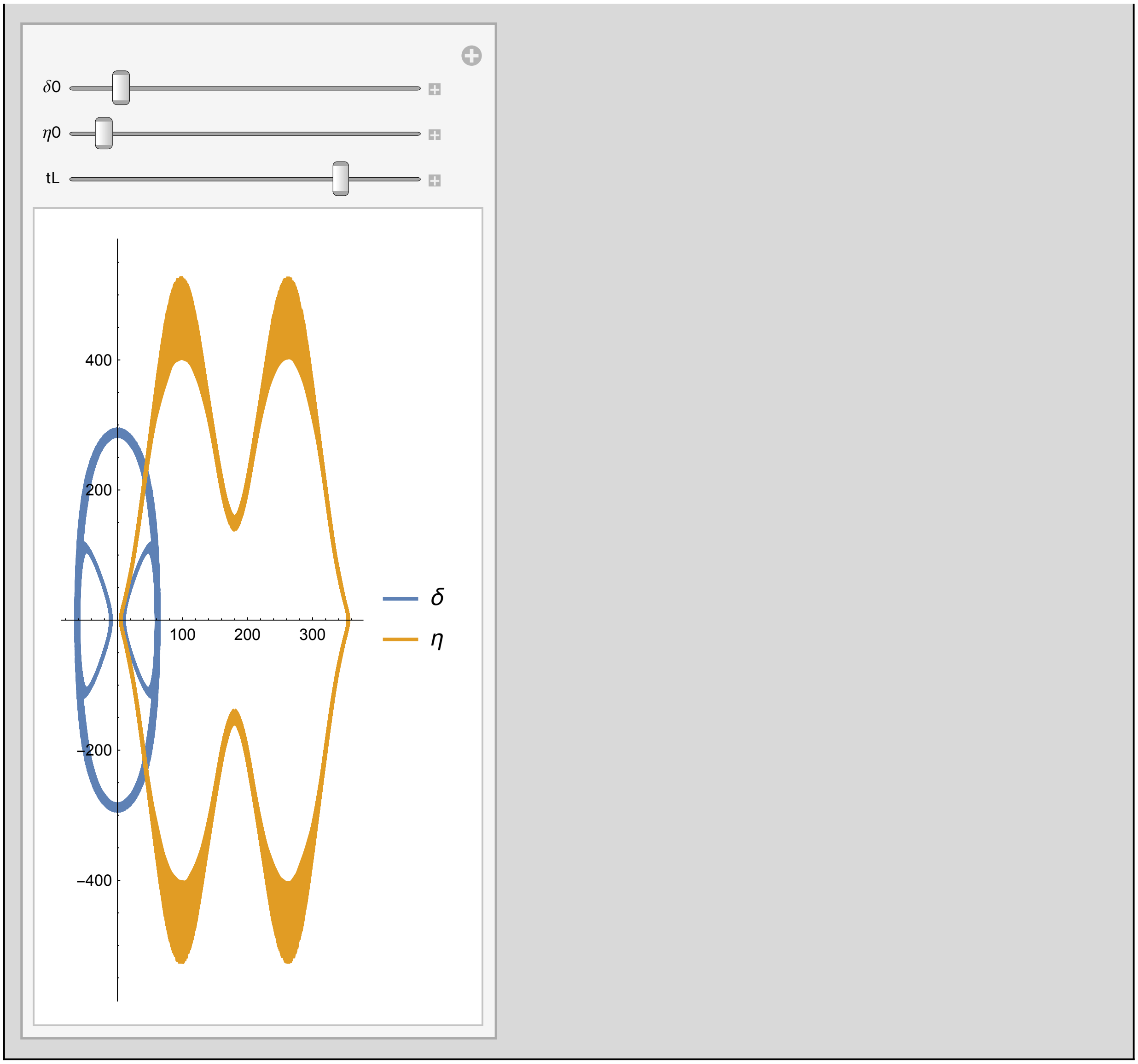
Phase Portrait

Plot the angles versus their velocities, proportional to their Hamiltonian conjugate momenta. If the solution were conservative, these phase portraits would be thin lines. When the time constant is long (400 seconds, here), however, they thicken up, showing that the solution is slightly non-conservative, but it’s much better than our RK4 on Quaternions. This is going to be a good ground-truth.

In[331]:=

```
Manipulate[DynamicModule[{
    ics = {δ[0] == δ0, η[0] == η0, δ'[0] == 0, η'[0] == 0}, solns, δx, δDotx, ηx, ηDotx},
   solns = NDSolve[Append[stEqns, ics] /. {c_z → 1.2},
     {δ[t], η[t], δ'[t], η'[t]}, {t, 0, tL}];
   δx = solns⟦1, 1, 2⟧;
   δDotx = solns⟦1, 3, 2⟧;
   ηx = solns⟦1, 2, 2⟧;
   ηDotx = solns⟦1, 4, 2⟧;
   ParametricPlot[{{δx, δDotx} / °, {ηx, ηDotx} / °},
    {t, 0, tL}, PlotLegends → {"δ", "η"}]],
  {{δ0, 10. °}, 0, 90. °}, {{η0, 5 °}, 0, 90. °}, {{tL, 400}, 0, 500}]
```

Out[331]=



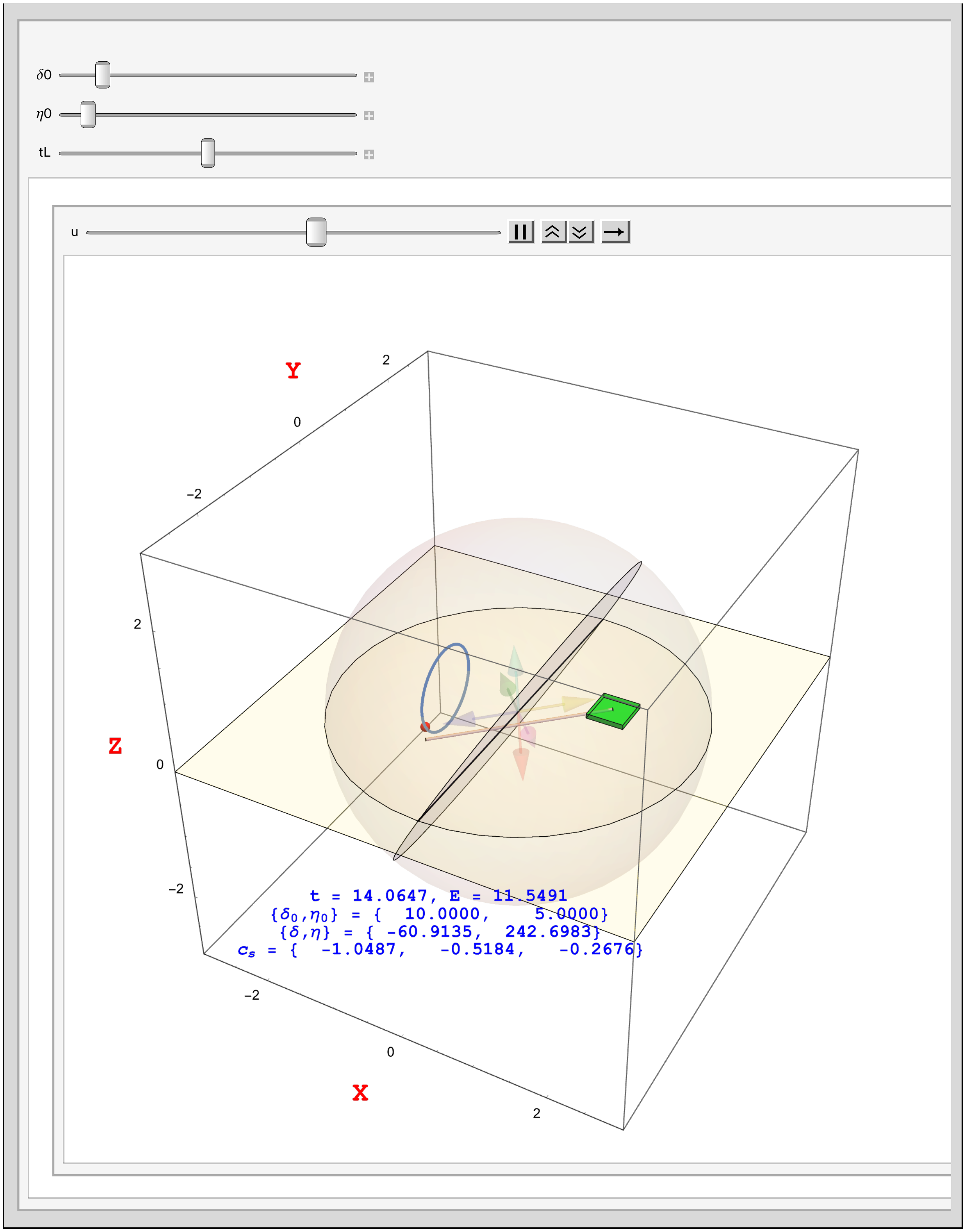
Ground-Truth Animation

This is pretty convincing. Energy is well conserved. If we were satisfied with an integrator that works only inside *Mathematica*, we'd be just about done. But we want an integrator we can code up in any programming language. Next stop is to do as well with a discrete Stern-Desbrun integrator, but we'll find that difficult.

In[332]:=

```
With[{epsilon = 0.0001, h = 1 / 15., w = 1 / 4., vu = 3.},
 With[{ztxt = -1.5, xtxt = 0, ytxt = -2, znl = 0.3},
  With[{kart = Cuboid[{-w, -w, epsilon}, {w, w, epsilon + h}],
    floor = Polygon[{{-vu, -vu, 0}, {vu, -vu, 0}, {vu, vu, 0}, {-vu, vu, 0}}],
    axisLabelStyle =
     text ↦ Style[text, Red, Bold, 18, FontFamily → "Courier New"],
    cb = rig["cb"], thin = {0, 0, .0001}, s = 2, o = {0, 0, 0}},
   With[{c3 = Cylinder[{o, thin}, s cb[[3]]],
     b3 = Sphere[o, s cb[[3]]], t3x = Tube[{-s cb[[3]] e2, s cb[[3]] e2}, .01]},
    Manipulate[DynamicModule[{
        ics = {δ[0] == δ0, η[0] == η0, δ'[0] == 0, η'[0] == 0},
        solns, δx, Dδx, ηx, Dηx, flt},
      solns = NDSolve[Append[stEqns, ics] /. {c_z → 1.2},
        {δ[t], η[t], δ'[t], η'[t]}, {t, 0, tL}];
      δx = solns[[1, 1, 2]];
      Dδx = solns[[1, 3, 2]];
      ηx = solns[[1, 2, 2]];
      Dηx = solns[[1, 4, 2]];
      Animate[
       Module[{renderPos = o, cs, rotfn},
        cs = RotationMatrix[-ηx /. t → u, e1].RotationMatrix[δx /. t → u, e2].cb;
        renderPos = -cs[[1]] e1 - cs[[2]] e2;
        rotfn = RotationTransform[-ηx /. t → u, e1]@*
          RotationTransform[δx /. t → u, e2];
        Show[{Graphics3D[{
            Text[font["t = " <> nfm@u <>
                ", E = " <> nfm@(Tnum[{δx /. t → u, ηx /. t → u}, {Dδx /. t → u,
                      Dηx /. t → u}] + Vnum[{δx /. t → u, ηx /. t → u}])],
             {xtxt, ytxt, ztxt}],
            Text[
             font["{δ_0,η_0} = " <> vfm@{δ0 / °, η0 / °}], {xtxt, ytxt, ztxt - znl}],
            Text[font["{δ,η} = " <> vfm@{(δx /. t → u) / °, (ηx /. t → u) / °}],
             {xtxt, ytxt, ztxt - 2 znl}],
            Text[font["c_s = " <> vfm@cs], {xtxt, ytxt, ztxt - 3 znl}],
            {Black, t3x},
            {White, Opacity[.7 / 8], c3, b3,
             GeometricTransformation[c3, RotationTransform[δx /. t → u, e2]]},
            GeometricTransformation[jack[0], rotfn],
            {Red, Sphere[cs, 1 / 16.]},
            {Yellow, Opacity[.3 / 4], floor},
            {Green, Opacity[.6], Translate[kart, renderPos]},
            {White, Opacity[.75], Translate[Translate[
                GeometricTransformation[rig["graphics primitives"], rotfn],
                cs], renderPos + (epsilon + h) e3]]}},
            ImageSize → Large, Axes → True,
            AxesLabel → axisLabelStyle /@ {"X", "Y", "Z"},
            PlotRange → {{-vu, vu}, {-vu, vu}, {-vu, vu}}],
          ParametricPlot3D[RotationMatrix[-ha, e1].
            RotationMatrix[δx /. t → u, e2].cb, {ha, 0, 2 π}]}]],
       {u, 0, tL}, AnimationRate → .5]],
     {{δ0, 10. °}, 0, 90. °}, {{η0, 5. °}, 0, 90. °}, {{tL, 25}, 0, 50}]]]]]
```

Out[332]=



t = 14.0647, E = 11.5491

$\{\delta_0,\eta_0\}$ = {   10.0000,   5.0000}

$\{\delta,\eta\}$ = { -60.9135,   242.6983}

c_s = {   -1.0487,   -0.5184,   -0.2676}

Discrete Lagrangian

Instead of integrating those equations, let's discretize the Lagrangian, itself, via Equation 7 of Reference 10. First, a reminder

In[333]:=

```
?? Lnum
```

Out[333]=

```
Symbol

Global`Lnum

Definitions
Lnum[{δ_, η_}, {Dδ_, Dη_}] := Tnum[{δ, η}, {Dδ, Dη}] - Vnum[{δ, η}]

Full Name   Global`Lnum
```

Whereas `LNum` takes a coordinate tuple q and a velocity tuple $\dot{q}$, the poorly named ***discrete Lagrangian*** L_d takes a pair of coordinate tuples: one, q_k, at time t_k and another, q_{k+1}, at time t_{k+1}. L_d is poorly named because it's actually an increment of ***action*** (units of energy × time), being the product of a value of `LNum` (units of energy) and the finite (constant) time increment `dt` (h in the paper). Despite the notation in the paper, L_d depends on dt. We capture that dependence in our rendition of L_d. Despite the fact that dt is a constant, for now, we might want to vary it later. We assume midpoint quadrature, halfway between q_k and q_{k+1}.

In[334]:=

```
ClearAll[Ld, pk, pkp1, δkm1, ηkm1, δk, ηk, δkp1, ηkp1, dt];
Ld[qk : {δk_, ηk_}, qkp1 : {δkp1_, ηkp1_}, dt_] :=
  dt Lnum[1/2 (qk + qkp1), 1/dt (qkp1 - qk)];
```

Discrete Euler-Lagrange Equations

The ***discrete Euler-Lagrange equations*** (***DELE***) are

$$D_1 L_d(q_k, q_{k+1}) + D_2 L_d(q_{k-1}, q_k) = 0 \tag{1}$$

As Stern and Desbrun point out, DELE imply a recurrence, automatically suitable for an integrator. I go a tiny step further and note that it's almost a foldable as it stands. We'll use that fact to advantage shortly. A wrinkle is that DELE require two initial positions (δ-η tuples), whereas we have initial positions and velocities. There is enough information to get two initial positions, however, via a numerical root in the position-momentum form shown in Section 5.3 of Reference 10.

First, we need the derivatives. $D_1 L_d(\{\delta_k, \eta_k\}, \{\delta_{k+1}, \eta_{k+1}\}) = \{\partial_{\delta_k} L_d(\,\ldots),\ \partial_{\eta_k} L_d(\,\ldots)\}$ and $D_2 L_d(\{\delta_k, \eta_k\}, \{\delta_{k+1}, \eta_{k+1}\}) = \{\partial_{\delta_{k+1}} L_d(\,\ldots),\ \partial_{\eta_{k+1}} L_d(\,\ldots)\}$. *Mathematica*'s *Evaluate in Place* is handy for this, because we only need the symbolic derivatives once. Here are closed forms for the conjugate momenta p_k and p_{k+1} via Equation 8 of Reference 10.

In[336]:=

```
pk[{δk_, ηk_}, {δkp1_, ηkp1_}, dt_] := (* -D1Ld(qk,qk+1) *)
  (* -{D[Ld[{δk,ηk},{δkp1,ηkp1},dt],δk]//FullSimplify,
    D[Ld[{δk,ηk},{δkp1,ηkp1},dt],ηk]//FullSimplify}, Evaluate in Place *)
  -{1/dt (-1.44 (-δk + δkp1) +
        5.886 dt^2 Cos[(ηk + ηkp1)/2] Sin[(δk + δkp1)/2] - 0.36 (ηk - ηkp1)^2 Sin[δk + δkp1]),
    1/dt (0.72 ηk - 0.72 ηkp1 + (0.72 ηk - 0.72 ηkp1) Cos[δk + δkp1] +
        5.886 dt^2 Cos[(δk + δkp1)/2] Sin[(ηk + ηkp1)/2])};

pkp1[{δk_, ηk_}, {δkp1_, ηkp1_}, dt_] := (* +D2Ld(qk,qk+1) *)
  (* +{D[Ld[{δk,ηk},{δkp1,ηkp1},dt],δkp1]//FullSimplify,
    D[Ld[{δk,ηk},{δkp1,ηkp1},dt],ηkp1]//FullSimplify},
  Evaluate in Place *)
  {1/dt (1.44 (-δk + δkp1) +
        5.886 dt^2 Cos[(ηk + ηkp1)/2] Sin[(δk + δkp1)/2] - 0.36 (ηk - ηkp1)^2 Sin[δk + δkp1]),
   1/dt (-0.72 ηk + 0.72 ηkp1 + (-0.72 ηk + 0.72 ηkp1) Cos[δk + δkp1] +
        5.886 dt^2 Cos[(δk + δkp1)/2] Sin[(ηk + ηkp1)/2])};
```

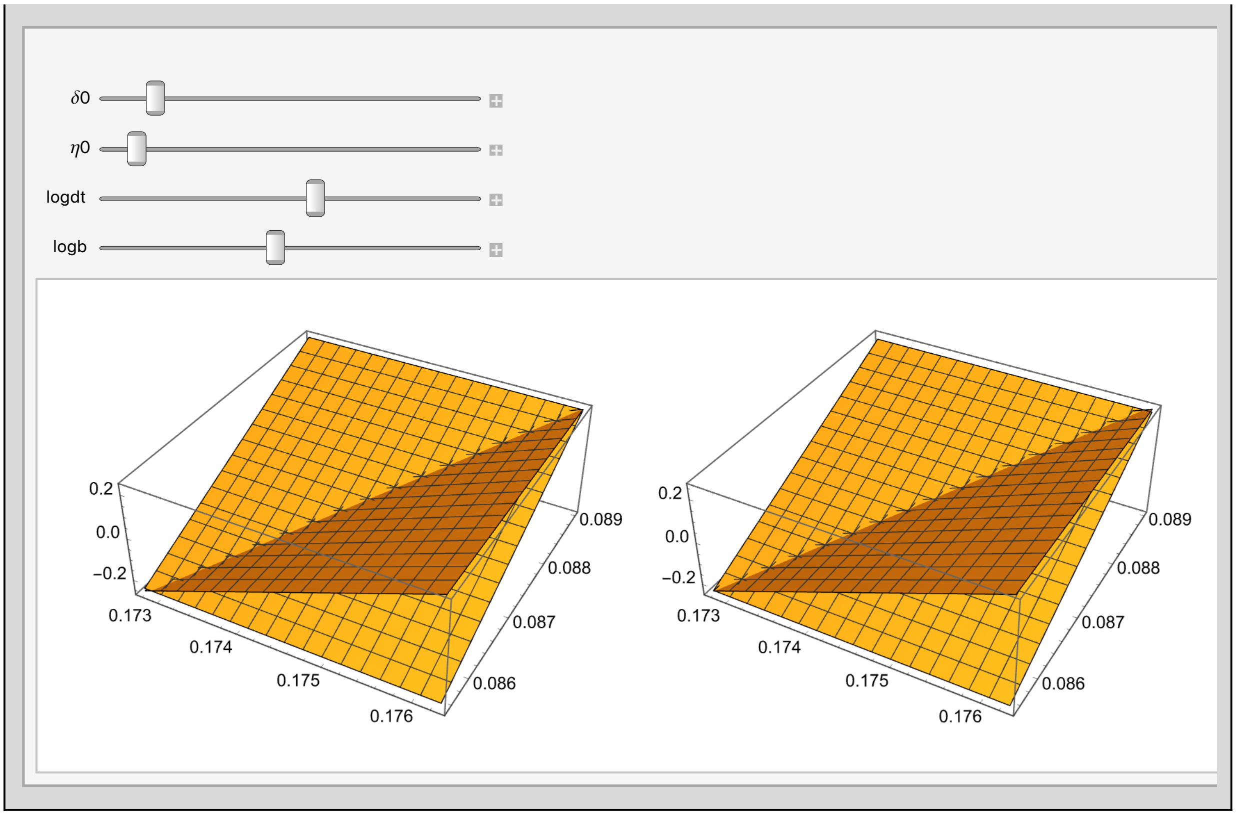
Visual Sanity Check

Plot momentum surfaces for a manipulable interval ±b around the manipulable initial conditions $\delta_0 = 10\,°$ and $\eta_o = 5\,°$ that we had in the ground truth default. We're looking to see that the zeros of these momenta are in range, because they imply the second position initials, δ_1 and η_1.

In[338]:=

```
Manipulate[
 With[{b = 10.^logb},
  GraphicsRow[
   {Plot3D[pk[{δ0, η0}, {δ1, η1}, 10.^logdt],
     {δ1, δ0 - b, δ0 + b}, {η1, η0 - b, η0 + b}],
    Plot3D[pkp1[{δ0, η0}, {δ1, η1}, 10.^logdt],
     {δ1, δ0 - b, δ0 + b}, {η1, η0 - b, η0 + b}]}]],
 {{δ0, 10. °}, 0, 90. °}, {{η0, 5. °}, 0, 90. °}, {{logdt, -2.}, -6, 1},
 {{logb, Log10[.1 °]}, Log10[0.0001 °], Log10[360 °]}]
```

Out[338]=



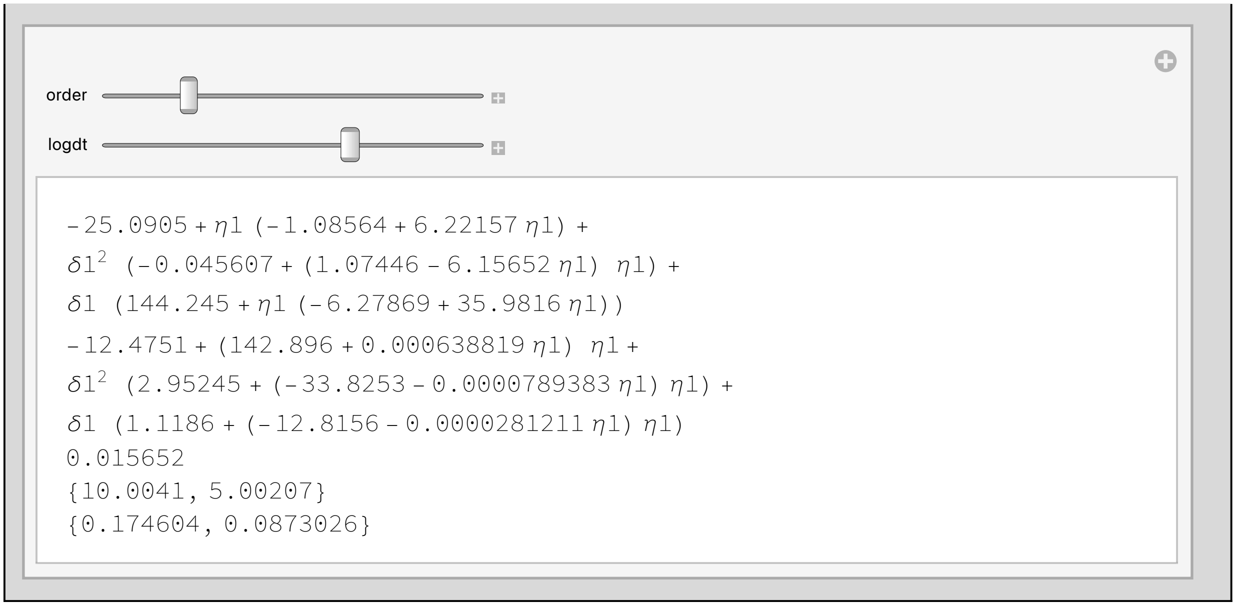
Numerical Solution for δ_1 and η_1

The equations are too hard to solve symbolically, so we suffice with a numerical solution of series expansions of the trigonometric functions. The following experiment shows that order-2 solutions, with a little safety margin, at dt = 0.01 are fast enough and good enough to get started.

In[339]:=

```
With[{b = 10. °, δ0 = 10. °, η0 = 5. °, pδ0 = 0.0, pη0 = 0.0},
 Manipulate[
  With[{dt0 = 10^logdt},
   With[{ps1 = Normal[
        Series[pk[{δ0, η0}, {δ1, η1}, dt0][[1]], {δ1, δ0, order}, {η1, η0, order}]],
      ps2 = Normal[Series[
         pk[{δ0, η0}, {δ1, η1}, dt0][[2]], {δ1, δ0, order}, {η1, η0, order}]]},
    With[{radians =
        AbsoluteTiming@NSolve[ps1 == pδ0 && ps2 == pη0 &&
           Abs[δ0 - δ1] < b && Abs[η0 - η1] < b,
          {δ1, η1}, Reals]},
      Column[{
        ps1 // FullSimplify,
        ps2 // FullSimplify,
        radians[[1]],
        {radians[[2, 1, 1, 2]], radians[[2, 1, 2, 2]]} / N[°],
        {radians[[2, 1, 1, 2]], radians[[2, 1, 2, 2]]}}]]]],
  {{order, 2}, 1, 6, 1}, {{logdt, -2}, -6, 0, 1}]]
```

Out[339]=



```
-25.0905 + η1 (-1.08564 + 6.22157 η1) +
δ1^2 (-0.045607 + (1.07446 - 6.15652 η1) η1) +
δ1 (144.245 + η1 (-6.27869 + 35.9816 η1))
-12.4751 + (142.896 + 0.000638819 η1) η1 +
δ1^2 (2.95245 + (-33.8253 - 0.0000789383 η1) η1) +
δ1 (1.1186 + (-12.8156 - 0.0000281211 η1) η1)
0.015652
{10.0041, 5.00207}
{0.174604, 0.0873026}
```

bootIntegrator

Package the solution in a function so we can call it to bootstrap the integrator.

In[340]:=

```
ClearAll[bootIntegrator];
bootIntegrator[b_, δ0_, η0_, pδ0_, pη0_, logdt_ : -2, order_ : 2] :=
  With[{dt0 = 10^logdt},
   With[{ps1 = Normal[
        Series[pk[{δ0, η0}, {δ1, η1}, dt0]⟦1⟧, {δ1, δ0, order}, {η1, η0, order}]],
     ps2 = Normal[Series[
         pk[{δ0, η0}, {δ1, η1}, dt0]⟦2⟧, {δ1, δ0, order}, {η1, η0, order}]]},
    NSolve[ps1 == pδ0 && ps2 == pη0 &&
      Abs[δ0 - δ1] < b && Abs[η0 - η1] < b,
     {δ1, η1}, Reals]]];
bootIntegrator[10. °, 10. °, 5. °, 0.0, 0.0]
```

Out[342]=

```
{{δ1 → 0.174604, η1 → 0.0873026}}
```

Solve $D_1 L_d(q_{k+1}, q_{k+2}) + D_2 L_d(q_k, q_{k+1}) == 0$ for q_{k+2} when $k \geq 0$. Reuse functions `pk` and `pkp1` from above, by advancing `pk` one step. Sneak in non-conservative `forces` for the right-hand side of the Equation 7 of Reference 10, looking forward to the future. Sanity check: expect δ and η to increase a little as the baton falls.

In[343]:=

```
(* -pk[{δkp1_,ηkp1_},{δkp2_,ηkp2_},dt_]:= D_1L_d(q_{k+1},q_{k+2}) *)
(* pkp1[{δk_,ηk_},{δkp1_,ηkp1_},dt_]:= D_2L_d(q_k,q_{k+1}) *)
```

In[344]:=

```
With[{b = 10. °, δ0 = 10. °, η0 = 5. °,
  pδ0 = 0.0, pη0 = 0.0, dt0 = 0.01, order = 2, forces = 0},
 With[{q1 = bootIntegrator[b, δ0, η0, pδ0, pη0]}, (* numerical *)
  With[{δk = δ0, ηk = η0, δkp1 = q1⟦1, 1, 2⟧, ηkp1 = q1⟦1, 2, 2⟧}, (* bootstrap *)
   Module[{δkp2, ηkp2},
    With[{dele =
       pkp1[{δk, ηk}, {δkp1, ηkp1}, dt0] - pk[{δkp1, ηkp1}, {δkp2, ηkp2}, dt0]},
     With[{solns = NSolve[
           Normal[Series[dele, {δkp2, δkp1, order}, {ηkp2, ηkp1, order}]] == forces &&
            Abs[δkp2 - δkp1] < b &&
            Abs[ηkp1 - ηkp2] < b,
           {δkp2, ηkp2}, Reals]},
      Column[{
        {δkp1, ηkp1},
        {solns⟦1, 1, 2⟧, solns⟦1, 2, 2⟧}
       }]]]]]]]
```

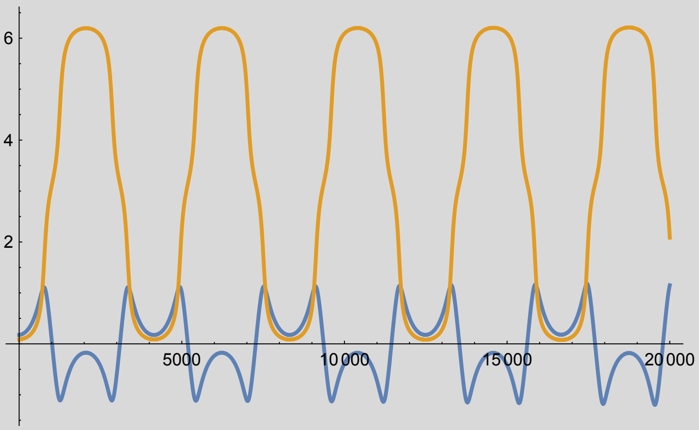
Out[344]=

```
{0.174604, 0.0873026}
{0.174816, 0.0874112}
```

Package the solver as a foldable and run it for a while. I had to reduce the time step to 1.25 msec to get plausible results. It takes too long, so I saved only a screen shot. It eventually goes wild, so we move on: too slow and not sufficiently conservative. The following cell is not evaluatable. Change it via *Mathematica*'s `Cell` menu if you want to devote some time to playing around with it.

```
ClearAll[foldableDele];
With[{b = 10. °, δ0 = 10. °, η0 = 5. °, pδ0 = 0.0, pη0 = 0.0},
 With[{q1 = bootIntegrator[b, δ0, η0, pδ0, pη0]}, (* numerical *)
  foldableDele[{{δk_, ηk_}, {δkp1_, ηkp1_}}, tkp1_] :=
   With[{order = 2, forces = 0},
    Module[{δkp2, ηkp2},
     With[{dele = pkp1[{δk, ηk}, {δkp1, ηkp1}, tkp1] -
         pk[{δkp1, ηkp1}, {δkp2, ηkp2}, tkp1]},
      With[{solns = NSolve[
           Normal[Series[dele, {δkp2, δkp1, order}, {ηkp2, ηkp1, order}]] == forces &
            Abs[δkp2 - δkp1] < b &&
            Abs[ηkp1 - ηkp2] < b,
           {δkp2, ηkp2}, Reals, WorkingPrecision → 24]},
       {{δkp1, ηkp1}, {solns⟦1, 1, 2⟧, solns⟦1, 2, 2⟧}}]]]];
  With[{tL = 20 000, dt0 = 0.00125},
   With[{qs =
      Quiet@FoldList[foldableDele,
        {{δ0, η0}, {q1⟦1, 1, 2⟧, q1⟦1, 2, 2⟧}}, ConstantArray[dt0, tL]]},
    qs;
    ListLinePlot[{#⟦2, 1⟧ & /@ qs, #⟦2, 2⟧ & /@ qs}]]]]]
```

Out[ ]=



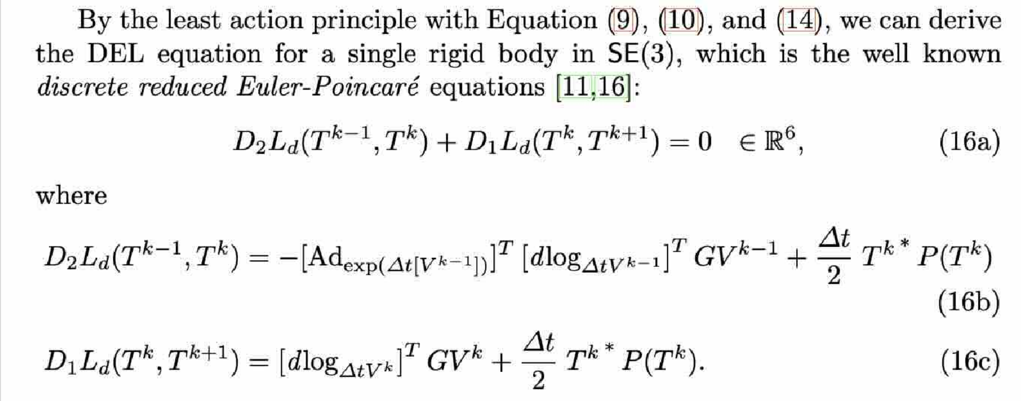
Conclusion: we leave further investigation of this solver to the future. In particular, we have not explored its energy conservation, purportedly automatic, but manifestly not.

DREPE, DRNEA, and CGDVIE3

Go back to quaternions, and from there to the group SE(3).[4] We'll integrate directly on the manifold of SE(3) via discrete methods. We'll bring up *Dzhanybekhov*, 4DISP in a later notebook.

There is a lot of machinery, here. It's mostly about implementing Equations 16b and 16c in Reference 2. Notice that Equation 16a is exactly the same as we have above in DEL, but in a 6-dimensional coordinate system on SE(3) instead of in our 2-dimensional Lagrangian coordinate system. These equations contain an undefined term, T^{k^*}. I have contacted the authors to find out what it might be. I suspect it to be an adjoint representation of a force in the co-tangent bundle, but getting this right will require more research.

Out[ ]=



By the least action principle with Equation (9), (10), and (14), we can derive the DEL equation for a single rigid body in SE(3), which is the well known *discrete reduced Euler-Poincaré* equations [11,16]:

$$D_2 L_d(T^{k-1}, T^k) + D_1 L_d(T^k, T^{k+1}) = 0 \quad \in \mathbb{R}^6, \tag{16a}$$

where

$$D_2 L_d(T^{k-1}, T^k) = -[\mathrm{Ad}_{\exp(\Delta t[V^{k-1}])}]^T \, [\mathrm{dlog}_{\Delta t V^{k-1}}]^T \, G V^{k-1} + \frac{\Delta t}{2} \, T^{k^*} \, P(T^k) \tag{16b}$$

$$D_1 L_d(T^k, T^{k+1}) = [\mathrm{dlog}_{\Delta t V^k}]^T \, G V^k + \frac{\Delta t}{2} \, T^{k^*} \, P(T^k). \tag{16c}$$

Utilities

Append Column

In[345]:=

```
<< Notation`
```

In[346]:=

```
ClearAll[appendColumn];
appendColumn[m_, c_] := ((mᵀ) ~Join~ (cᵀ))ᵀ
```

Append Matrix

In[348]:=

```
ClearAll[appendMatrix];
appendMatrix[m1_, m2_] := Fold[appendColumn, m1, Map[List, m2ᵀ, {2}]];
```

rotMatFromQ: Rotation Matrix From Quaternion

In[350]:=

```
ClearAll[rotMatFromQ];
rotMatFromQ[q_] :=
  appendColumn[
   {rv[q, e1]}ᵀ,
   appendColumn[
    {rv[q, e2]}ᵀ,
    {rv[q, e3]}ᵀ]];
```

3-2-1 Object Rotation From *ψ*, *θ*, *ϕ*

SE(3) includes SO(3) as a sub-manifold. We have two representations for SO(3): 2 quaternions for each rotation matrix. We need utilities to go back and forth amongst these representations, picking one of the two quaternions arbitrarily.

R: Frame Rotation

Given an object with components {*x*, *y*, *z*}, calculate its new components {*x*', *y*', *z*'} in the same, fixed frame after rotating the object.

In[352]:=

```
R[ψ_, θ_, ϕ_] :=
 RotationMatrix[ϕ, e1].
  RotationMatrix[θ, e2].
  RotationMatrix[ψ, e3]
```

Unit Tests

Expect *R*.*x* == *q v q*⋆, where x is a random vector and v is its quaternion representation.

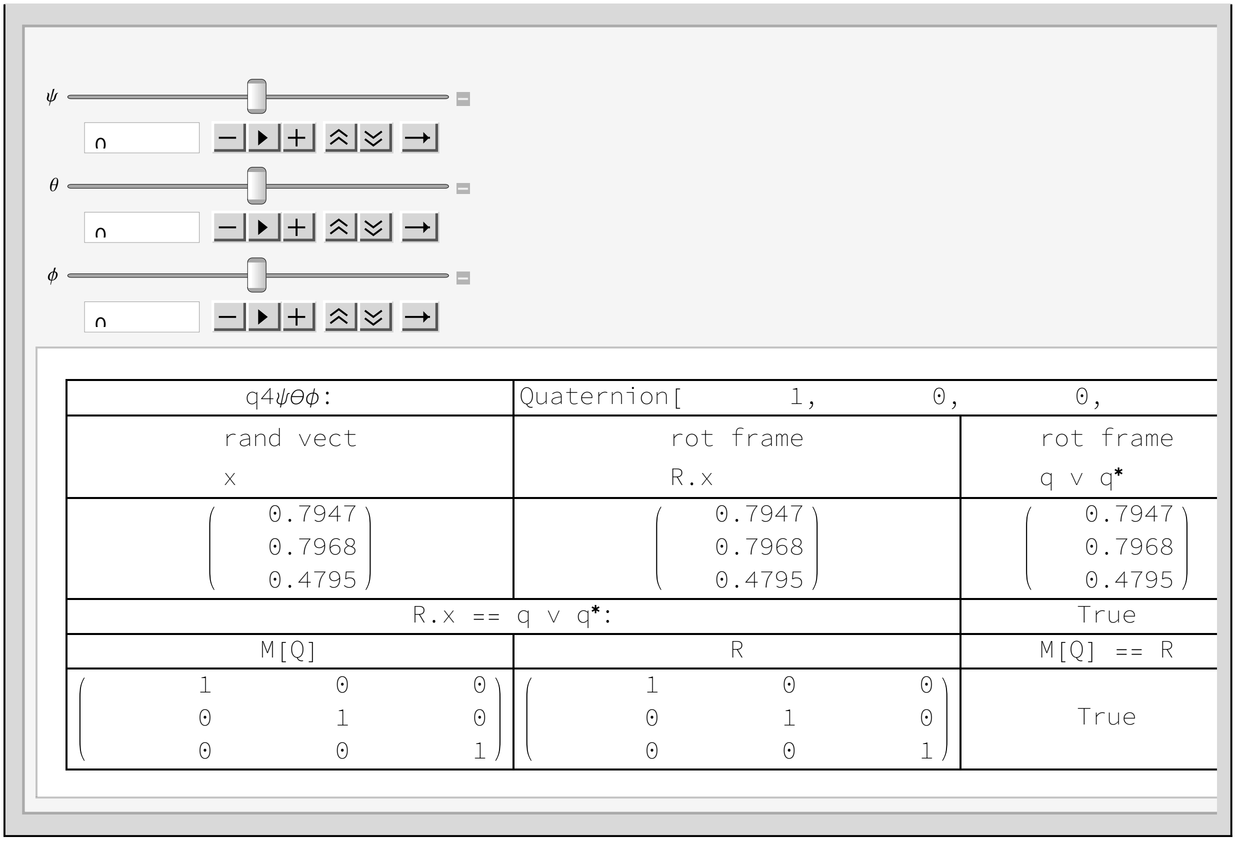
In[353]:=

```
ClearAll[pnf, pretty];
pnf = NumberForm[Chop[#], {7, 4},
    NumberSigns → {"-", " "}, NumberPadding → {" ", "0"}] &;
pretty[n_?NumberQ] := (pnf[n]);
pretty[m_] := MatrixForm[Map[pnf, m, {2}]];
pretty[q_Quaternion] := Map[pnf, q, {1}];
```

In[358]:=

```
With[{o33 = ConstantArray[0, {3, 3}]},
 Manipulate[
  Module[{r = R[ψ, θ, ϕ], q = Q[ψ, θ, ϕ], x, m, q2},
   x = RandomReal[{-1, 1}, {3, 1}];
   m = rotMatFromQ[q];
   Grid[
     {{"q4ψθϕ:", pretty@Chop@q, SpanFromLeft},
      {"rand vect\nx", "rot frame\nR.x", "rot frame\nq v q*"},
      {x // pretty,
       r.x // pretty, rv[q, x] // pretty},
      {"R.x == q v q*:", SpanFromLeft, Chop[r.x - rv[q, x]] == (List /@ o)},
      {"M[Q]", "R", "M[Q] == R"},
      {Chop@m // pretty, Chop@r // pretty, Chop[m - r] == o33}},
     Frame → All]],
  {{ψ, 0}, -π, π, Appearance → "Open"},
  {{θ, 0}, -π, π, Appearance → "Open"},
  {{ϕ, 0}, -π, π, Appearance → "Open"}]]
```

Out[358]=



q4ψθϕ:	Quaternion[1, 0, 0,	
rand vect x	rot frame R.x	rot frame q v q*
$\begin{pmatrix}0.7947\\0.7968\\0.4795\end{pmatrix}$	$\begin{pmatrix}0.7947\\0.7968\\0.4795\end{pmatrix}$	$\begin{pmatrix}0.7947\\0.7968\\0.4795\end{pmatrix}$
R.x == q v q*:		True
M[Q]	R	M[Q] == R
$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$	$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$	True

Automated tests; the next test should produce `{Null, {}}`.

In[359]:=

```
testConfluence[n_] :=
 With[{oo = List /@ o, o33 = ConstantArray[0, {3, 3}]},
  Reap[
   Do[
    Module[{r, q, ψ, θ, ϕ, m, x},
     ψ = RandomReal[{0, 2 π}];
     θ = RandomReal[{-π, π}];
     ϕ = RandomReal[{-π, π}];
     r = R[ψ, θ, ϕ];
     q = Q[ψ, θ, ϕ];
     m = rotMatFromQ[q];
     x = RandomReal[{-1, 1}, {3, 1}];
     If[Chop[r.x - rv[q, x]] ≠ oo,
      Sow[{r.x, rv[q, x], Chop[r.x - rv[q, x]]}]];
     If[Chop[m - r] ≠ o33,
      Sow[{m, r, Chop[m - r]}]]], {n}]]]
testConfluence[1000]
```

Out[360]=

```
{Null, {}}
```

yprFromRotMat: ψ, θ, ϕ From Rotation Matrix

For graphics, we'll need yaw ψ, pitch θ, and roll ϕ from- and to- the rotation-matrix representation of SO(3).

In[361]:=

```
R[ψ, θ, ϕ] /. shorteningRules // MatrixForm
```

Out[361]//MatrixForm=

$$\begin{pmatrix} C_\theta\, C_\psi & -C_\theta\, S_\psi & S_\theta \\ C_\psi\, S_\theta\, S_\phi + C_\phi\, S_\psi & C_\phi\, C_\psi - S_\theta\, S_\phi\, S_\psi & -C_\theta\, S_\phi \\ -C_\phi\, C_\psi\, S_\theta + S_\phi\, S_\psi & C_\psi\, S_\phi + C_\phi\, S_\theta\, S_\psi & C_\theta\, C_\phi \end{pmatrix}$$

The following orthogonality unit-test should produce the Identity matrix in three dimensions.

In[362]:=

```
(R[ψ, θ, ϕ].Transpose[R[ψ, θ, ϕ]] // FullSimplify) /. shorteningRules // MatrixForm
```

Out[362]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The following unit-test should produce 1.

In[363]:=

```
Norm[R[RandomReal[], RandomReal[], RandomReal[]]]
```

Out[363]=

```
1.
```

Definition

Now let ypr stand for yaw-pitch-roll. Sorry about the proliferation of symbols, they’re just easier to keyboard than are ψ, θ, and ϕ.

In[364]:=

```
ClearAll[yprFromRotMat];
               ⎛r11_ r12_ r13_⎞
yprFromRotMat[ ⎜  _  r22_ r23_⎟ ] :=
               ⎝  _    _  r33_⎠
  With[{θ = -ArcTan[Norm[{r23, r33}], (-r13)]},
   With[{ψ = -ArcTan[r11, r12]},
    With[{ϕ = -ArcTan[r33, r23]},
     {ψ, θ, ϕ}]]];
```

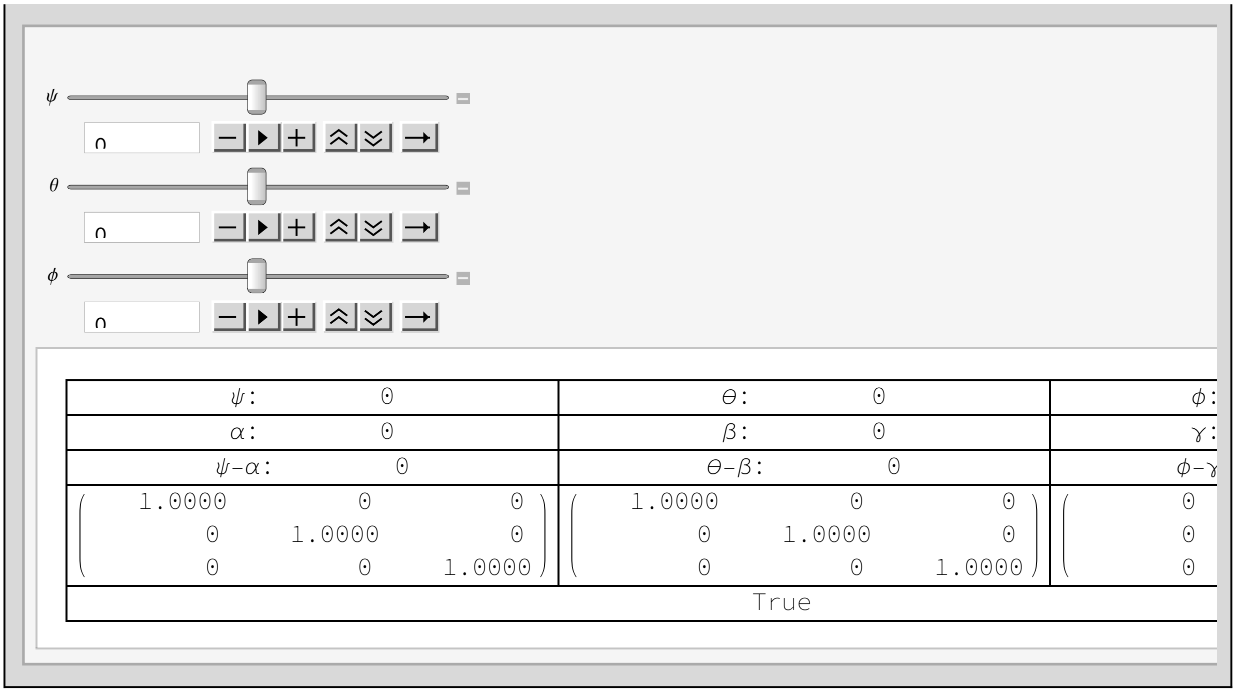
Unit Test

The mapping form yaw, pitch, and roll to a rotation matrix is not one-to-one, therefore its inverse does not exist. However, as with the log map from the Lie group SO(3) to the Lie algebra $\mathfrak{so}(3)$ below, we can write a pseudoinverse that picks values from a principal branch of yaw, pitch, and roll. We test by mapping back to a rotation matrix and checking for equality there. This testing strategy is called *round-tripping*.

In[366]:=

```
With[{o33 = ConstantArray[0, {3, 3}]},
 Manipulate[Module[{r = R[ψ, θ, ϕ], r2, x, q2, α, β, γ},
   x = RandomReal[{-1, 1}, {3, 1}];
   {α, β, γ} = yprFromRotMat[r];
   r2 = R[α, β, γ];
   Grid[{{"ψ: " <> ToString[pretty[ψ / °]],
      "θ: " <> ToString[pretty[θ / °]], "ϕ: " <> ToString[pretty[ϕ / °]]},
     {"α: " <> ToString[pretty[α / °]],
      "β: " <> ToString[pretty[β / °]], "γ: " <> ToString[pretty[γ / °]]},
     {"ψ-α: " <> ToString[pretty[(ψ - α) / °]], "θ-β: " <>
       ToString[pretty[(θ - β) / °]], "ϕ-γ: " <> ToString[pretty[(ϕ - γ) / °]]},
     {pretty[r], pretty[r2], pretty[r - r2]},
     {Round[r, 10^-6] === Round[r2, 10^-6], SpanFromLeft, SpanFromLeft}},
    Frame → All]],
  {{ψ, 0.0}, -1. π, 1. π, Appearance → "Open"},
  {{θ, 0.0}, -1. π, 1. π, Appearance → "Open"},
  {{ϕ, 0.0}, -1. π, 1. π, Appearance → "Open"}]]
```

Out[366]=



```
ψ:      0   |  θ:      0   |  ϕ:
α:      0   |  β:      0   |  γ:
ψ-α:     0  |  θ-β:     0  |  ϕ-γ
( 1.0000       0       0 )  ( 1.0000       0       0 )  (  0
(      0  1.0000       0 )  (      0  1.0000       0 )  (  0
(      0       0  1.0000 )  (      0       0  1.0000 )  (  0
                          True
```

qFromRotMat: Quaternion From Rotation Matrix

We pick one of the quaternions arbitrarily. This is quite a delicate operation. To save space and your time, we do not go over it in detail, but appeal rather to the successful round-tripping unit test.

Definition

In[367]:=

```
ClearAll[brack];
brack[u_, m_, i_, k_, s_] := u (m⟦k, i⟧ + s m⟦i, k⟧)

ClearAll[qFromRotMat];
qFromRotMat[r_] :=
 Module[{tr, t0, t1, t2, t3, u, qs, qw, qx, qy, qz},
  tr = Tr@r;
  If[Chop@tr > 0,
   u = 1 / (2 Sqrt[tr + 1]);
   {qw, qx, qy, qz} = qs =
      -1
     {───, brack[u, r, 3, 2, -1], brack[u, r, 1, 3, -1], brack[u, r, 2, 1, -1]},
      4 u
   If[(r⟦1, 1⟧ > r⟦2, 2⟧) && (r⟦1, 1⟧ > r⟦3, 3⟧),
    (* What guarantee that arg of sqrt ≥ 0? *)
    u = 1 / (2 Sqrt[1 + r⟦1, 1⟧ - r⟦2, 2⟧ - r⟦3, 3⟧]);
    {qw, qx, qy, qz} = qs =
                                  1
      {-brack[u, r, 3, 2, -1], ───, brack[u, r, 1, 2, 1], brack[u, r, 3, 1, 1]},
                                 4 u
    If[r⟦2, 2⟧ > r⟦3, 3⟧,
     u = 1 / (2 Sqrt[1 - r⟦1, 1⟧ + r⟦2, 2⟧ - r⟦3, 3⟧]);
     {qw, qx, qy, qz} = qs =
                                                        1
       {-brack[u, r, 1, 3, -1], brack[u, r, 1, 2, 1], ───, brack[u, r, 2, 3, 1]},
                                                       4 u
     u = 1 / (2 Sqrt[1 - r⟦1, 1⟧ - r⟦2, 2⟧ + r⟦3, 3⟧]);
     {qw, qx, qy, qz} = qs =
       {-brack[u, r, 2, 1, -1],
                                                     1
         brack[u, r, 3, 1, 1], brack[u, r, 2, 3, 1], ───}]]];
                                                    4 u
  Quaternion[qw, qx, qy, qz]]
```

Unit Test

This next should return `{Null, {}}`.

In[371]:=

```
testConfluence2[n_] :=
 With[{oo = List /@ o, o33 = ConstantArray[0, {3, 3}]},
  Reap[
   Do[
    Module[{r, q, q2, ψ, θ, ϕ, m, x},
     ψ = RandomReal[{-π, π}];
     θ = RandomReal[{-π, π}];
     ϕ = RandomReal[{-π, π}];
     q = Q[ψ, θ, ϕ];
     m = rotMatFromQ[q];
     q2 = qFromRotMat[m];
     x = RandomReal[{-1, 1}, {3, 1}];
     If[Chop[rv[q, x] - rv[q2, x]] ≠ oo,
      Sow[<|"r" → r, "x" → x, "q" → "r.x" → r.x,
        "rv[q.x]" → rv[q, x],
        "rv[q2.x]" → rv[q2, x],
        "rv[q,x]-rv[q2,x]" → Chop[rv[q, x] - rv[q2, x]]|>]]], {n}]]]
testConfluence2[1000]
```

Out[372]=

```
{Null, {}}
```

Proofs and Tests

Conceptual Checks

Test the effects of frame rotations of yaw, pitch, and roll on the axes of the BODY FRAME.

Consider e_1, pointing in the positive x direction of the body, that is, NORTH. After a small yaw, we expect this vector to have a small, positive y component, a zero z component, and an x component less than 1.

To turn e_1 into a column vector, all we have to do is map *List* over it.

In[373]:=

```
  N@Pi
R[────, 0, 0].(List /@ e1) // MatrixForm
   12
```

Out[373]//MatrixForm=

```
⎛0.965926⎞
⎜0.258819⎟
⎝   0.   ⎠
```

Looks good!

After a small pitch, we expect e_1 to have a small, negative z component (that is, up), a zero y component, and an x component less than 1.

```
In[374]:=
R[0, N@Pi/12, 0].(List /@ e1) // MatrixForm
Out[374]//MatrixForm=
( 0.965926 )
(    0.    )
(-0.258819 )
```

Good!

Roll should not affect e_1:

```
In[375]:=
R[0, 0, N@Pi/12].(List /@ e1) // MatrixForm
Out[375]//MatrixForm=
(1.)
(0.)
(0.)
```

Expect yaw to give e_2 a small, negative x component, a reduced y component, and a zero z component:

```
In[376]:=
R[N@Pi/12, 0, 0].(List /@ e2) // MatrixForm
Out[376]//MatrixForm=
(-0.258819 )
( 0.965926 )
(    0.    )
```

Expect pitch to leave e_2 unchanged:

```
In[377]:=
R[0, N@Pi/12, 0].(List /@ e2) // MatrixForm
Out[377]//MatrixForm=
(0.)
(1.)
(0.)
```

Expect roll to give e_2 a small, positive z component (that is, down), a reduced y component, and to leave the *x* component unchanged:

```
In[378]:=
R[0, 0, N@Pi/12].(List /@ e2) // MatrixForm
Out[378]//MatrixForm=
(    0.    )
( 0.965926 )
( 0.258819 )
```

Yaw should not affect e_3:

```
In[379]:=
R[N@Pi/12, 0, 0].(List /@ e3) // MatrixForm
Out[379]//MatrixForm=
( 0. )
( 0. )
( 1. )
```

Pitching e_3 up should yield a small, positive *x* component, a reduced *z* component, and should leave *y* unchanged:

```
In[380]:=
R[0, N@Pi/12, 0].(List /@ e3) // MatrixForm
Out[380]//MatrixForm=
( 0.258819 )
(    0.    )
( 0.965926 )
```

Rolling e_3 should yield a small, negative *y* component, a reduced *z* component, and an unchanged *x* component:

```
In[381]:=
R[0, 0, N@Pi/12].(List /@ e3) // MatrixForm
Out[381]//MatrixForm=
(     0.    )
( -0.258819 )
(  0.965926 )
```

Investigate Formulae

In[382]:=

```
<< Notation`
Quiet[Symbolize[ c_ψ ];
 Symbolize[ s_ψ ];
 Symbolize[ c_θ ];
 Symbolize[ s_θ ];
 Symbolize[ c_ϕ ];
 Symbolize[ s_ϕ ];]
```

In[384]:=

```
shorteningRules = {Cos[x_] :→ c_x, Sin[y_] :→ s_y};
```

In[385]:=

```
lengtheningRules = {c_x_ :→ Cos[x], s_y_ :→ Sin[y]};
```

In[386]:=

```
R[ψ, θ, ϕ] /. shorteningRules // MatrixForm
```

Out[386]//MatrixForm=

$$\begin{pmatrix} C_\theta\, C_\psi & -C_\theta\, S_\psi & S_\theta \\ C_\psi\, S_\theta\, S_\phi + C_\phi\, S_\psi & C_\phi\, C_\psi - S_\theta\, S_\phi\, S_\psi & -C_\theta\, S_\phi \\ -C_\phi\, C_\psi\, S_\theta + S_\phi\, S_\psi & C_\psi\, S_\phi + C_\phi\, S_\theta\, S_\psi & C_\theta\, C_\phi \end{pmatrix}$$

For verifying frame rotation, it’s best to reverse the signs of all angles. This next one matches Guangyu Liu’s dissertation on the inverted, spherical pendulum, his Equation 2.8:

In[387]:=

```
R[-ψ, -θ, -ϕ] /. shorteningRules // MatrixForm
```

Out[387]//MatrixForm=

$$\begin{pmatrix} C_\theta\, C_\psi & C_\theta\, S_\psi & -S_\theta \\ C_\psi\, S_\theta\, S_\phi - C_\phi\, S_\psi & C_\phi\, C_\psi + S_\theta\, S_\phi\, S_\psi & C_\theta\, S_\phi \\ C_\phi\, C_\psi\, S_\theta + S_\phi\, S_\psi & -C_\psi\, S_\phi + C_\phi\, S_\theta\, S_\psi & C_\theta\, C_\phi \end{pmatrix}$$

This is not the same as the transpose of $R[\psi, \theta, \phi]$:

In[388]:=

```
R[ψ, θ, ϕ]ᵀ /. shorteningRules // MatrixForm
```

Out[388]//MatrixForm=

$$\begin{pmatrix} C_\theta\, C_\psi & C_\psi\, S_\theta\, S_\phi + C_\phi\, S_\psi & -C_\phi\, C_\psi\, S_\theta + S_\phi\, S_\psi \\ -C_\theta\, S_\psi & C_\phi\, C_\psi - S_\theta\, S_\phi\, S_\psi & C_\psi\, S_\phi + C_\phi\, S_\theta\, S_\psi \\ S_\theta & -C_\theta\, S_\phi & C_\theta\, C_\phi \end{pmatrix}$$

but that, of course, is the same as the product of the transposes of the composed matrices, in opposite order:

In[389]:=

```
RotationMatrix[ψ, e3]ᵀ.
   RotationMatrix[θ, e2]ᵀ.
   RotationMatrix[ϕ, e1]ᵀ /. shorteningRules // MatrixForm
```

Out[389]//MatrixForm=

$$\begin{pmatrix} C_\theta\ C_\psi & C_\psi\ S_\theta\ S_\phi + C_\phi\ S_\psi & -C_\phi\ C_\psi\ S_\theta + S_\phi\ S_\psi \\ -C_\theta\ S_\psi & C_\phi\ C_\psi - S_\theta\ S_\phi\ S_\psi & C_\psi\ S_\phi + C_\phi\ S_\theta\ S_\psi \\ S_\theta & -C_\theta\ S_\phi & C_\theta\ C_\phi \end{pmatrix}$$

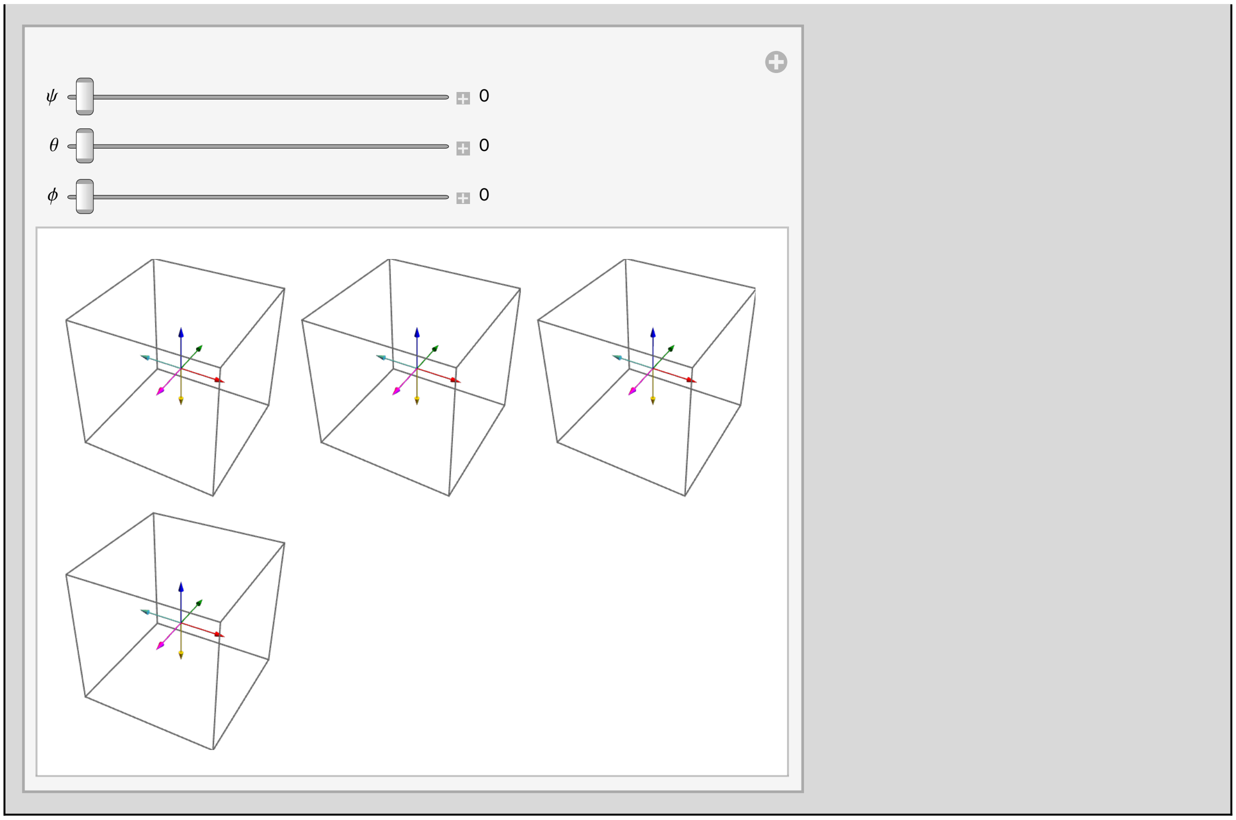
What is Liu up to?

In the following demonstration, the first and last images should be always the same. The second and third images are for inspection.

In[390]:=

```
With[{rg = {-1.5, 1.5}},
 Manipulate[
  GraphicsGrid[{{
     Graphics3D[GeometricTransformation[
       jack[rq0, 1.0], R[ψ, θ, ϕ]],
      PlotRange → {rg, rg, rg}, ImageSize → Medium],
     Graphics3D[GeometricTransformation[
       jack[rq0, 1.0], R[ψ, θ, ϕ]ᵀ],
      PlotRange → {rg, rg, rg}, ImageSize → Medium],
     Graphics3D[GeometricTransformation[
       jack[rq0, 1.0], R[-ψ, -θ, -ϕ]],
      PlotRange → {rg, rg, rg}, ImageSize → Medium]}, {
     Graphics3D[GeometricTransformation[
       jack[rq0, 1.0], RollPitchYawMatrix[{ψ, θ, ϕ}]],
      PlotRange → {rg, rg, rg}, ImageSize → Medium]
    }}],
  {{ψ, 0}, 0, 2 Pi, Appearance → "Labeled"},
  {{θ, 0}, 0, 2 Pi, Appearance → "Labeled"},
  {{ϕ, 0}, 0, 2 Pi, Appearance → "Labeled"}]]
```

Out[390]=



Our *R* matrix is algebraically identical to *Mathematica*'s built-in:

In[391]:=

```
R[ψ, θ, ϕ] === RollPitchYawMatrix[{ψ, θ, ϕ}]
```

Out[391]=

```
True
```

Another proof:

These two forms are algebraically equal, at least according to *Mathematica*, which cancels divisors without stipulation that they be non-zero. Because of that feature, we also need the numerical work above.

In[392]:=

$$\left(\texttt{R}[\psi,\theta,\phi].\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} == \texttt{rv}\left[\texttt{Q}[\psi,\theta,\phi],\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}\right]\right)\ \texttt{// FullSimplify}$$

Out[392]=

```
True
```

DREPE, DRNEA, and CGDVIE3 Theory

We need this theory to decode the abstract notation in the paper into practical matrix operations.

Manifolds Cheat Sheet

Via References [14] and [26].

Let M be a manifold of abstract points (later, we’ll put them in a Lie Group). A *coordinate chart* maps an open subset of abstract points to real n-vectors, i.e., Euclidean geometries. The manifold is *locally flat*.

Where charts overlap, the transformation functions are infinitely differentiable. That fact lets us define differentiability on the manifold itself.

Compatible charts ϕ and ϕ' have smooth (C^∞) coordinate-transformation functions where they overlap. The requirement of *openness* means there can be a topology.

We pun the manifold itself, M, to mean a union of compatible charts. Under physical circumstances, there will be *tangent spaces* almost everywhere. Tangent spaces contain *tangent vectors*, which are equivalence classes of curves passing through a point. It’s possibly unfamiliar to think of equivalence classes of curves as vectors, but they satisfy all 8 axioms of a vector space.

NOTION	NOTATION	DEFINITION	REMARKS
Coordinate Chart	(U, φ)	$U \subset M$ $\varphi : U \leftrightarrow \varphi(U) \subset \mathbb{R}^n$	$\varphi(m) =$ $\{x^1, x^2, \ldots, x^n\}$

Compatible Charts	$(U, \varphi), (U', \varphi')$	$\varphi' \circ \varphi^{-1} \mid \varphi(U \cap U')$ is C^∞; $\varphi(U \cap U')$ and $\varphi'(U \cap U')$ are open	
Differentiable Manifold	Every $m \in M$ is in at least one chart	M is a union of compatible charts	
Differentiable Structure	Maximal atlas		
Neighborhood	φ^{-1} applied to neighborhood in $\mathbb{R}^n$	Hausdorff topology : $m \neq m' \Rightarrow$ $\exists$ non − intersecting neighborhoods in M	
Equivalent curves	$c_1(0) = c_2(0)$ $(\varphi \circ c_1)'(0) = (\varphi \circ c_2)'(0)$	$c_1 : \mathbb{R} \to M$ $c_2 : \mathbb{R} \to M$	for some chart φ
Tangent Vector	$v(m)$	Equivalence class of curves	
Equivalence Class	[c (t)]	All curves with same value and derivatives through some chart φ at $t = 0$	
Tangent elsewhere on a curve	$\forall\ c(t)$, def $c'(s)$ at $c(s)$ $\left.\frac{dc}{dt}\right\|_{t=0}$	$c'(s) \in$ eqv. class $[(t \longmapsto c(s+t)) \mid_{t=0}]$	finite distance s down curve $c(t)$
Tangent Space	$T_m M$	Space of all tangent vectors at $m \in M$	THEOREM : $T_m M$ is a vector space
Components of a Vector	$v^i = \frac{d}{dt}(\varphi \circ c)^i \mid_{t=0}$	Superscript is contravariant index	
Tangent Bundle	$TM = \bigsqcup_{m \in M} T_m M$	Includes local coordinates and components of vectors	dimension is 2 n; ⊔ is disjoint union
Natural Projection	$\tau_M : TM \to M$ $\tau_M^{-1}(m) = T_m M$	Returns attachment point of a vector	$\tau_M^{-1}(m)$ is the ***fiber*** of TM at m
Differentiable, Derivative	With $f : M \to N$ $T_m f : T_m M \to T_{f(m)} N$	$T_m f$ is a linear map i.e., a matrix, s.t. $T_m f \cdot v =$ $T_m f \cdot \left.\frac{dc}{dt}\right\|_{t=0} =$ $\left.\frac{d}{dt} f(c(t))\right\|_{t=0}$	

Lie Group Cheat Sheet

Via Reference 24.

Lie Groups are both manifolds *and* groups (https://en.wikipedia.org/wiki/Group_theory). Being a group means that moving around on the manifold is easy: it is group multiplication (you can never leave the manifold, you can always go back where you came from, you’re allowed to go nowhere, it doesn’t matter whether you go to A first, stop and have a coffee, then rush through B to C; or whether you rush through A, have a coffee at B, then go to C). That means it’s easier than average to set up charts and thereby to do calculations. The notation is crufty but the ideas are not.

Adjoints Cheat Sheet (UNFINISHED)

This looks forward to a complete understanding of Equation 16b of Reference 2, and a force model that lets us do 4DISP. We leave it to the future.

Via private communication from Professor Frank C. Park, an author of Reference [2].

- Let $\boldsymbol{V} \in \mathfrak{se}(3) = (\vec{\omega}, \vec{v})$ be a 2-tuple of ordinary 3-vectors ≡ Euclidean 6-dimensional column vector in the $\mathbb{R}^6$ representation of $\mathfrak{se}(3)$, representing an angular-velocity 3-vector $\vec{\omega}$ and a velocity 3-vector $\vec{v}$.
- Likewise, let $\boldsymbol{F} \in \mathfrak{se}^\star(3) = \left(\underset{\rightharpoonup}{\mu}, \underset{\rightharpoonup}{f}\right)$ be a 2-tuple of 1-forms, i.e., 3-dimensional covectors ≡ Euclidean 6-dimensional row vector in the $\mathbb{R}^6$ representation of $\mathfrak{se}(3)$, representing a moment 3-covector $\underset{\rightharpoonup}{\mu}$ and a force 3-covector $\underset{\rightharpoonup}{f}$.
- Let $\boldsymbol{T} = (R, \vec{p}) \in \mathrm{SE}(3)$, where R is a 3×3 rotation matrix in SO(3), $\vec{p}$ is a 3-dimensional column vector representing displacement, and the final row of $\boldsymbol{T}$ is suppressed.
- Let × denote the usual three-dimensional vector cross-product, extended to work on covectors

1. $\mathrm{ad}_{\boldsymbol{V}} : \mathfrak{se}(3) \to \mathfrak{se}(3)$

1.1. $\mathrm{ad}_{\boldsymbol{V}_1}(\boldsymbol{V}_2) \overset{\Delta}{=} (\vec{\omega}_1 \times \vec{\omega}_2, \vec{v}_1 \times \vec{\omega}_2 - \vec{\omega}_1 \times \vec{v}_2)$
With $\boldsymbol{V}_1 = (\vec{\omega}_1, \vec{v}_1)$ and $\boldsymbol{V}_2 = (\vec{\omega}_2, \vec{v}_2)$,

2. $\mathrm{ad}_{\boldsymbol{V}}{}^\star : \mathfrak{se}^\star(3) \to \mathfrak{se}^\star(3)$

2.1. $\mathrm{ad}_{\boldsymbol{V}}{}^\star(\boldsymbol{F}) \overset{\Delta}{=} \left(\underset{\rightharpoonup}{\mu} \times \vec{\omega} + \underset{\rightharpoonup}{f} \times \vec{v}, \underset{\rightharpoonup}{f} \times \vec{\omega}\right)$

3. $\mathrm{Ad}_{\boldsymbol{T}} : \mathfrak{se}(3) \to \mathfrak{se}(3)$

3.1. $\mathrm{Ad}_{\boldsymbol{T}}(\boldsymbol{V}) \overset{\Delta}{=} (R.\vec{\omega}, R.\vec{v} + \vec{p} \times R.\vec{\omega})$

4. $\mathrm{Ad}_{\boldsymbol{T}}{}^\star : \mathfrak{se}^\star(3) \to \mathfrak{se}^\star(3)$

4.1. $\mathrm{Ad}_{\boldsymbol{T}}{}^\star(\boldsymbol{F}) = \left(R^{\mathsf{T}}.\underset{\rightharpoonup}{\mu} - (R^{\mathsf{T}}.\vec{p}) \times \underset{\rightharpoonup}{f}, R^{\mathsf{T}}.\underset{\rightharpoonup}{f}\right)$

DRNEA (Reference 2)

SO(3), $\mathfrak{so}(3)$

Definitions

SO(3) is the set of 3×3 orthogonal matrices with unit determinant +1 (no reflections). Its Lie algebra is the set $\mathfrak{so}(3)$ of 3×3 skew-symmetric matrices $\omega_\times$ such that the matrix exponential $e^{\omega_\times}$ is in SO(3). The notation $\omega_\times$ is clarified below.

Quoting reference [6]:

- **Definition 3.18.** “Let G be a matrix Lie group. The Lie algebra of G, denoted $\mathfrak{g}$, is the set of all matrices X such that e^{tX} is in G for all real numbers t.”

The definition gives us a way to generate an element of G from any element X of $\mathfrak{g}$. Does it go the other way? Can we recover $g = e^{tX} \in G$ from an element X of $\mathfrak{g}$?

Assume we can. Write an element of $g \in G$ as e^{tX}. The formal derivative of e^{tX}, evaluated at $t=0$, namely $\left(\frac{d\,e^{tX}}{dt} = X\,e^{tX}\right)\Big|_{t=0}$, recovers X. Generalize: find a one-variable parameterization of the Lie group, differentiate $g(t)$ with respect to the one parameter t, and evaluate the derivative at $t=0$.

The recovered element $X \in \mathfrak{g}$ is unique, but, typically, many elements in the Lie algebra $\mathfrak{g}$ map to the same element g in the Lie group, as we show below.

Example

Let $G = \text{SO}(3)$, rotation matrices in Euclidean 3-space. Here is a yaw-only object-rotation matrix in SO(3):

In[393]:=

```
R[ψ, 0, 0] // MatrixForm
```

Out[393]//MatrixForm=

```
⎛ Cos[ψ]  -Sin[ψ]  0 ⎞
⎜ Sin[ψ]   Cos[ψ]  0 ⎟
⎝   0        0     1 ⎠
```

By hypothesis, it corresponds to some as-yet-unknown element X of the Lie algebra $\mathfrak{so}(3)$. Identify the parameter t of the ***curve*** e^{tX} with ψ and evaluate the derivative:

In[394]:=

```
D[R[ψ, 0, 0], ψ] // MatrixForm
```

Out[394]//MatrixForm=

```
⎛-Sin[ψ]  -Cos[ψ]  0⎞
⎜ Cos[ψ]  -Sin[ψ]  0⎟
⎝   0        0     0⎠
```

Evaluate at $t = \psi = 0$:

In[395]:=

```
Gso33 = D[R[ψ, 0, 0], ψ] /. {ψ → 0} // MatrixForm
```

Out[395]//MatrixForm=

```
⎛0  -1  0⎞
⎜1   0  0⎟
⎝0   0  0⎠
```

That’s one ***generator*** or *basis element* of $\mathfrak{so}(3)$. Here are the other two, by the same procedure on the other two angles, namely pitch θ and roll ϕ:

In[396]:=

```
Gso32 = D[R[0, θ, 0], θ] /. {θ → 0} // MatrixForm
```

Out[396]//MatrixForm=

```
⎛ 0  0  1⎞
⎜ 0  0  0⎟
⎝-1  0  0⎠
```

In[397]:=

```
Gso31 = D[R[0, 0, ϕ], ϕ] /. {ϕ → 0} // MatrixForm
```

Out[397]//MatrixForm=

```
⎛0  0   0⎞
⎜0  0  -1⎟
⎝0  1   0⎠
```

Given the basis matrices above, we can form *all* skew-symmetric matrices by linear combinations:

$$a_1\, G^{(1)}_{\mathfrak{so}(3)} + a_2\, G^{(2)}_{\mathfrak{so}(3)} + a_3\, G^{(3)}_{\mathfrak{so}(3)}\ ;\quad a_i \in \mathbb{R},\ \ i \in [1\,..\times 3]$$

It is convenient to say that the Lie algebra is this large set of all skew-symmetric matrices. However, some of those skew-symmetric matrices map to the same rotations, as demonstrated below. The mapping from this Lie algebra to its Lie group is many-to-one, and the inverse mapping either doesn’t exist (to a mathematician) or is multi-valued (to a physicist), requiring a particular choice to produce a single value.

Naming Convention (*Hungarian Types*)

We write reminders of the types of variables and of the output types of functions as suffixes like `So3` or `so3` for $\mathfrak{so}(3)$, `R3` for $\mathbb{R}^3$, `SO3` for SO(3), and so on. For example, `hatSo3` returns an element of $\mathfrak{so}(3)$, taking an element of $\mathbb{R}^3$ as input. `unHatR3` returns an element of $\mathbb{R}^3$, taking an element of SO(3) as input. We don't write reminders of input types; remember them or look them up. To ease the cognitive load, we provide convenience overloads based on *Mathematica* pattern-matching. We may write an overload that automatically senses the input types you want. For example, we have `TSE3[ψ,θ,ϕ,x,y,z]` and also `TSE3[Quat[w,ix,jy,kz],{x,y,z}]`. The former takes yaw, pitch, roll angles and three Cartesian displacements. The latter takes a rotation quaternion and a list of Cartesian displacement. They both return elements of SE(3), that is, properly formatted 4×4 matrices.

The type of 6×6 matrices is `6R6`. The type of 6×1 column vectors is `6R1`. The type of 1×6 row vectors is `1R6`. The type of a 2×3 matrix is `2R3`, and so on. The type of a flat list or vector with 6 elements is `R6`.

Ideally, types would contain units-of-measure, but such would lead to excessively long reminders, and not work at all for matrices that contain elements with different units-of-measure. For now, we live with shape types (e.g., 6R6) and domain types (e.g., SO(3)). See https://math.stackexchange.com/questions/1275724/when-should-matrices-have-units-of-measurement and *Multidimensional Analysis* (https://a.co/d/8c5RqOX).

We tend to camel-case, where the first character in a name is lower-case and intermediate sub-names are capitalized. However, due to the many capital letters in the mathematical references, due to ambiguity about whether or how to capitalize after a fully capitalized acronym, and due to ambiguity about whether to capitalize after numerals, we admit to inconsistency in this convention.

Names that begin with dollar signs are *Mathematica* built-ins, like `$MachineEpsilon = 2.2204*^-16`. Names that end with dollar signs signify *ad-hoc* variables, that is, mutable variables. Names *without* dollar signs at the end may be trusted to have the same meanings everywhere in the document, but names with dollar signs at the ends, such as `T0$`, `T1$`, and `T2$`, may not be so trusted. We feel free to change their meanings at will to serve some purpose near-at-hand.

Vectors as Lists, Column Vectors, Row Vectors

Mathematica and Python share the feature that vectors are sometimes just flat lists and sometimes are lists of lists, that is, $n\times1$ matrices — column vectors, or $1\times n$ matrices — row vectors. Many operations, like dot products, work whether you give them a list or a list of lists. But some operations, like `Transpose`, are only meaningful on lists of lists. This intentional ambiguity is intentionally risky, but we live with it for now.

hatSo3: $\mathbb{R}^3 \to \mathfrak{so}(3)$, unHatR3: $\mathfrak{so}(3) \to \mathbb{R}^3$

An element of $\mathfrak{so}(3)$ is any skew-symmetric matrix $\omega_\times$. The subscript × means "skew symmetric" and should evoke 3D cross product in your mind. Such a matrix has only three independent elements, so it corresponds bijectively to an element of $\mathbb{R}^3$. The *hat* map is invertible; it just rearranges components from a vector ω in $\mathbb{R}^3$ into a matrix $\omega_\times$. The inverse map *unHat* moves the elements of back into a vector ω. Both directions are total, meaning that every $\omega\in\mathbb{R}^3$ corresponds to an element of $\mathfrak{so}(3)$, and vice versa.

In[398]:=

```
ClearAll[hatSo3, unHatR3];
                         ⎛ 0   -ω3  ω2 ⎞
hatSo3[ω1_, ω2_, ω3_] := ⎜ ω3   0  -ω1 ⎟;
                         ⎝-ω2  ω1   0  ⎠
(* Convenience overload, taking a vector-as-list;
the other convenience overloads,
taking a vector-as-column-vector will be introduced if needed (TODO). *)

hatSo3[{ω1_, ω2_, ω3_}] := hatSo3[ω1, ω2, ω3];

unHatR3[ωHat_] := {ωHat⟦3, 2⟧, -ωHat⟦3, 1⟧, ωHat⟦2, 1⟧};
```

The square of *hat* appears below in Rodrigues’ formula.

In[402]:=

```
hatSo3[ω1, ω2, ω3].hatSo3[ω1, ω2, ω3] // MatrixForm
```

Out[402]//MatrixForm=

$$\begin{pmatrix} -\omega2^2-\omega3^2 & \omega1\,\omega2 & \omega1\,\omega3 \\ \omega1\,\omega2 & -\omega1^2-\omega3^2 & \omega2\,\omega3 \\ \omega1\,\omega3 & \omega2\,\omega3 & -\omega1^2-\omega2^2 \end{pmatrix}$$

- **Unit Test**

In[403]:=

```
With[{ωx = hatSo3[ω1, ω2, ω3]},
 unHatR3[ωx]]
```

Out[403]=

```
{ω1, ω2, ω3}
```

expSO3: 𝔰𝔬(3) → SO(3), AKA Rodrigues’ Formula

- *“Rodrigues’” is spelled with no “s” after the possessive apostrophe at the end because the last “s” in “Rodrigues” is voiced. There is no consensus for such punctuation, but this is the way AMS [American Mathematical Society] does it. APA [American Psychological Association] would have us write “Rodrigues’s.”*

Rodrigues’ formula (https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula) remarkably presages Lie theory by some decades.

The following version of the *exp* map is not suitable for symbolic computations because it has a numerical test for small angles. We factor out the small-angle computations because we need them later for SE(3) and 𝔰𝔢(3). We make a symbolic version later below.

■ θThresh: Global Constant

When angles are smaller than `θThresh`, functions use an explicit Taylor series rather than built-ins to prevent division by small floating-point numbers.

Truncate all series in the angular variable θ at the eighth degree because 0.01 to the eighth power is 10^{-16}, at or near the precision of double-precision floating-point numbers, and we use 0.01 frequently as a discrete increment. In this sense, all our series “know” that the threshold for transitioning from built-in functions to explicit Taylor series is 0.01. We do not expect this “knowledge leak” to be trouble, but we may need to keep any eye out for it.

In[404]:=

```
ClearAll[θThresh];
θThresh = 0.01;(* θ^8 ≈ $MachineEpsilon *)
```

■ sθOverθ: Sin[θ]/θ

In[406]:=

```
ClearAll[sθOverθ];
sθOverθ[θ_] := If[θ < θThresh, (* θ^8 ≈ $MachineEpsilon *)
   1 - θ^2/3 ! + θ^4/5 ! - θ^6/7 ! + θ^8/9 !, (* Series[Sin[θ]/θ,{θ,0,9}] *)
   Sin[θ]/θ];
```

■ oneMcθOverθ2: (1 – Cos[θ]) / θ²

In[408]:=

```
ClearAll[oneMcθOverθ2];
oneMcθOverθ2[θ_] := If[θ < θThresh,
   1/2 ! - θ^2/4 ! + θ^4/6 ! - θ^6/8 ! + θ^8/10 !,
   (1 - Cos[θ])/θ^2];(* Series[(1-Cos[θ])/θ^2,{θ,0,9}] *)
```

■ expSO3: $e^{\omega_\times}$

In[410]:=

```
ClearAll[expSO3];
expSO3[ωHat_] :=
  With[{ω = unHatR3[ωHat]},
   With[{θ = Sqrt[ω.ω]},
    With[{c1 = sθOverθ[θ], c2 = oneMcθOverθ2[θ]},
     IdentityMatrix[3] + c1 ωHat + c2 ωHat.ωHat]]];
```

The expression

$$\exp[\omega_\times] = I + \frac{\mathrm{Sin}[\theta]}{\theta}\,\omega_\times + \frac{1-\mathrm{Cos}[\theta]}{\theta^2}\,\omega_\times^2, \text{ with } \theta = \sqrt{\omega_\times\cdot\omega_\times} \tag{2}$$

is Rodrigues’ rotation formula when the norm (Euclidean length) of $\omega_\times$ is equal to the angle of twist. Many other derivations of Rodrigues’ formula assume ω is a unit vector and present $\exp[\theta\,\omega]$, resulting in no denominators in the coefficients. Getting rid of denominators might reduce the need for explicit series and therefore the risk of truncation error (TODO: consider rewriting all the code without denominators).

logSo3: SO(3) → $\mathfrak{so}(3)$

log takes us back to a canonical member of $\mathfrak{so}(3)$ on the principal branch.

- **θOver2sθ: θ/(2 Sin[θ])**

In[412]:=

```
ClearAll[θOver2sθ];
θOver2sθ[θ_] := If[θ < θThresh,
                                               
   1   θ²    7 θ⁴     31 θ⁶      127 θ⁸
   - + -- + ----- + ------- + ---------- ,
   2   12    720     30 240    1 209 600
   (* https://dlmf.nist.gov/4.19.4, http://oeis.org/A036280,
   http://oeis.org/A036281, Series[θ/(2Sin[θ]),{θ,0,9}] *)
      θ
   --------
   2 Sin[θ] ];
```

- **logSo3**

In[414]:=

```
ClearAll[logSo3];
logSo3[R_] :=
  With[{θ = ArcCos[1/2 (Tr[R] - 1)]},
   (R - Rᵀ) * θOver2sθ[θ]];
```

- **Unit Test**

Numerically, log and exp round trip to a part in 10^6 almost all the time. It fails occasionally at one part in 10^7, frequently enough that the following test of 10 000 trials will usually not succeed if `rounders` is 10^{-7}. Because `logSo3` canonicalizes $\omega_\times$, we must compare results in SO(3) and not in $\mathfrak{so}(3)$.

In[416]:=

```
Select[Table[
   With[{big = 10.0, rounder = 10^-6},
    With[{ω = RandomReal[{-big, big}, 3]},
     With[{RSO3 = expSO3[hatSo3[ω]]},
      With[{R2SO3 = expSO3[logSo3[RSO3]]},
       With[{
          lhs = N@Round[RSO3, rounder],
          rhs = N@Round[R2SO3, rounder]},
         lhs == rhs
       ]]]]], 10 000], # == False &] // Length
```

Out[416]=

```
0
```

logSo3symbolic

For some purposes, we need an exact, or symbolic, version of `logSo3`:

In[417]:=

```
ClearAll[logSo3symbolic];
logSo3symbolic[R_] :=
  With[{θ = ArcCos[1/2 (Tr[R] - 1)]},
   (R - R^T) * θ/(2 Sin[θ])];
```

Remarks

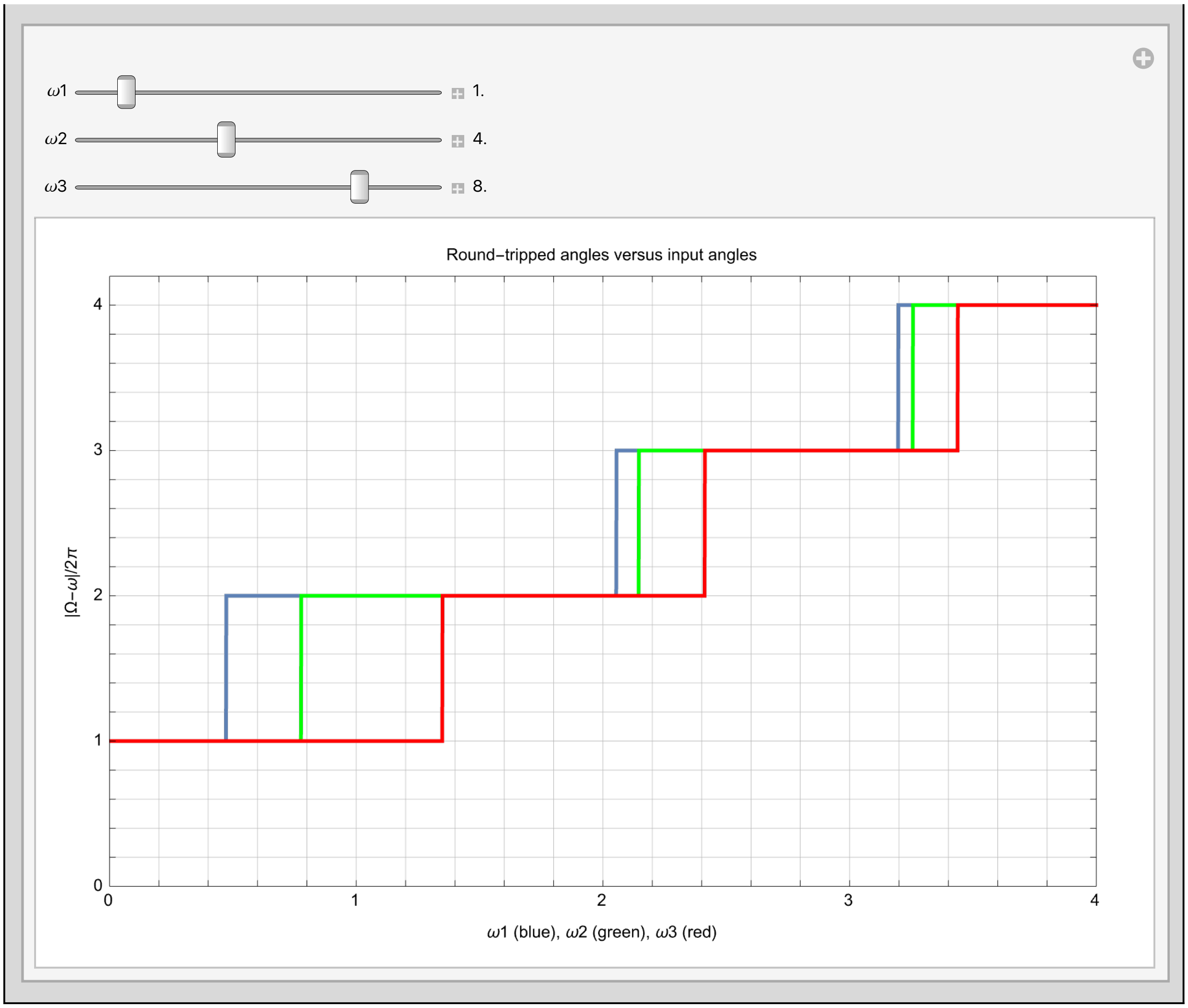
Multiplication in the reals is commutative; log on the reals converts a commutative product into a commutative sum. We might wish for something similar with rotation matrices, but it cannot be. Matrix products do not commute, but sums do. The log of XY cannot, in general, equal the log of YX. What is the log of a product of rotation matrices? The answer is in an application of the Baker-Campbell-Hausdorff formula.[11-13] This is non-trivial but not immediately needed in the current work.

Geometrically, $\omega \equiv \omega_\times$ is an ***axis-angle vector***. Its direction specifies the axis about which a twist occurs, and its length specifies the counterclockwise angle of twist in radians. Many different twist vectors map to the same rotation matrix. In the following demonstration, you may choose components for ω varying from 0 through some big number, and three sweeps are shown in red, green, and blue. Each sweep shows the difference in `Norm` (Euclidean length or magnitude) in units of 2π, between your chosen ω and `unhat[log[exp[hat[ω]]]]`, which latter specifies a canonical choice for the same rotation. Each sweep covers one of the three components, the other two being fixed by your interactive choices. This demonstration illustrates that `unhat[log[exp[hat[ω]]]` finds a the shortest twist vector (of smallest norm) that specifies the same rotation as the original choice, subtracting the maximum allowable multiples of 2π along the original direction of ω.

In[419]:=

```
With[{big = 10., maxsweep = 4, unit = 2 π}, Manipulate[
  With[{ω = {ω1, ω2, ω3}, uleh = unHatR3@*logSo3@*expSO3@*hatSo3,
    klugh = {Ω, ω} ↦ (Norm[Ω - ω] / unit)},
   Grid[{{Show[{
         Plot[klugh[{w1 * unit, ω2, ω3}, uleh[{w1 * unit, ω2, ω3}]],
          {w1, 0, maxsweep}, GridLines → Full, Frame → True,
          FrameLabel → {{"|Ω-ω|/2π", ""}, {"ω1 (blue), ω2 (green), ω3 (red)",
             "Round-tripped angles versus input angles"}},
          ImageSize → Large, PlotRange → {{0, maxsweep}, {0, 1.05 maxsweep}}],
         Plot[klugh[{ω1, w2 * unit, ω3}, uleh[{ω1, w2 * unit, ω3}]],
          {w2, 0, maxsweep}, ColorFunction → (Green &)],
         Plot[klugh[{ω1, ω2, w3 * unit}, uleh[{ω1, ω2, w3 * unit}]],
          {w3, 0, maxsweep}, ColorFunction → (Red &)]
        }], SpanFromLeft}}]],
  {{ω1, .10 * big}, 0, big, Appearance → "Labeled"},
  {{ω2, .4 * big}, 0, big, Appearance → "Labeled"},
  {{ω3, .8 * big}, 0, big, Appearance → "Labeled"}]]
```

Out[419]=



ω1 1.

ω2 4.

ω3 8.

Round–tripped angles versus input angles
|Ω−ω|/2π
0
1
2
3
4
0
1
2
3
4
ω1 (blue), ω2 (green), ω3 (red)

Just as the exponential map from $\theta \in \mathfrak{so}(2)$ to SO(2), $e^{i\theta} = \cos\theta + i\sin\theta$, maps many angles to the same complex-number “rotator,” the exponential map from $\omega_\times \in \mathfrak{so}(3)$ to SO(3) maps many $\omega_\times$ to the same rotation matrix. The log or pseudoinverse-exp map “unwinds” multiples of 2π on its principal branch. Similarly, from SO(2), the log or inverse-exp map is `ArcTan`, which also requires a choice of principal branch.

Adjoint in SO(3)

The adjoint Adj_R of element R of SO(3) is another 3×3 matrix defined by the equation

$$\exp[(\mathrm{Adj}_R \cdot \omega)_\times] = R \cdot \exp[\omega_\times] \cdot R^{\mathsf{T}} \tag{3}$$

ω is a three-vector inside the left-hand side and the skew-symmetric *hat* form $\omega_\times$ appears inside the right-hand side. The right-hand side’s expression is a matrix product, obviously in SO(3) because all its factors are in SO(3).

- **Unit Tests**

Reference [4] asserts that the adjoint of *R* in SO(3) is just *R* again. Check this assertion.

As noted above, we need a symbolic version of the exp map from 𝔰𝔬(3) to SO(3).

In[420]:=

```
ClearAll[expSO3symbolic];
expSO3symbolic[ωHat_] :=
 With[{ω = unHatR3[ωHat]},
  With[{θ = Sqrt[ω.ω]}, (* punning row and column vectors as lists *)
   IdentityMatrix[3] + Sin[θ]/θ ωHat + (1 - Cos[θ])/θ^2 ωHat.ωHat]]
```

The “theta” for pitch in our general rotation matrix is different from the θ in Rodrigues’ rotation formula, so we use here a “curly theta,” ϑ, for pitch.

The check below is an unevaluatable cell because it takes a couple of minutes to run, but its output is deterministic. Set the Cell Properties to Evaluatable in the *Mathematica* menu system if you’d like to check it.

```
(expSO3symbolic[hatSo3[R[ψ, ϑ, ϕ].{ω1, ω2, ω3}]]
 == R[ψ, ϑ, ϕ].expSO3symbolic[hatSo3[{ω1, ω2, ω3}]].(R[ψ, ϑ, ϕ]ᵀ)) //
 FullSimplify
```

Out[ ]=

```
True
```

For a numerical verification, sample the domain randomly.The following is good to one part in 10^{10} almost all the time. I have not seen a failure despite many evaluations; it fails occasionally with `rounder` at 10^{-11} and frequently at 10^{-12}:

In[422]:=

```
And @@ Table[With[{big = 10., rounder = 10^-10},
   Module[{
     ψ = RandomReal[{0, 2 π}],
     ϑ = RandomReal[{-π, π}],
     ϕ = RandomReal[{0, 2 π}],
     ω = RandomReal[{0, big}, 3],
     ωx, Rx, lhs, rhs},
    ωx = hatSo3[ω];
    Rx = R[ψ, ϑ, ϕ];
    lhs = N@Round[expSO3[hatSo3[Rx.ω]], rounder];
    rhs = N@Round[Rx.expSO3[ωx].Rx^T, rounder];
    lhs == rhs
   ]], 1000]
```

Out[422]=

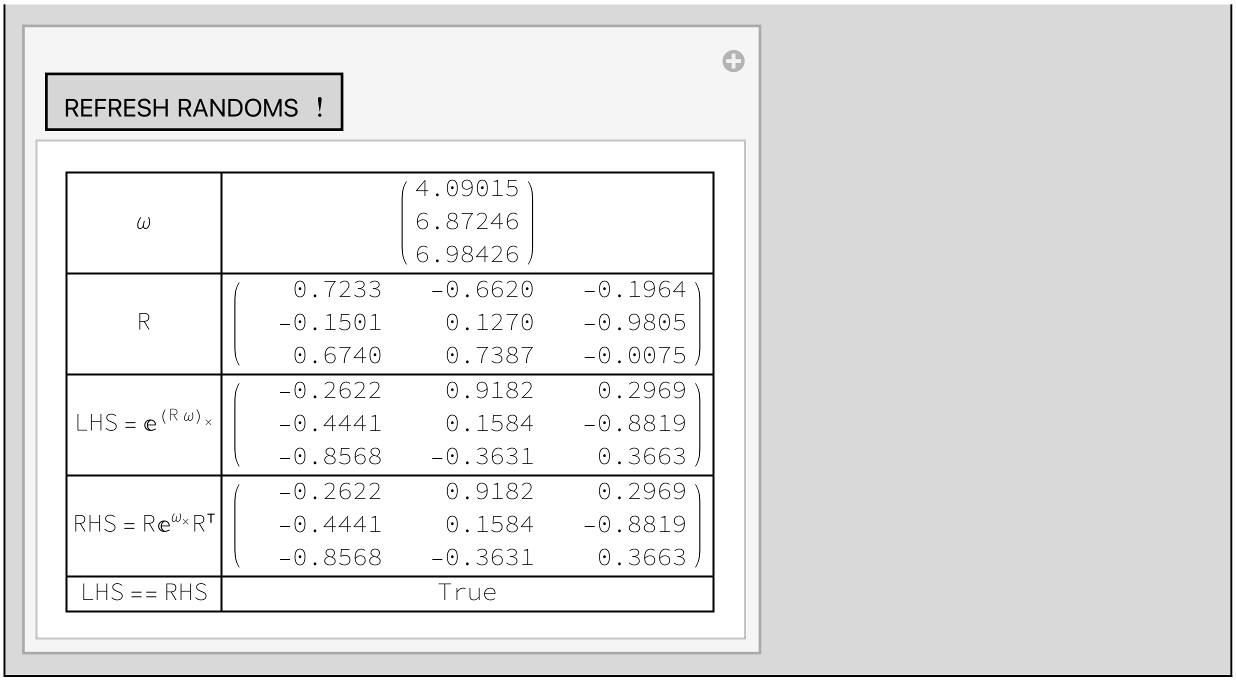
```
True
```

Here is an interactive version of the same demonstration. Expect the last two rows always to be equal.

In[423]:=

```
With[{big = 10., rounder = 10^-10},
 DynamicModule[{
   ψ = RandomReal[{0, 2 π}],
   ϑ = RandomReal[{-π, π}],
   ϕ = RandomReal[{0, 2 π}],
   ω = RandomReal[{0, big}, 3],
   ωx, Rx, lhs, rhs},
  Manipulate[
   ωx = hatSo3[ω];
   Rx = R[ψ, ϑ, ϕ];
   lhs = N@Round[expSO3[hatSo3[Rx.ω]], rounder];
   rhs = N@Round[Rx.expSO3[ωx].Rx^T, rounder];
   Grid[{
     {"ω", pretty@ω},
     {"R", pretty@Rx},
     {"LHS = e^(R ω)×", pretty@lhs},
     {"RHS = Re^ω×R^T", pretty@rhs},
     {"LHS == RHS", lhs == rhs}
    }, Frame → All],
   Control[Button[" REFRESH RANDOMS! ", (
       ψ = RandomReal[{0, 2 π}];
       ϑ = RandomReal[{-π, π}];
       ϕ = RandomReal[{0, 2 π}];
       ω = RandomReal[{0, big}, 3];
      ) &]]
  ]]]
```

Out[423]=



ω	$\begin{pmatrix}4.09015\\6.87246\\6.98426\end{pmatrix}$
R	$\begin{pmatrix}0.7233 & -0.6620 & -0.1964\\-0.1501 & 0.1270 & -0.9805\\0.6740 & 0.7387 & -0.0075\end{pmatrix}$
LHS = $e^{(R\,\omega)_\times}$	$\begin{pmatrix}-0.2622 & 0.9182 & 0.2969\\-0.4441 & 0.1584 & -0.8819\\-0.8568 & -0.3631 & 0.3663\end{pmatrix}$
RHS = $R e^{\omega_\times} R^T$	$\begin{pmatrix}-0.2622 & 0.9182 & 0.2969\\-0.4441 & 0.1584 & -0.8819\\-0.8568 & -0.3631 & 0.3663\end{pmatrix}$
LHS == RHS	True

SE(3)

Definitions

Elements of SE(3) are 4×4 matrices, with a 3×3 rotation from SO(3) in the Northwest block and a ***homogeneous*** 4-vector translation in the East column. The homogeneous 4-vector is a column vector in $\mathbb{R}^3$ with a 1 appended to the bottom. It represents a translation in 3-space. This representational trick is universal in computer graphics, even baked into graphics hardware, though not usually called out as a Lie-group method! Its usual motivation is that it converts affine transformations of the form $R_3 \cdot x_3 + t_3$ into linear transformations $R_4 \cdot x_4$, implemented more easily in GPU hardware.

SE3Form

For display:

In[424]:=

```
ClearAll[SE3Form];
SE3Form[m_] :=
  DisplayForm@
   RowBox[{"(",
     GridBox[m, GridBoxDividers → {
        "Columns" → {False, False, False, True},
        "Rows" → {False, False, False, True}}], ")"}];
```

pickSO3: SE(3) → SO(3)

Get the rotational block in SO (3) from an element of SE (3):

In[426]:=

```
ClearAll[pickSO3];
pickSO3[elemSE3_] := elemSE3⟦1 ;; 3, 1 ;; 3⟧;
```

Get the three-vector part of the homogeneous translation element out of an element of SE (3):

pickR3: SE(3) → $\mathbb{R}^3$

In[428]:=

```
ClearAll[pickR3];
pickR3[elemSE3_] := elemSE3⟦1 ;; 3, 4⟧;
```

$\mathfrak{se}(3)$

se3Form

Members of $\mathfrak{se}(3)$ are also 4 × 4 block matrices, so the same form works for them.

In[430]:=

```
ClearAll[se3Form];
se3Form = SE3Form;
```

hatSe3: $\mathbb{R}^6$ → $\mathfrak{se}(3)$, unHatR6: $\mathfrak{se}(3)$ → $\mathbb{R}^6$, unHat2R3: $\mathfrak{se}(3)$ → $\mathbb{R}^{2\times3}$

Elements of $\mathfrak{se}(3)$ have six independent real values, so a *hat* bijection with $\mathbb{R}^6$ exists. For convenience, we also support *hat* from a pair of members of $\mathbb{R}^3$.

In[432]:=

```
ClearAll[hatSe3, ω1, ω2, ω3, ux, uy, uz, vx, vy, vz];
hatSe3[ω1_, ω2_, ω3_, ux_, uy_, uz_] :=
  (  0  -ω3  ω2  ux )
  ( ω3   0  -ω1  uy )
  (-ω2  ω1   0   uz ) ;
  (  0   0   0   0  )
(* convenience overloads *)
(* from 2R3 *)
hatSe3[{{ω1_, ω2_, ω3_}, {ux_, uy_, uz_}}] := hatSe3[ω1, ω2, ω3, ux, uy, uz];
(* from two R3's *)
hatSe3[{ω1_, ω2_, ω3_}, {ux_, uy_, uz_}] := hatSe3[ω1, ω2, ω3, ux, uy, uz];
(* from R6 *)
hatSe3[{ω1_, ω2_, ω3_, ux_, uy_, uz_}] := hatSe3[ω1, ω2, ω3, ux, uy, uz];
hatSe3[ω1, ω2, ω3, ux, uy, uz] // se3Form
```

Out[437]//DisplayForm=

```
(  0  -ω3  ω2 |ux )
( ω3   0  -ω1 |uy )
(-ω2  ω1   0  |uz )
( ------------+-- )
(  0   0   0  |0  )
```

`unHat2R3` always produces a pair of members of $\mathbb{R}^3$, that is, a member of $\mathbb{R}^{2\times3}$, an element of type **2R3**:

In[438]:=

```
ClearAll[unHatR6, unHat2R3];
unHatR6 = Flatten@*unHat2R3;
unHat2R3[elemSe3_] := {unHatR3[pickSO3[elemSe3]], pickR3[elemSe3]};
```

Round-trip test:

In[441]:=

```
unHat2R3[hatSe3[( ω1 ω2 ω3 )]] // MatrixForm
                ( ux uy uz )
```

Out[441]//MatrixForm=

```
( ω1  ω2  ω3 )
( ux  uy  uz )
```

Despite appearances under `MatrixForm`, `unHatR6` doesn’t produce a proper column vector, but rather a flat list, which *Mathematica* often silently treats as a column vector:

In[442]:=

```
unHatR6[hatSe3[ω1, ω2, ω3, ux, uy, uz]] // MatrixForm
```

Out[442]//MatrixForm=

```
⎛ω1⎞
⎜ω2⎟
⎜ω3⎟
⎜ux⎟
⎜uy⎟
⎝uz⎠
```

but not for `Transpose`. The result is a flat list with curly braces, not a row vector with round parentheses.

In[443]:=

```
unHatR6[hatSe3[ω1, ω2, ω3, ux, uy, uz]] // Transpose
```

Out[443]=

```
{ω1, ω2, ω3, ux, uy, uz}
```

The following is a general rule or convention, worth highlighting prominently

To get a proper column vector, just map `List` over any flat list:

In[444]:=

```
List /@ unHatR6[hatSe3[ω1, ω2, ω3, ux, uy, uz]]
```

Out[444]=

```
{{ω1}, {ω2}, {ω3}, {ux}, {uy}, {uz}}
```

It (confusingly) has the same `MatrixForm` as a flat list

In[445]:=

```
(List /@ unHatR6[hatSe3[ω1, ω2, ω3, ux, uy, uz]]) // MatrixForm
```

Out[445]//MatrixForm=

```
⎛ω1⎞
⎜ω2⎟
⎜ω3⎟
⎜ux⎟
⎜uy⎟
⎝uz⎠
```

but will `Transpose`. The result is a row vector: a list of one row-as-a-list.

In[446]:=

```
(List /@ unHatR6[hatSe3[ω1, ω2, ω3, ux, uy, uz]]) // Transpose
```

Out[446]=

```
{{ω1, ω2, ω3, ux, uy, uz}}
```

Under `MatrixForm`, the transpose of a proper column vector ($n \times 1$ matrix) displays as a proper row vector (1 × *n* matrix) — with round parentheses instead of curly braces:

In[447]:=

```
(List /@ unHatR6[hatSe3[ω1, ω2, ω3, ux, uy, uz]]) // Transpose // MatrixForm
```

Out[447]//MatrixForm=

```
( ω1  ω2  ω3  ux  uy  uz )
```

expSE3: $\mathfrak{se}(3)$ → SE(3)

As before, factor out the numerical Taylor series of the coefficients:

In[448]:=

```
       θ - Sin[θ]
Series[----------, {θ, 0, 9}]
          θ³
```

Out[448]=

$$\frac{1}{6}-\frac{\theta^2}{120}+\frac{\theta^4}{5040}-\frac{\theta^6}{362\,880}+\frac{\theta^8}{39\,916\,800}+O[\theta]^{10}$$

- **oneMaOverθ2: (θ – Sin[θ]) / θ³**

In[449]:=

```
ClearAll[oneMaOverθ2];
oneMaOverθ2[A_, θ_] := If[θ < θThresh,
    1   θ²   θ⁴   θ⁶   θ⁸
    - - -- + -- - -- + ---,
    6   5!   7!   9!   11!
    1 - A
    -----];
     θ²
```

- **expSE3: $\mathfrak{se}(3)$ → SE(3)**

In[451]:=

```
ClearAll[expSE3];
expSE3[elemSe3_] :=
  Module[{u, ω, ωx, θ, A, B, C, R, V},
   {ω, u} = unHat2R3[elemSe3];
   θ = Sqrt[ω.ω];
   A = sθOverθ[θ];
   B = oneMcθOverθ2[θ];
   C = oneMaOverθ2[A, θ];
   ωx = hatSo3[ω];
   R = expSO3[ωx];
   V = IdentityMatrix[3] + B ωx + C ωx.ωx;
   appendColumn[R, List /@ (V.u)] ~ Join ~ {{0, 0, 0, 1}}];
```

- **Unit Test**

Next we show a particular element of SE(3) for a randomly generated element of $\mathfrak{se}(3)$:

In[453]:=

```
With[{big = 10.0},
  With[{elemSe3 = hatSe3 @@ RandomReal[{-big, big}, 6]},
   expSE3[elemSe3]]] // SE3Form
```

Out[453]//DisplayForm=

```
 0.803475  -0.434156  0.407353 | -0.588382
 0.372885   0.900396  0.224152 |  2.40875
-0.464096  -0.0282045 0.885336 |  2.44551
---------------------------------------------
     0          0         0    |     1
```

logSe3: SE(3) → $\mathfrak{se}(3)$, logR6: SE(3) → $\mathbb{R}^6$

We need the following series:

In[454]:=

```
Series[1/θ^2 (1 - (θ Sin[θ]) / (2 (1 - Cos[θ]))), {θ, 0, 9}]
```

Out[454]=

$$\frac{1}{12}+\frac{\theta^2}{720}+\frac{\theta^4}{30\,240}+\frac{\theta^6}{1\,209\,600}+\frac{\theta^8}{47\,900\,160}+O[\theta]^{10}$$

and a numerical version of the inverse of matrix *V* from the *exp* map above:

In[455]:=

```
ClearAll[vInv];
vInv[ωx_, θ_] :=
  With[{d = If[θ < θThresh,
        1/12 + θ^2/720 + θ^4/30240 + θ^6/1209600 + θ^8/47900160,
        1/θ^2 (1 - (θ Sin[θ])/(2 (1 - Cos[θ])))]},
   IdentityMatrix[3] - 1/2 ωx + d ωx.ωx];
```

The following, for any element of SE(3), picks a member of $\mathfrak{se}(3)$ from the principal branch.

In[457]:=

```
ClearAll[logSe3];
logSe3[elemSE3_] :=
  Module[{R = pickSO3@elemSE3, θ, ωx, vi, t, u, result},
   θ = ArcCos[1/2 (Tr[R] - 1)];
   ωx = logSo3[R];
   vi = vInv[ωx, θ];
   t = pickR3@elemSE3;
   u = vi.t;
   result = appendColumn[ωx, List /@ u] ~ Join ~ {{0, 0, 0, 0}};
   result];
```

- **Unit Test**

As with SO(3) and 𝔰𝔬(3), log and exp in the SE(3) and 𝔰𝔢(3) round-trip to one part in 10^6 almost all the time. There are rare failures, typically 1 in 10 000, at one part in 10^7.

In[459]:=

```
Select[Table[
   With[{big = 10.0, rounder = 10^-6},
    With[{uω = RandomReal[{-big, big}, 6]},
     With[{uωx = hatSe3[uω]},
      With[{E = expSE3[uωx]},
       With[{uωx2 = logSe3[E]},
        With[{E2 = expSE3[uωx2]},
         With[{
            lhs = N@Round[E, rounder],
            rhs = N@Round[E2, rounder]},
          lhs == rhs]
        ]]]]]], 10 000], # == False &] // Length
```

Out[459]=

```
0
```

ad6R6: 𝔰𝔢(3) → 𝔰𝔢(3) → $\mathbb{R}^6$, Lie Bracket Operator

- **Lie bracket *V*.*W* – *W*.*V* : 𝔰𝔢(3) × 𝔰𝔢(3) → 𝔰𝔢(3)**

In[460]:=

```
ClearAll[lieBracketSe3];
lieBracketSe3[Vse3_, Wse3_] := Vse3.Wse3 - Wse3.Vse3;
```

- **Unit Test**

Presented below, after the definition of `VR6` and `VhatSe3`.

ad6R6: operator form of the Lie bracket

Do not confuse this with the adjoint. I do not know why it’s written this way in the original references.

Consider the following operator form:

In[462]:=

```
ClearAll[ad6R6];
ad6R6[ω1_, ω2_, ω3_, vx_, vy_, vz_] :=
  Join[
   appendMatrix[hatSo3[ω1, ω2, ω3], ConstantArray[0, {3, 3}]],
   appendMatrix[hatSo3[vx, vy, vz], hatSo3[ω1, ω2, ω3]]];
(* Convenience overloads *)
ad6R6[{{ω1_, ω2_, ω3_}, {vx_, vy_, vz_}}] := ad6R6[ω1, ω2, ω3, vx, vy, vz];
ad6R6[{ω1_, ω2_, ω3_, vx_, vy_, vz_}] := ad6R6[ω1, ω2, ω3, vx, vy, vz];
ad6R6[ω1, ω2, ω3, vx, vy, vz] // MatrixForm
```

Out[466]//MatrixForm=

$$\begin{pmatrix} 0 & -\omega 3 & \omega 2 & 0 & 0 & 0 \\ \omega 3 & 0 & -\omega 1 & 0 & 0 & 0 \\ -\omega 2 & \omega 1 & 0 & 0 & 0 & 0 \\ 0 & -vz & vy & 0 & -\omega 3 & \omega 2 \\ vz & 0 & -vx & \omega 3 & 0 & -\omega 1 \\ -vy & vx & 0 & -\omega 2 & \omega 1 & 0 \end{pmatrix}$$

Unit Tests

Verify Equation 13 of Reference 2:

In[467]:=

```
With[{V = {ω1, ω2, ω3, vx, vy, vz}, W = {υ1, υ2, υ3, ux, uy, uz}},
   ad6R6[V].W] // hatSe3 // se3Form
```

Out[467]//DisplayForm=

$$\left(\begin{array}{ccc|c} 0 & -\upsilon 2\,\omega 1+\upsilon 1\,\omega 2 & -\upsilon 3\,\omega 1+\upsilon 1\,\omega 3 & -vz\,\upsilon 2+vy\,\upsilon 3+uz\,\omega 2-uy\,\omega 3 \\ \upsilon 2\,\omega 1-\upsilon 1\,\omega 2 & 0 & -\upsilon 3\,\omega 2+\upsilon 2\,\omega 3 & vz\,\upsilon 1-vx\,\upsilon 3-uz\,\omega 1+ux\,\omega 3 \\ \upsilon 3\,\omega 1-\upsilon 1\,\omega 3 & \upsilon 3\,\omega 2-\upsilon 2\,\omega 3 & 0 & -vy\,\upsilon 1+vx\,\upsilon 2+uy\,\omega 1-ux\,\omega 2 \\ \hline 0 & 0 & 0 & 0 \end{array}\right)$$

Powers

We’ll need powers of this operator form. Precompute a few for demonstration:

In[468]:=

```
With[{V = {ω₁, ω₂, ω₃, vₓ, v_y, v_z}},
    With[{a = ad6R6[V]},
     FoldList[Dot, ConstantArray[a, 3]]]] // FullSimplify //
  Map[MatrixForm] // Column
```

Out[468]=

$$\begin{pmatrix} 0 & -\omega_3 & \omega_2 & 0 & 0 & 0 \\ \omega_3 & 0 & -\omega_1 & 0 & 0 & 0 \\ -\omega_2 & \omega_1 & 0 & 0 & 0 & 0 \\ 0 & -v_z & v_y & 0 & -\omega_3 & \omega_2 \\ v_z & 0 & -v_x & \omega_3 & 0 & -\omega_1 \\ -v_y & v_x & 0 & -\omega_2 & \omega_1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} -\omega_2^2-\omega_3^2 & \omega_1\,\omega_2 & \omega_1\,\omega_3 & 0 & 0 & 0 \\ \omega_1\,\omega_2 & -\omega_1^2-\omega_3^2 & \omega_2\,\omega_3 & 0 & 0 & 0 \\ \omega_1\,\omega_3 & \omega_2\,\omega_3 & -\omega_1^2-\omega_2^2 & 0 & 0 & 0 \\ -2\,(v_y\,\omega_2+v_z\,\omega_3) & v_y\,\omega_1+v_x\,\omega_2 & v_z\,\omega_1+v_x\,\omega_3 & -\omega_2^2-\omega_3^2 & \omega_1\,\omega_2 & \omega_1\,\omega_3 \\ v_y\,\omega_1+v_x\,\omega_2 & -2\,(v_x\,\omega_1+v_z\,\omega_3) & v_z\,\omega_2+v_y\,\omega_3 & \omega_1\,\omega_2 & -\omega_1^2-\omega_3^2 & \omega_2\,\omega_3 \\ v_z\,\omega_1+v_x\,\omega_3 & v_z\,\omega_2+v_y\,\omega_3 & -2\,(v_x\,\omega_1+v_y\,\omega_2) & \omega_1\,\omega_3 & \omega_2\,\omega_3 & -\omega_1^2-\omega_2^2 \end{pmatrix}$$

$$\left(\begin{array}{ccc} 0 & \omega_3\,(\omega_1^2+\omega_2^2+\omega_3^2) & -\omega \\ -\omega_3\,(\omega_1^2+\omega_2^2+\omega_3^2) & 0 & \omega \\ \omega_2\,(\omega_1^2+\omega_2^2+\omega_3^2) & -\omega_1\,(\omega_1^2+\omega_2^2+\omega_3^2) & \\ 0 & 2\,(v_x\,\omega_1+v_y\,\omega_2)\,\omega_3+v_z\,(\omega_1^2+\omega_2^2+3\,\omega_3^2) & -2\,\omega_2\,(v_x\,\omega_1+ \\ -2\,(v_x\,\omega_1+v_y\,\omega_2)\,\omega_3-v_z\,(\omega_1^2+\omega_2^2+3\,\omega_3^2) & 0 & 2\,\omega_1\,(v_y\,\omega_2+ \\ 2\,\omega_2\,(v_x\,\omega_1+v_z\,\omega_3)+v_y\,(\omega_1^2+3\,\omega_2^2+\omega_3^2) & -2\,\omega_1\,(v_y\,\omega_2+v_z\,\omega_3)-v_x\,(3\,\omega_1^2+\omega_2^2+\omega_3^2) & \end{array}\right.$$

These get big fast, so it's better to compute powers numerically. These converge with the angular parts of the *V*'s in the canonical region [$-\pi/2$, $\pi/2$], but it's probably best to continue this series until it no longer grows because it doesn't seem reasonable to pick a fixed `nmax` ahead of time. The following demonstration fixed `nmax` at a small value and seldom converges. Notice that the odd terms beyond the second term are exactly zero due to the Bernoulli numbers' vanishing.

In[469]:=

```
With[{nmax = 4},
     With[{V = {RandomReal[{-π / 2, π / 2}, 3], RandomReal[{-100, 100}, 3]},
       bjjs = N@Table[BernoulliB[n] / n !, {n, nmax}]},
      With[{a = ad6R6[V]},
       With[{advjs = FoldList[Dot, ConstantArray[a, nmax]]},
        MapThread[Times, {bjjs, advjs}]]]]] //
    Chop // Map[pretty] // Map[MatrixForm] // Column
```

Out[469]=

```
        0     0.2759    0.1006         0         0         0
  -0.2759          0   -0.7036         0         0         0
  -0.1006     0.7036         0         0         0         0
        0    32.6641  -31.8291         0    0.2759    0.1006
 -32.6641          0  -41.8206   -0.2759         0   -0.7036
  31.8291    41.8206         0   -0.1006    0.7036         0

  -0.0287     0.0236   -0.0647         0         0         0
   0.0236    -0.1904   -0.0093         0         0         0
  -0.0647    -0.0093   -0.1684         0         0         0
  -3.8731    -6.0619  -11.5064   -0.0287    0.0236   -0.0647
  -6.0619   -25.6234    1.8318    0.0236   -0.1904   -0.0093
 -11.5064     1.8318  -17.4804   -0.0647   -0.0093   -0.1684

       0        0        0        0        0        0
       0        0        0        0        0        0
       0        0        0        0        0        0
       0        0        0        0        0        0
       0        0        0        0        0        0
       0        0        0        0        0        0

  -0.0011     0.0009   -0.0025         0         0         0
   0.0009    -0.0074   -0.0004         0         0         0
  -0.0025    -0.0004   -0.0065         0         0         0
  -0.2851    -0.1240   -0.7498   -0.0011    0.0009   -0.0025
  -0.1240    -1.8872    0.0275    0.0009   -0.0074   -0.0004
  -0.7498     0.0275   -1.4683   -0.0025   -0.0004   -0.0065
```

dlog6R6: $\mathfrak{se}(3) \to \mathfrak{se}\ (3) \to \mathbb{R}^6$, Inverse Right Trivialized Tangent Operator

See https://maxime-tournier.github.io/notes/lie-groups.html.

This version keeps computing the series until it no longer changes within machine double precision. It should produce maximum accuracy of the right-trivialized tangent operator. It also bypasses computation of odd-numbered terms, which are exactly zero, beyond the second, saving computation time.

It takes in any of the convenience forms that `ad6R6` takes.

In[470]:=

```
ClearAll[dlog6R6];
With[{
   bjj = j ↦ N[BernoulliB[j]] / N[j !],
   jcrash = 1000,
   a0 = IdentityMatrix[6]},
  dlog6R6[V_, jmax_ : jcrash] :=
   With[{a1 = ad6R6[V]},
    Module[{j, aj, lastAccumulator, accumulator},
     For[
      (*start*)
      j = 2;
      aj = a1;
      lastAccumulator = bjj[0] * a0;
      accumulator = lastAccumulator + bjj[1] * aj,

      (*test*)
      (j ≤ jmax) && (Chop[accumulator] =!= Chop[lastAccumulator]),

      (*incr*)
      j += 2,

      (*body*)
      lastAccumulator = accumulator;
      aj = aj.a1;
      accumulator = lastAccumulator + bjj[j] * aj
     ];
     <|"j" → j, "dlog[V]^6×6" → accumulator|>
    ]]];
```

■ Unit Tests

Here are few test cases, verified by Jeongseok Lee (first author of Reference 2; private communication). Use the 2×3 representation for elements of 𝔰𝔢(3) because we want different test ranges for the angular parts than for the velocity parts.

In[472]:=

```
Table[With[{V2R3 = {
      RandomReal[{-π / 2, π / 2}, 3],
      RandomReal[{-100, 100}, 3]}},
   AssociateTo[dlog6R6[V2R3], "V^2×3" → V2R3]], 5]
```

Out[472]=

```
{⟨|j → 28,
   dlog[V]^6×6 → {{0.770212, -0.722076, -0.398746, 0., 0., 0.}, {0.689731, 0.827197,
       -0.213962, 0., 0., 0.}, {0.452397, 0.0107538, 0.934476, 0., 0., 0.},
     {-25.5661, -38.2772, -24.8665, 0.770212, -0.722076, -0.398746},
     {25.7, -18.1945, -47.4082, 0.689731, 0.827197, -0.213962},
     {43.8297, 21.4414, -12.6193, 0.452397, 0.0107538, 0.934476}},
   V^2×3 → {{-0.226482, 0.857832, -1.4229}, {-69.6049, 70.0475, -65.826}}|⟩,
  ⟨|j → 26, dlog[V]^6×6 → {{0.935078, -0.415551, 0.14716, 0., 0., 0.},
     {0.399794, 0.937664, 0.164086, 0., 0., 0.}, {-0.185751, -0.118645, 0.984032,
      0., 0., 0.}, {7.05298, 42.6585, 48.2913, 0.935078, -0.415551, 0.14716},
     {-41.7453, 15.5845, -39.3188, 0.399794, 0.937664, 0.164086},
     {-32.6741, 45.6349, -0.48042, -0.185751, -0.118645, 0.984032}},
   V^2×3 → {{0.283407, -0.333708, -0.817296}, {-85.2618, -81.0356, 84.9081}}|⟩,
  ⟨|j → 24, dlog[V]^6×6 → {{0.978318, 0.17452, 0.187237, 0., 0., 0.},
     {-0.132876, 0.960641, -0.317134, 0., 0., 0.}, {-0.218767, 0.296265, 0.95476,
      0., 0., 0.}, {4.92797, -23.4941, -18.1986, 0.978318, 0.17452, 0.187237},
     {13.5077, 10.1035, 42.2272, -0.132876, 0.960641, -0.317134},
     {26.1135, -37.4205, 11.2147, -0.218767, 0.296265, 0.95476}},
   V^2×3 → {{-0.614487, -0.406725, 0.307942}, {80.0582, 44.5689, -37.2024}}|⟩,
  ⟨|j → 32, dlog[V]^6×6 → {{0.663657, 0.620258, 0.793224, 0., 0., 0.},
     {-0.857244, 0.71921, 0.356636, 0., 0., 0.}, {-0.528259, -0.690895, 0.75659,
      0., 0., 0.}, {34.6378, -8.11121, -60.9883, 0.663657, 0.620258, 0.793224},
     {45.2208, 30.5942, -27.3887, -0.857244, 0.71921, 0.356636},
     {27.7198, 62.9647, 37.053, -0.528259, -0.690895, 0.75659}},
   V^2×3 → {{1.06276, -1.34069, 1.49898}, {-93.501, 92.3116, -56.6945}}|⟩,
  ⟨|j → 30, dlog[V]^6×6 → {{0.853626, -0.0260719, 0.670792, 0., 0., 0.},
     {-0.24568, 0.866156, 0.59446, 0., 0., 0.}, {-0.624726, -0.642698, 0.727959,
      0., 0., 0.}, {-16.8531, -27.0466, 36.2989, 0.853626, -0.0260719, 0.670792},
     {20.8523, 12.8609, -22.8425, -0.24568, 0.866156, 0.59446},
     {-48.3047, 30.1963, -7.55638, -0.624726, -0.642698, 0.727959}},
   V^2×3 → {{1.24864, -1.30755, 0.221647}, {-53.2889, -85.6427, -48.3006}}|⟩}
```

Adj6R6: Adjoint Operator in SE(3)

In[473]:=

```
ClearAll[Adj6R6];
Adj6R6[TconfigSE3_] :=
  With[{R = pickSO3[TconfigSE3], pHat = hatSo3[pickR3[TconfigSE3]]},
   Join[
    appendMatrix[R, ConstantArray[0, {3, 3}]],
    appendMatrix[pHat.R, R]]];
```

- **Unit Test**

Presented below, after the definition of `VR6` and `VhatSe3`.

Lagrangian

TSE3: Configuration

- **TSE3**

Represent the configuration of (the body-fixed, center-of-gravity, principal-axis frame of) any rigid body as an element of SE(3). In the yaw-pitch-roll parameterization for SO(3). The initial conditions below are the same as we had for *Dzhanybekhov* above.

In[475]:=

```
initωb$ = {6., 0., 0.01};
initQuat$ = rq[π / 4., {0., 1.0, 0.}];
```

In[477]:=

```
ClearAll[TSE3, ψ, θ, ϕ, x, y, z];
TSE3[ψ_, θ_, ϕ_, x_, y_, z_] :=
  appendColumn[R[ψ, θ, ϕ], {{x}, {y}, {z}}]~Join~{{0, 0, 0, 1}};
TSE3[q : Quaternion[_, _, _, _], d : {_, _, _}] :=
  appendColumn[rotMatFromQ[q], (List /@ d)]~Join~{{0, 0, 0, 1}};
          ⎛_ _ _⎞
TSE3[r :  ⎜_ _ _⎟, d : {_, _, _}] := appendColumn[r, (List /@ d)]~Join~{{0, 0, 0, 1}}
          ⎝_ _ _⎠
```

- **Unit Tests**

In[481]:=

```
TSE3[ψ, θ, ϕ, x, y, z] /. shorteningRules // SE3Form
TSE3[initQuat$, {0, 0, 0}] // SE3Form
TSE3[Quaternion[w, i, j, k], {x, y, z}] // SE3Form
TSE3[R[ψ, θ, ϕ], {x, y, z}] /. shorteningRules // SE3Form
```

Out[481]//DisplayForm=

$$\left(\begin{array}{ccc|c} C_\theta\ C_\psi & -C_\theta\ S_\psi & S_\theta & x \\ C_\psi\ S_\theta\ S_\phi + C_\phi\ S_\psi & C_\phi\ C_\psi - S_\theta\ S_\phi\ S_\psi & -C_\theta\ S_\phi & y \\ -C_\phi\ C_\psi\ S_\theta + S_\phi\ S_\psi & C_\psi\ S_\phi + C_\phi\ S_\theta\ S_\psi & C_\theta\ C_\phi & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

Out[482]//DisplayForm=

$$\left(\begin{array}{ccc|c} 0.707107 & 0. & 0.707107 & 0 \\ 0. & 1. & 0. & 0 \\ -0.707107 & 0. & 0.707107 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

Out[483]//DisplayForm=

$$\left(\begin{array}{ccc|c} i^2 - j^2 - k^2 + w^2 & 2\ i\ j - 2\ k\ w & 2\ i\ k + 2\ j\ w & x \\ 2\ i\ j + 2\ k\ w & -i^2 + j^2 - k^2 + w^2 & 2\ j\ k - 2\ i\ w & y \\ 2\ i\ k - 2\ j\ w & 2\ j\ k + 2\ i\ w & -i^2 - j^2 + k^2 + w^2 & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

Out[484]//DisplayForm=

$$\left(\begin{array}{ccc|c} C_\theta\ C_\psi & -C_\theta\ S_\psi & S_\theta & x \\ C_\psi\ S_\theta\ S_\phi + C_\phi\ S_\psi & C_\phi\ C_\psi - S_\theta\ S_\phi\ S_\psi & -C_\theta\ S_\phi & y \\ -C_\phi\ C_\psi\ S_\theta + S_\phi\ S_\psi & C_\psi\ S_\phi + C_\phi\ S_\theta\ S_\psi & C_\theta\ C_\phi & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

VR6, VhatSe3: Velocity in $\mathfrak{se}(3)$

Equation 5 of Reference 2, with two representations of *V*: as a proper column vector (list of lists) and as a 4 × 4 block matrix.

In[485]:=

```
ClearAll[VR6, VhatSe3, ω1, ω2, ω3, vx, vy, vz];
VR6[ω1_, ω2_, ω3_, vx_, vy_, vz_] := Transpose@{{ω1, ω2, ω3, vx, vy, vz}};
VhatSe3[ω1_, ω2_, ω3_, vx_, vy_, vz_] :=
  appendColumn[hatSo3[ω1, ω2, ω3], {{vx}, {vy}, {vz}}]~Join~{{0, 0, 0, 0}};
VR6[ω1, ω2, ω3, vx, vy, vz] // MatrixForm
VhatSe3[ω1, ω2, ω3, vx, vy, vz] // se3Form
```

Out[488]//MatrixForm=

$$\begin{pmatrix} \omega 1 \\ \omega 2 \\ \omega 3 \\ \mathrm{vx} \\ \mathrm{vy} \\ \mathrm{vz} \end{pmatrix}$$

Out[489]//DisplayForm=

$$\left(\begin{array}{ccc|c} 0 & -\omega 3 & \omega 2 & \mathrm{vx} \\ \omega 3 & 0 & -\omega 1 & \mathrm{vy} \\ -\omega 2 & \omega 1 & 0 & \mathrm{vz} \\ \hline 0 & 0 & 0 & 0 \end{array}\right)$$

- **Unit Tests for lieBracketSe3**

In[490]:=

```
With[{
   V = VhatSe3[ω1, ω2, ω3, vx, vy, vz],
   W = VhatSe3[υ1, υ2, υ3, ux, uy, uz]},
  lieBracketSe3[V, W]] // se3Form
```

Out[490]//DisplayForm=

$$\left(\begin{array}{ccc|c} 0 & -\upsilon 2\,\omega 1 + \upsilon 1\,\omega 2 & -\upsilon 3\,\omega 1 + \upsilon 1\,\omega 3 & -\mathrm{vz}\,\upsilon 2 + \mathrm{vy}\,\upsilon 3 + \mathrm{uz}\,\omega 2 - \mathrm{uy}\,\omega 3 \\ \upsilon 2\,\omega 1 - \upsilon 1\,\omega 2 & 0 & -\upsilon 3\,\omega 2 + \upsilon 2\,\omega 3 & \mathrm{vz}\,\upsilon 1 - \mathrm{vx}\,\upsilon 3 - \mathrm{uz}\,\omega 1 + \mathrm{ux}\,\omega 3 \\ \upsilon 3\,\omega 1 - \upsilon 1\,\omega 3 & \upsilon 3\,\omega 2 - \upsilon 2\,\omega 3 & 0 & -\mathrm{vy}\,\upsilon 1 + \mathrm{vx}\,\upsilon 2 + \mathrm{uy}\,\omega 1 - \mathrm{ux}\,\omega 2 \\ \hline 0 & 0 & 0 & 0 \end{array}\right)$$

Notice the following identity between the Lie bracket and some traditional Gibbs cross products, taking advantage of a convenience overload of `hatSe3`:

In[491]:=

```
With[{ω = {ω1, ω2, ω3}, υ = {υ1, υ2, υ3}, v = {vx, vy, vz}, u = {ux, uy, uz}},
  hatSe3[ω × υ, ω × u - υ × v]] // se3Form
```

Out[491]//DisplayForm=

$$\left(\begin{array}{ccc|c} 0 & -\upsilon 2\,\omega 1 + \upsilon 1\,\omega 2 & -\upsilon 3\,\omega 1 + \upsilon 1\,\omega 3 & -\mathrm{vz}\,\upsilon 2 + \mathrm{vy}\,\upsilon 3 + \mathrm{uz}\,\omega 2 - \mathrm{uy}\,\omega 3 \\ \upsilon 2\,\omega 1 - \upsilon 1\,\omega 2 & 0 & -\upsilon 3\,\omega 2 + \upsilon 2\,\omega 3 & \mathrm{vz}\,\upsilon 1 - \mathrm{vx}\,\upsilon 3 - \mathrm{uz}\,\omega 1 + \mathrm{ux}\,\omega 3 \\ \upsilon 3\,\omega 1 - \upsilon 1\,\omega 3 & \upsilon 3\,\omega 2 - \upsilon 2\,\omega 3 & 0 & -\mathrm{vy}\,\upsilon 1 + \mathrm{vx}\,\upsilon 2 + \mathrm{uy}\,\omega 1 - \mathrm{ux}\,\omega 2 \\ \hline 0 & 0 & 0 & 0 \end{array}\right)$$

- **Unit Tests for Adj6R6**

In[492]:=

```
Adj6R6[TSE3[ψ, θ, ϕ, p_x, p_y, p_z]] /. shorteningRules // MatrixForm
```

Out[492]//MatrixForm=

$$\left(\begin{array}{ccc}
C_\theta\ C_\psi & -C_\theta\ S_\psi & \\
C_\psi\ S_\theta\ S_\phi + C_\phi\ S_\psi & C_\phi\ C_\psi - S_\theta\ S_\phi\ S_\psi & -(\\
-C_\phi\ C_\psi\ S_\theta + S_\phi\ S_\psi & C_\psi\ S_\phi + C_\phi\ S_\theta\ S_\psi & C \\
-p_z\ (C_\psi\ S_\theta\ S_\phi + C_\phi\ S_\psi) + p_y\ (-C_\phi\ C_\psi\ S_\theta + S_\phi\ S_\psi) & p_y\ (C_\psi\ S_\phi + C_\phi\ S_\theta\ S_\psi) - p_z\ (C_\phi\ C_\psi - S_\theta\ S_\phi\ S_\psi) & C_\theta\ C_\phi\ p_y \\
C_\theta\ C_\psi\ p_z - p_x\ (-C_\phi\ C_\psi\ S_\theta + S_\phi\ S_\psi) & -C_\theta\ p_z\ S_\psi - p_x\ (C_\psi\ S_\phi + C_\phi\ S_\theta\ S_\psi) & -C_\theta\ C_\phi \\
-C_\theta\ C_\psi\ p_y + p_x\ (C_\psi\ S_\theta\ S_\phi + C_\phi\ S_\psi) & C_\theta\ p_y\ S_\psi + p_x\ (C_\phi\ C_\psi - S_\theta\ S_\phi\ S_\psi) & -p_y\ S_\theta
\end{array}\right.$$

In[493]:=

```
With[{big = 10.},
 With[{bigRange = {-big, big}},
  With[{Vcomponents = RandomReal[bigRange, 6]},
   With[{VSe3 = VhatSe3 @@ Vcomponents,
     Vr6 = VR6 @@ Vcomponents,
     TsE3 = TSE3 @@ RandomReal[bigRange, 6]},
    Print[<|
       "[V]" → (VSe3 // MatrixForm),
       "Vr6" → (Vr6 // MatrixForm),
       "TsE3" → (TsE3 // MatrixForm),
       "[AdT]" → (Adj6R6[TsE3] // MatrixForm),
       "[AdT].Vr6" → (Adj6R6[TsE3].Vr6 // MatrixForm),
       "AdTVSe3" → (unHatR6[TsE3.VSe3.Inverse[TsE3]] // Chop // MatrixForm)|>];
   ]]]]
```

```
         (    0      -2.64711  -6.1757    7.66837  )
         ( 2.64711      0      6.29826   -6.76245  )
⟨| [V] → ( 6.1757   -6.29826     0      -0.385756  ),
         (    0         0        0          0      )

       ( -6.29826 )
       ( -6.1757  )          ( 0.0904754  -0.767949   0.634089   -7.33609  )
       ( 2.64711  )          ( 0.350569   -0.57139   -0.742034    6.57273  )
Vr6 →  ( 7.66837  ), TsE3 →  ( 0.932156    0.289428   0.217523   -0.123235 ),
       ( -6.76245 )          (    0           0          0           1     )
       ( -0.385756)

        ( 0.0904754  -0.767949   0.634089       0           0           0      )
        ( 0.350569   -0.57139   -0.742034       0           0           0      )
        ( 0.932156    0.289428   0.217523       0           0           0      )
[AdT] → ( 6.17002     1.83192    1.33827    0.0904754  -0.767949   0.634089  ),
        ( 6.82723     2.21791    1.51762    0.350569   -0.57139   -0.742034  )
        ( -3.16648    9.23929    1.27593    0.932156    0.289428   0.217523  )

              (  5.85129  )              (  5.85129  )
              ( -0.643492 )              ( -0.643492 )
              ( -7.08258  )              ( -7.08258  )
[AdT].Vr6 →   ( -40.9888  ), AdTVSe3 →   ( -40.9888  ) |⟩
              ( -45.841   )              ( -45.841   )
              ( -28.6313  )              ( -28.6313  )
```

In[494]:=

```
With[{reps = 1000},
 Select[Table[
    With[{big = 10., rounder = 10^-6},
     With[{bigRange = {-big, big}},
      With[{Vcomponents = RandomReal[bigRange, 6]},
       With[{VSe3 = VhatSe3 @@ Vcomponents,
         Vr6 = VR6 @@ Vcomponents,
         TsE3 = TSE3 @@ RandomReal[bigRange, 6]},
        With[{
          adtvr6 = Adj6R6[TsE3].Vr6,
          adtvse3 = List /@ unHatR6[TsE3.VSe3.Inverse[TsE3]]},
         With[{lhs = N[Round[adtvr6, rounder]],
           rhs = N[Round[adtvse3, rounder]]},
          lhs === rhs
         ]]]]]], reps], # === False &] // Length]
```

Out[494]=

```
0
```

PR, PrigR: Potential Energy

l is the half-length of the baton of 4DISP. The potential energy is *mg l* Cos[*θ*] Cos[*ϕ*]. The cosines can be retrieved from element [[3, 3]] of the configuration $T \in$ SE(3). The `R` suffix is a reminder that the potential energy is a member of ℝ, the real numbers. `PRigR` fixes `PR` for specific mass and half-length, yielding a function of configuration in TSE3 alone.

In[495]:=

```
ClearAll[PR];
PR[TSE3_, m_, g_, l_] := With[{cθcϕ = TSE3⟦3, 3⟧}, m g l cθcϕ];
```

In[497]:=

```
ClearAll[PrigR];
PrigR = With[{m = rig["mass"], g = -9.81, l = rig["half length"]},
   TSE3 ↦ PR[TSE3, m, g, l]];
```

G6R6: Grig6R6: Inertial Matrix

We choose a 6×6 representation for the inertial matrix *G* to ease the computation of kinetic energy. With the $\mathbb{R}^6$ representation for velocity, the kinetic energy is just half of $V^T G\, V$. *G* includes a mass block as well as the moment of inertia.

Numerical examples for 4DISP and for *Dzhanybekhov* are shown:

In[499]:=

```
ClearAll[Grig6R6, G6R6];
G6R6[𝐼_, m_] := With[{zero3 = ConstantArray [0, {3, 3}]},
   appendMatrix[𝐼, zero3] ~ Join ~ appendMatrix[zero3, m IdentityMatrix[3]]];
(Grig6R6 = G6R6[rig["moment of inertia"]〚1〛, rig["mass"]]) // MatrixForm
```

Out[501]//MatrixForm=

```
⎛0.4801    0.      0.     0   0   0 ⎞
⎜  0.    0.4801    0.     0   0   0 ⎟
⎜  0.      0.    0.0002   0   0   0 ⎟
⎜  0       0       0     1.  0.  0. ⎟
⎜  0       0       0     0.  1.  0. ⎟
⎝  0       0       0     0.  0.  1. ⎠
```

In[502]:=

```
ClearAll[Gdzhany6R6];
(Gdzhany6R6 = G6R6[dzhanybekhov["moment of inertia"]〚1〛,
     dzhanybekhov["mass"]]) // MatrixForm
```

Out[503]//MatrixForm=

```
⎛0.0801587      0.          0.       0 0 0⎞
⎜    0.      0.0825397      0.       0 0 0⎟
⎜    0.         0.      0.00396825   0 0 0⎟
⎜    0          0           0        1 0 0⎟
⎜    0          0           0        0 1 0⎟
⎝    0          0           0        0 0 1⎠
```

LR, LrigR: Lagrangian

As a (desirable) consequence of using proper column vectors (lists of lists), inner products have the type of a 1 × 1 matrix. The price to pay is that we must fish the scalar out via the `Part` notation [[1, 1]].

Equation 6 of Reference 2, again with an example from 4DISP’s *rig*:

In[504]:=

```
ClearAll[LR];
                        1
LR[VR6_, G6R6_, PR_] := ─ (VR6ᵀ.G6R6.VR6)〚1, 1〛 - PR;
                        2
```

In[506]:=

```
ClearAll[LrigR];
LrigR[ψ_, θ_, ϕ_, x_, y_, z_, ω1_, ω2_, ω3_, vx_, vy_, vz_] :=
  With[{
    tt = TSE3[ψ, θ, ϕ, x, y, z],
    vv = VR6[ω1, ω2, ω3, vx, vy, vz],
    m = rig["mass"],
    g = 9.81,
    l = rig["half length"],
    𝓘 = rig["moment of inertia"]⟦1⟧},
   With[{
     pp = PR[tt, m, g, l],
     gg = G6R6[𝓘, m]},
    LR[vv, gg, pp]
   ]];
LrigR @@ ConstantArray[0, 12]
```

Out[508]=

```
-11.772
```

This value matches the display from the ground truth.

LdRigR: Trapezoidal Quadrature

Equation 9 of Reference 2, writing *h* instead of Δ*t*. This has units of action, rather than units of energy like the non-discrete Lagrangian. The “units mistake” is universal in the literature, so we live with it.

In[509]:=

```
ClearAll[LdRigR];
With[{
   m = rig["mass"],
   g = 9.81,
   l = rig["half length"],
   𝓘 = rig["moment of inertia"]⟦1⟧},
  With[{G = G6R6[𝓘, m]},
   LdRigR[Tk_, Tkp1_, h_ : 0.01] :=
    With[{ΔTk = Inverse[Tk].Tkp1},
     With[{Vk = List /@ (1/h unHatR6[logSe3[ΔTk]])},
      With[{
        Lk = LR[Vk, G, PR[Tk, m, g, l]],
        Lkp1 = LR[Vk, G, PR[Tkp1, m, g, l]]},
       (*Print[<|"ΔTk"→ΔTk,"Vk"→Vk|>,"Lk"→Lk];*)
       h/2 (Lk + Lkp1)]]]]];
LdRigR[TSE3 @@ RandomReal[{-10, 10}, 6], TSE3 @@ RandomReal[{-10, 10}, 6]]
```

Out[511]=

```
2488.36
```

Variational Integrators in SE (3) (TODO)

Here are the discrete, reduced, Euler-Poincaré equations (**DREPE**; equations 16 and 17 of reference [2]). These are transcribed up to uncertainty about the meaning of T^{k*}, undefined in the paper, but possibly an adjoint representation or a member of the co-tangent space of T^k. We took a guess via `Transpose`, which converts vectors to co-vectors, and verified it on the *Dzhanybekhov* benchmark. (TODO: this is almost certainly wrong and works only by accident on *Dzhanybekhov*.)

$F^k \in \mathfrak{se}^*(3)$ is the integral of the virtual work of external forces over time interval h, loosely represented by a flat list in $\mathbb{R}^6$. Properly, it should be a row vector in $\mathbb{R}^{1\times 6}$ (TODO: fix this mess).

In[512]:=

```
ClearAll[tkStarR6];
tkStarR6[TkSE3_] := (unHatR6@*Transpose)[TkSE3];
(* (unHatR6@*Transpose@*Inverse)[TkSE3] ? *)

ClearAll[ΔTSE3];
ΔTSE3[TkmSE3_, TkSE3_] := Inverse[TkmSE3].TkSE3;

ClearAll[D2Ld, D1Ld, DREPE];
```

```
With[{print = Null &}, (* change to "Print" or "Echo" if you want to *)
  (* "km" abbreviates "k-1" *)
  D2Ld[TkmSE3_, TkSE3_, G6R6_, PR_, h_ : 0.01] :=
   With[{ΔTkmSE3 = ΔTSE3[TkmSE3, TkSE3]},
    With[{hVkmSe3 = logSe3[ΔTkmSE3]}, (* canonicalize ΔT *)
     With[{exphVkmSE3 = expSE3[hVkmSe3]}, (* back into SE(3) *)
      With[{AdjTexphVkm6R6 = Transpose[Adj6R6[exphVkmSE3]]},
       With[{hVkmR6 = unHatR6[hVkmSe3]},
        With[{dlogThVkm6R6 = Transpose[dlog6R6[hVkmR6]["dlog[V]^{6×6}"]]},
                     1
         With[{term1 = - (-AdjTexphVkm6R6.dlogThVkm6R6.G6R6.hVkmR6)},
                     h
          With[{TkStarR6 = tkStarR6[TkSE3]},
                       h
           With[{term2 = - PR[TkSE3] TkStarR6},
                       2
            With[{result = term1 + term2},
             print[<|
               "TkSE3" → SE3Form[TkSE3],
               "TkmSE3" → SE3Form[TkmSE3],
               "ΔTkmSE3" → SE3Form[ΔTkmSE3],
               "exphVkmSE3" → SE3Form[exphVkmSE3],
               "AdjTexphVkm6R6" → MatrixForm[AdjTexphVkm6R6],
               "hVkmR6" → MatrixForm[hVkmR6],
               "dlogThVkm6R6" → MatrixForm[dlogThVkm6R6],
               "G6R6" → MatrixForm[G6R6],
               "TkStarR6" → MatrixForm[TkStarR6],
               "D2Ld.term1" → MatrixForm[term1],
               "D2Ld.term2" → MatrixForm[term2], "D2Ld.result" → result|>];
             result]]]]]]]]]];
  (* "km" abbreviates "k-1" *)
  D1Ld[TkSE3_, TkpSE3_, G6R6_, PR_, h_ : 0.01] :=
   With[{ΔTkSE3 = ΔTSE3[TkSE3, TkpSE3]},
    With[{hVkSe3 = logSe3[ΔTkSE3]},
     With[{hVkR6 = unHatR6[hVkSe3]},
      With[{dlogThVk6R6 = Transpose[dlog6R6[hVkR6]["dlog[V]^{6×6}"]]},
                     dlogThVk6R6.G6R6.hVkR6
       With[{term1 = ----------------------},
                               h
        With[{TkStarR6 = tkStarR6[TkSE3]},
                     h
         With[{term2 = - PR[TkSE3] TkStarR6},
                     2
          With[{result = term1 + term2},
           print[<|
             "TkSE3" → SE3Form[TkSE3],
             "TkpSE3" → SE3Form[TkpSE3],
             "ΔTkSE3" → SE3Form[ΔTkSE3],
             "hVkSe3" → se3Form[hVkSe3],
             "hVkR6" → MatrixForm[hVkR6],
             "dlogThVk6R6" → MatrixForm[dlogThVk6R6],
             "G6R6" → MatrixForm[G6R6],
             "TkStarR6" → MatrixForm[TkStarR6],
             "D1Ld.term1" → MatrixForm[term1],
             "D1Ld.term2" → MatrixForm[term2],
             "D1Ld.result" → result|>];
           result]]]]]]]];
  DREPE[TkmSE3_, TkSE3_, TkpSE3_, G6R6_, PR_, FkR6_, h_ : 0.01] :=
   D2Ld[TkmSE3, TkSE3, G6R6, PR, h] + D1Ld[TkSE3, TkpSE3, G6R6, PR, h] + FkR6;];
```

■ Unit Tests

In[518]:=

```
With[{big = 10, o6 = ConstantArray[0, 6]},
 With[{bigRange = {-big, big}},
  With[{tk = TSE3 @@ RandomReal[bigRange, 6],
    tkp = TSE3 @@ RandomReal[bigRange, 6],
    tkm = TSE3 @@ RandomReal[bigRange, 6]},
   <|"D2Ld" → D2Ld[tkm, tk, Grig6R6, PrigR],
    "D1Ld" → D1Ld[tk, tkp, Grig6R6, PrigR],
    "DREPE" → DREPE[tkm, tk, tkp, Grig6R6, PrigR, o6] |>]]]

With[{big = 10, o6 = ConstantArray[0, 6]},
 With[{bigRange = {-big, big}},
  With[{tk = TSE3 @@ RandomReal[bigRange, 6],
    tkp = TSE3 @@ RandomReal[bigRange, 6],
    tkm = TSE3 @@ RandomReal[bigRange, 6]},
   <|"D2Ld" → D2Ld[tkm, tk, Gdzhany6R6, 0 &],
    "D1Ld" → D1Ld[tk, tkp, Gdzhany6R6, 0 &],
    "DREPE" → DREPE[tkm, tk, tkp, Gdzhany6R6, 0 &, o6] |>]]]
```

Out[518]=

```
<|D2Ld → {159.338, 3400.6, -531.097, 1053.98, -383.65, -1019.61},
 D1Ld → {499.624, 5514.69, -1279.54, 2141.23, -149.637, 367.372},
 DREPE → {658.962, 8915.29, -1810.64, 3195.21, -533.288, -652.238} |>
```

Out[519]=

```
<|D2Ld → {11 545., 15 539.1, 4169.67, -2725.08, 1218.78, 1805.2},
 D1Ld → {299.755, -257.148, 22.3662, -154.717, -39.8258, 635.147},
 DREPE → {11 844.8, 15 281.9, 4192.04, -2879.8, 1178.95, 2440.35} |>
```

One way to go about finding a root for T^{k+1} is to integrate forward in SO(3) using RK4. Reference 2 does not find roots this way.

In[520]:=

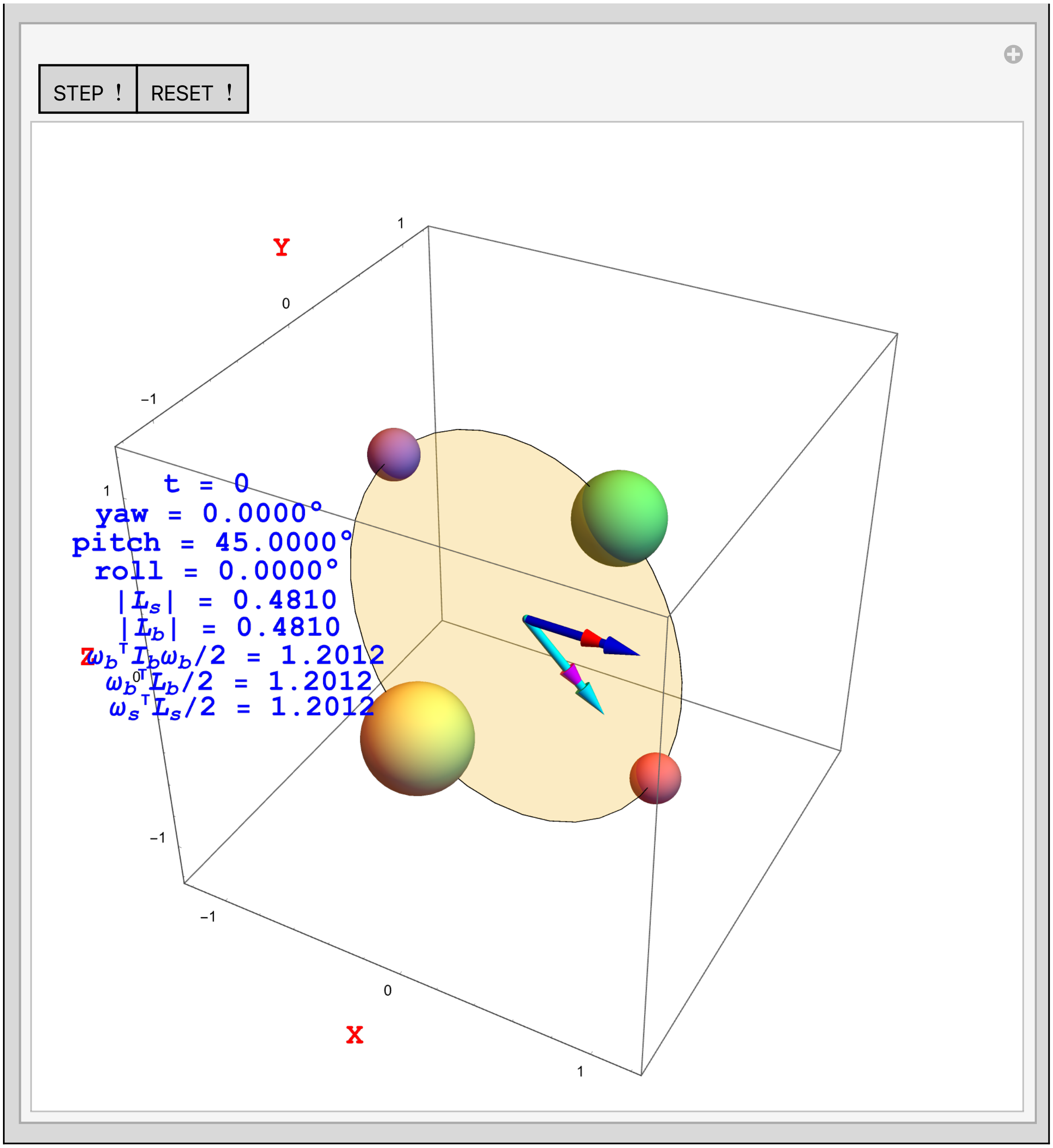
```
ClearAll[showSO3Apparatus];
showSO3Apparatus[t_, ωb_, Lb_, ωs_, Ls_, Ib_, rotMatSO3_, apparatus_] :=
  (Module[{placeZ = 1, y, p, r},
    With[{arrowDiameter = 0.02, displaceZ = -0.15},
     {y, p, r} = yprFromRotMat[rotMatSO3];
     Show[{
       Graphics3D[{
         Text[myFont[Blue][
           "t = " <> ToString[NumberForm[t, {10, 2}]]], {-1, -1, placeZ}],
         placeZ += displaceZ;
         Text[myFont[Blue][
           "yaw = " <> ToString[NumberForm[y / °, {10, 4}]] <> "°"],
          {-.999, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "pitch = " <> ToString[NumberForm[p / °, {10, 4}]] <> "°"],
          {-.999, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "roll = " <> ToString[NumberForm[r / °, {10, 4}]] <> "°"],
          {-.999, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "|L_s| = " <> ToString[NumberForm[Sqrt[Ls.Ls], {10, 4}]]],
          {-.975, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "|L_b| = " <> ToString[NumberForm[Sqrt[Lb.Lb], {10, 4}]]],
          {-.975, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "ω_b^TI_bω_b/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Ib.ωb], {10, 4}]]],
          {-.95, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "ω_b^TL_b/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Lb], {10, 4}]]],
          {-.95, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "ω_s^TL_s/2 = " <> ToString[NumberForm[Sqrt[1/2 ωs.Ls], {10, 4}]]],
          {-.95, -1, placeZ}],

         Magenta, Arrow[Tube[{o, Ls}, arrowDiameter]],
         Text[myFont[Black, 12]["Inertial-Frame Ang Mom"],
          {-1.9, +0.2, 1}, Background → Magenta],

         Red, Arrow[Tube[{o, Lb}, arrowDiameter]],
         Text[myFont[Black, 12]["Body-Frame Ang Mom"],
          {-1.7, -0.1, 1}, Background → Red],

         Cyan, Arrow[Tube[{o, ωs / 10}, arrowDiameter * 1.10]],
         Text[myFont[Black, 12]["Inertial-Frame Ang Vel"],
          {-1.4, -0.4, 1}, Background → Cyan],

         Blue, Arrow[Tube[{o, ωb / 10}, arrowDiameter * 1.10]],
         Text[myFont[White, 12]["Body-Frame Ang Vel"],
          {-1.3, -0.7, 1}, Background → Blue],

         White, GeometricTransformation[
          apparatus["graphics primitives"],
          rotMatSO3]}
       ]},
      Axes → True,
      PlotRange → ConstantArray[plotRg, 3],
      AxesLabel → myFont[Red] /@ {"X", "Y", "Z"},
      ImageSize → Large]]]);

ClearAll[initialConditionsSO3ForcedRotationalMotion];
initialConditionsSO3ForcedRotationalMotion[ωb_, rotMatSO3_, Ib_, Ibi_] :=
  With[{rq = qFromRotMat[rotMatSO3], Lb = Ib.ωb},
   {ωb,
    rotMatSO3,
    (* ωs *)rv[rq, ωb],
    Lb,
    (* Ls *)rv[rq, Lb]}];

ClearAll[oneStepSO3ForcedRotationalMotion];
oneStepSO3ForcedRotationalMotion[ωb_, rotMatSO3_, Ib_, Ibi_, h_, τs_] :=
  With[{rq = qFromRotMat[rotMatSO3]},
   With[{
     ωbNew = ωbn[ωb, Ib, Ibi, h, τs, rq], (* calls rk4 *)
     rqNew = rqb2sn[rq, ωb, h]}, (* calls rk4 *)
     With[{LbNew = Ib.ωbNew},
      {ωbNew,
       rotMatFromQ[rqNew],
       (* ωs *)rv[rqNew, ωbNew],
       LbNew,
       (* Ls *)rv[rqNew, LbNew]}]]];

ClearAll[runSimSO3ForcedRotationalMotion];
runSimSO3ForcedRotationalMotion[
   apparatus_,
   ωbIn_ : {6., .01, 0},
   rqIn_ : rq[π / 4.0, {0, 1., 0}],
   fs_ : {0, 0, 0},
   τs_ : {0, 0, 0}] :=
  With[{ibibi = apparatus["moment of inertia"]},
   With[{Ib = ibibi⟦1⟧, Ibi = ibibi⟦2⟧},
    DynamicModule[
     {t = 0, h = 0.01, ωb = ωbIn, ωs, Lb, Ls, rotMat = rotMatFromQ[rqIn]},
     Dynamic[t += dt;
      {ωb, rotMat, ωs, Lb, Ls} =
       (* calls rk4 *)
       oneStepSO3ForcedRotationalMotion[ωb, rotMat, Ib, Ibi, h, τs];
      showSO3Apparatus[t, ωb, Lb, ωs, Ls, Ib, rotMat, apparatus]]]]];
```

Demonstration of step-by-step RK4 integration on SO(3).

In[528]:=

```
With[{apparatus = dzhanybekhov, h = 0.01,
  ωbIn = {6., .01, 0.0}, rqIn = rq[π / 4.0, {0, 1, 0}], τs = {0, 0, 0}},
 With[{ibibi = apparatus["moment of inertia"]},
  With[{Ib = ibibi⟦1⟧, Ibi = ibibi⟦2⟧},
   DynamicModule[{ωb = ωbIn, rotMatSO3 = rotMatFromQ[rqIn], ωs, Lb, Ls, t = 0},
    With[{
      init = Function[(* of no arguments *)
        {ωb, rotMatSO3, ωs, Lb, Ls} =
         initialConditionsSO3ForcedRotationalMotion[
          ωbIn, rotMatFromQ[rqIn], Ib, Ibi];
        t = 0;
        showSO3Apparatus[t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, apparatus]],
      step = Function[(* of no arguments *)
        {ωb, rotMatSO3, ωs, Lb, Ls} =
         (* calls rk4 *)
         oneStepSO3ForcedRotationalMotion[ωb, rotMatSO3, Ib, Ibi, h, τs];
        t += h;
        showSO3Apparatus[t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, apparatus]]},
     DynamicModule[{dpy = init[]},
      Manipulate[dpy,
       Row[{Button[" STEP ! ", dpy = step[]],
         Button[" RESET ! ", dpy = init[]]}]]]]]]]]
```

Out[528]=



Root Finding

To step from T^{k-1} and T^k to T^{k+1}, find the root T^{k+1} of $D_1 L_d(T^k, T^{k+1}) + D_2 L_d(T^{k-1}, T^k) + F^k = 0$, where $F^k \in \mathfrak{se}^*(3)$ is the integral of the virtual work of the external force over time interval h.

Start with a certain orientation and angular velocity:

In[529]:=

```
o3 = {0, 0, 0};
```

In[530]:=

```
(T0$ = TSE3[initQuat$, o3]) // SE3Form
```

Out[530]//DisplayForm=

```
⎛  0.707107 0. 0.707107│0⎞
⎜     0.     1.    0.  │0⎟
⎜ -0.707107 0. 0.707107│0⎟
⎜──────────────────────┼─⎟
⎝      0      0     0  │1⎠
```

In[531]:=

```
ClearAll[stepDzhanyRK4];
With[{apparatus = dzhanybekhov},
  With[{ibibi = apparatus["moment of inertia"]},
   With[{Ib = ibibi⟦1⟧, Ibi = ibibi⟦2⟧},
    stepDzhanyRK4[Tk_, ωbIn_, h_, τs_ : o3] :=
     Module[{ωb = ωbIn, rotMatSO3 = pickSO3[Tk], ωs, Lb, Ls},
      {ωb, rotMatSO3, ωs, Lb, Ls} =
       oneStepSO3ForcedRotationalMotion[ωb, rotMatSO3, Ib, Ibi, h, τs];
      <|"TSE3" → TSE3[rotMatSO3, o3], "ωb" → ωb, "Lb" → Lb, "ωs" → ωs, "Ls" → Ls|>
     ]]]];
```

In[533]:=

```
ClearAll[adHocStep];
adHocStep[T_, w_, h_] :=
  (step$ = stepDzhanyRK4[T, w, h];
   wb$ = step$["ωb"];
   Tkp$ = step$["TSE3"]);
adHocStep[T0$, initωb$, 0.01];
(T1$ = Tkp$) // SE3Form
```

Out[536]//DisplayForm=

```
⎛  0.707109   0.0423303  0.705836 │0⎞
⎜ 0.00009994  0.998201  -0.059964 │0⎟
⎜ -0.707105   0.0424716  0.705832 │0⎟
⎜─────────────────────────────────┼─⎟
⎝     0           0          0    │1⎠
```

In[537]:=

```
adHocStep[T1$, wb$, 0.01];
(T2$ = Tkp$) // SE3Form
```

Out[538]//DisplayForm=

```
⎛   0.707119    0.084508   0.702026 │0⎞
⎜ 0.000199122  0.992809  -0.119712  │0⎟
⎜  -0.707094   0.0847906  0.702017  │0⎟
⎜───────────────────────────────────┼─⎟
⎝      0           0          0     │1⎠
```

DREPE should have magnitude close to zero.

In[539]:=

```
DREPE[T0$, T1$, T2$, Gdzhany6R6, 0 &, 0, 0.01] // Norm
```

Out[539]=

```
7.99789 × 10^-9
```

`T2$` is very nearly a root of DREPE. Is it close enough? Hard to say, yet, but we study the numerics below and we will refine the root with ***bandit search*** when needed (https://en.wikipedia.org/wiki/Multi-armed_bandit).

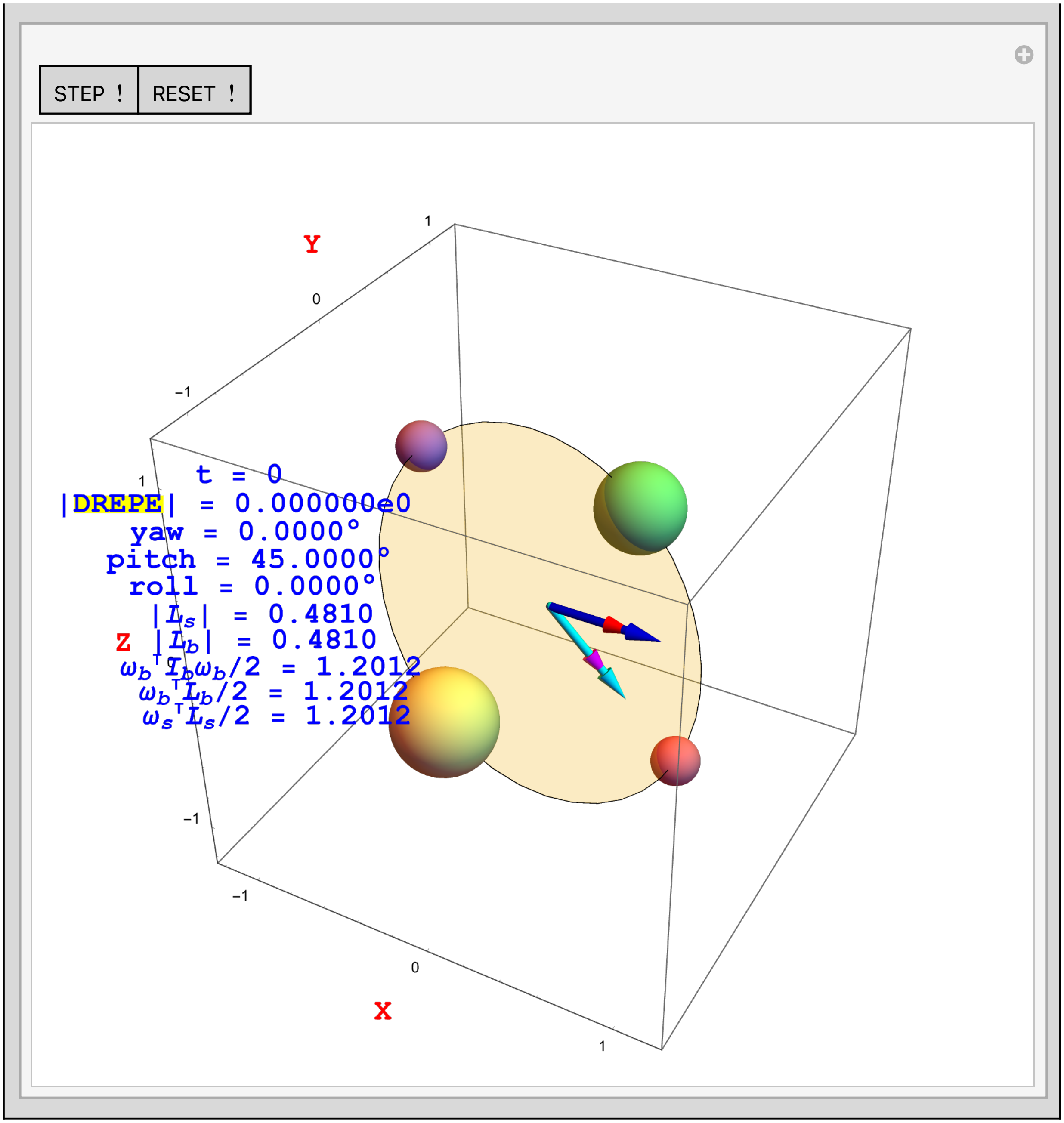
Adapt the step-by-step demonstration above to DREPE. The first step produces a macroscopic value of DREPE, that is, a failure to find a root. This failure is due only to the fact that DREPE requires two configuration estimates to bootstrap — we've seen this before in DELE. From the second step onward, the roots proposed by RK4 produce only microscopic values of DREPE, showing that the proposed roots are good for *Dzhanybekhov*. **We suspect that they will not be good for 4DISP.**

In[540]:=

```
ClearAll[showSO3ApparatusWithDREPE];
showSO3ApparatusWithDREPE[
   t_, ωb_, Lb_, ωs_, Ls_, Ib_, rotMatSO3_, DREPENorm_, apparatus_] :=
  (Module[{placeZ = 1, y, p, r},
    With[{arrowDiameter = 0.02, displaceZ = -0.15},
     {y, p, r} = yprFromRotMat[rotMatSO3];
     Show[{
       Graphics3D[{
         Text[myFont[Blue][
           "t = " <> ToString[NumberForm[t, {10, 2}]]], {-1, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "|\!\(\*StyleBox[\"DREPE\",Background->RGBColor[1,
               1, 0]]\)| = " <>
            ToString[NumberForm[DREPENorm, {10, 6}, NumberFormat →
                (Row[{#1, "e", If[#3 === "", "0 ", #3]}] &)]]],
          {-.999, -1, placeZ}],

         placeZ += displaceZ;
         Text[myFont[Blue][
           "yaw = " <> ToString[NumberForm[y / °, {10, 4}]] <> "°"],
          {-.999, -1, placeZ}],
placeZ += displaceZ;
Text[myFont[Blue][
   "pitch = " <> ToString[NumberForm[p / °, {10, 4}]] <> "°"],
  {-.999, -1, placeZ}],

placeZ += displaceZ;
Text[myFont[Blue][
   "roll = " <> ToString[NumberForm[r / °, {10, 4}]] <> "°"],
  {-.999, -1, placeZ}],

placeZ += displaceZ;
Text[myFont[Blue][
   "|Ls| = " <> ToString[NumberForm[Sqrt[Ls.Ls], {10, 4}]]],
  {-.975, -1, placeZ}],

placeZ += displaceZ;
Text[myFont[Blue][
   "|Lb| = " <> ToString[NumberForm[Sqrt[Lb.Lb], {10, 4}]]],
  {-.975, -1, placeZ}],

placeZ += displaceZ;
Text[myFont[Blue][
   "ωb^T Ib ωb/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Ib.ωb], {10, 4}]]],
  {-.95, -1, placeZ}],

placeZ += displaceZ;
Text[myFont[Blue][
   "ωb^T Lb/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Lb], {10, 4}]]],
  {-.95, -1, placeZ}],

placeZ += displaceZ;
Text[myFont[Blue][
   "ωs^T Ls/2 = " <> ToString[NumberForm[Sqrt[1/2 ωs.Ls], {10, 4}]]],
  {-.95, -1, placeZ}],

Magenta, Arrow[Tube[{o, Ls}, arrowDiameter]],
Text[myFont[Black, 12]["Inertial-Frame Ang Mom"],
  {-1.9, +0.2, 1}, Background → Magenta],
         Red, Arrow[Tube[{o, Lb}, arrowDiameter]],
         Text[myFont[Black, 12]["Body-Frame Ang Mom"],
          {-1.7, -0.1, 1}, Background → Red],

         Cyan, Arrow[Tube[{o, ωs / 10}, arrowDiameter ∗ 1.10]],
         Text[myFont[Black, 12]["Inertial-Frame Ang Vel"],
          {-1.4, -0.4, 1}, Background → Cyan],

         Blue, Arrow[Tube[{o, ωb / 10}, arrowDiameter ∗ 1.10]],
         Text[myFont[White, 12]["Body-Frame Ang Vel"],
          {-1.3, -0.7, 1}, Background → Blue],

         White, GeometricTransformation[
          apparatus["graphics primitives"],
          rotMatSO3]}
       ]},
      Axes → True,
      PlotRange → ConstantArray[plotRg, 3],
      AxesLabel → myFont[Red] /@ {"X", "Y", "Z"},
      ImageSize → Large]]]);

With[{apparatus = dzhanybekhov, h = 0.01,
  ωbIn = {6., .01, 0.0}, rqIn = rq[π / 4.0, {0, 1, 0}], τs = {0, 0, 0}},
 With[{ibibi = apparatus["moment of inertia"]},
  With[{Ib = ibibi〚1〛, Ibi = ibibi〚2〛},
   DynamicModule[{ωb = ωbIn, rotMatSO3 = rotMatFromQ[rqIn],
     ωs, Lb, Ls, t = 0, Tkm, Tk, Tkp, DREPENorm},
    With[{
      init = Function[(* of no arguments *)
        Tkm = Tk = TSE3[rqIn, o3];
        {ωb, rotMatSO3, ωs, Lb, Ls} =
         initialConditionsSO3ForcedRotationalMotion[
          ωbIn, rotMatFromQ[rqIn], Ib, Ibi]; t = 0;
        Tkp = TSE3[rotMatSO3, o3];
        DREPENorm = Norm[DREPE[Tkm, Tk, Tkp, Gdzhany6R6, 0 &, 0, h]];
        showSO3ApparatusWithDREPE[
         t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, DREPENorm, apparatus]],
      step = Function[(* of no arguments *)
        Tkm = Tk; Tk = Tkp;
        {ωb, rotMatSO3, ωs, Lb, Ls} =
         oneStepSO3ForcedRotationalMotion[ωb, rotMatSO3, Ib, Ibi, h, τs];
        Tkp = TSE3[rotMatSO3, o3]; (* still a quat in the sim *)
```

```
         DREPENorm = Norm[DREPE[Tkm, Tk, Tkp, Gdzhany6R6, 0 &, 0, h]];
         t += h;
         showSO3ApparatusWithDREPE[
          t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, DREPENorm, apparatus]]},
      DynamicModule[{dpy = init[]},
       Manipulate[dpy,
        Row[{Button[" STEP ! ", dpy = step[]],
          Button[" RESET ! ", dpy = init[]]}]]]]]]]]
```

Out[542]=



t = 0

|DREPE| = 0.000000e0

yaw = 0.0000°

pitch = 45.0000°

roll = 0.0000°

$|L_s|$ = 0.4810

$|L_b|$ = 0.4810

$\omega_b{}^T I_b \omega_b/2$ = 1.2012

$\omega_b{}^T L_b/2$ = 1.2012

$\omega_s{}^T L_s/2$ = 1.2012

Adapt the continuous demonstration of Dzhanybekhov to CGDVIE3 and DREPE. This only tracks the microscopic values of DREPE; it does not update the roots, proposed by RK4. **Note the time incre-**

ment, *h* = 0.03 is three times larger (better) than we had with RK4. 0.03 is much too large to produce decent roots of DREPE for 4DISP; even *h* = 0.01 is too large for 4DISP.

In[543]:=

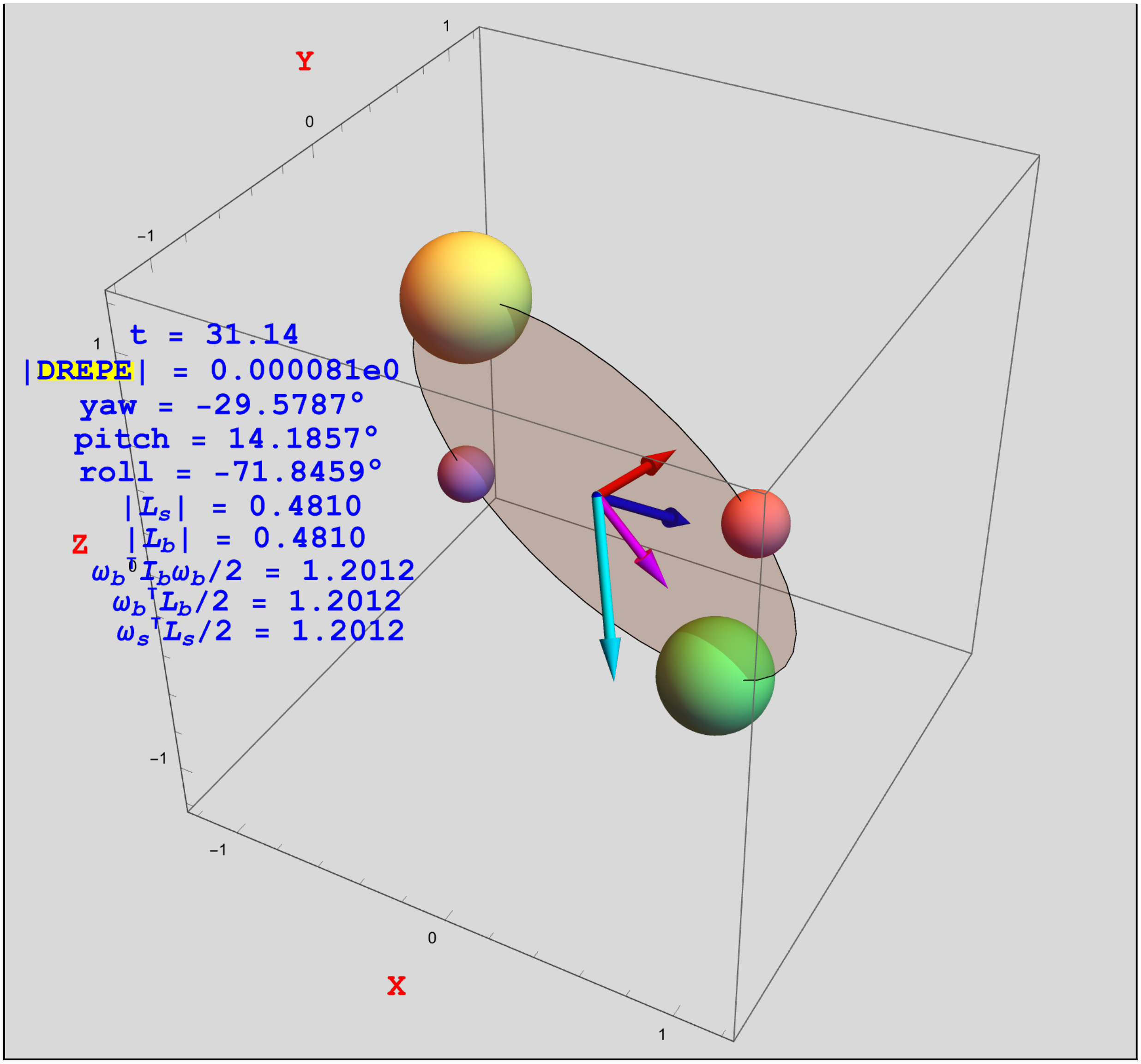
```
ClearAll[runSimForcedRotationalMotionWithDREPE];
runSimForcedRotationalMotionWithDREPE[
   apparatus_,
   h_,
   ωbIn_ : {6., .01, 0},
   rqIn_ : rq[π / 4.0, {0, 1., 0}],
   fs_ : {0, 0, 0},
   τs_ : {0, 0, 0}] :=
  With[{ibibi = apparatus["moment of inertia"]},
   With[{Ib = ibibi⟦1⟧, Ibi = ibibi⟦2⟧},
    DynamicModule[
     {t = 0, ωb = ωbIn, ωs, Lb, Ls,
      rotMatSO3 = rotMatFromQ[rqIn], Tkm, Tk, Tkp, DREPENorm},
     With[{
       init = Function[(* of no arguments *)
         Tkm = Tk = TSE3[rqIn, o3];
         {ωb, rotMatSO3, ωs, Lb, Ls} =
          initialConditionsSO3ForcedRotationalMotion[
           ωbIn, rotMatFromQ[rqIn], Ib, Ibi]; t = 0;
         Tkp = TSE3[rotMatSO3, o3];
         DREPENorm = Norm[DREPE[Tkm, Tk, Tkp, Gdzhany6R6, 0 &, 0, h]];
         showSO3ApparatusWithDREPE[
          t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, DREPENorm, apparatus]],
       step = Function[(* of no arguments *)
         Tkm = Tk; Tk = Tkp;
         {ωb, rotMatSO3, ωs, Lb, Ls} =
          oneStepSO3ForcedRotationalMotion[ωb, rotMatSO3, Ib, Ibi, h, τs];
         Tkp = TSE3[rotMatSO3, o3];
         (* still a quat in the sim *)
         DREPENorm = Norm[DREPE[Tkm, Tk, Tkp, Gdzhany6R6, 0 &, 0, h]];
         t += h;
         showSO3ApparatusWithDREPE[
          t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, DREPENorm, apparatus]]},
      init[];
      Dynamic[t += h;
       step[]]]]]];
```

This also loses energy and angular momentum, but more slowly than does TQDL & RK4 and at a time step 3 times more coarse — 0.3 instead of the .01 we needed with TDQL and RK4. **This is a significant improvement.** Note that the quantity |DREPE| is ideally zero. We see that it fluctuates around some small values, suggesting overall good performance.

In[545]:=

```
runSimForcedRotationalMotionWithDREPE[dzhanybekhov, 0.03]
```

Out[545]=

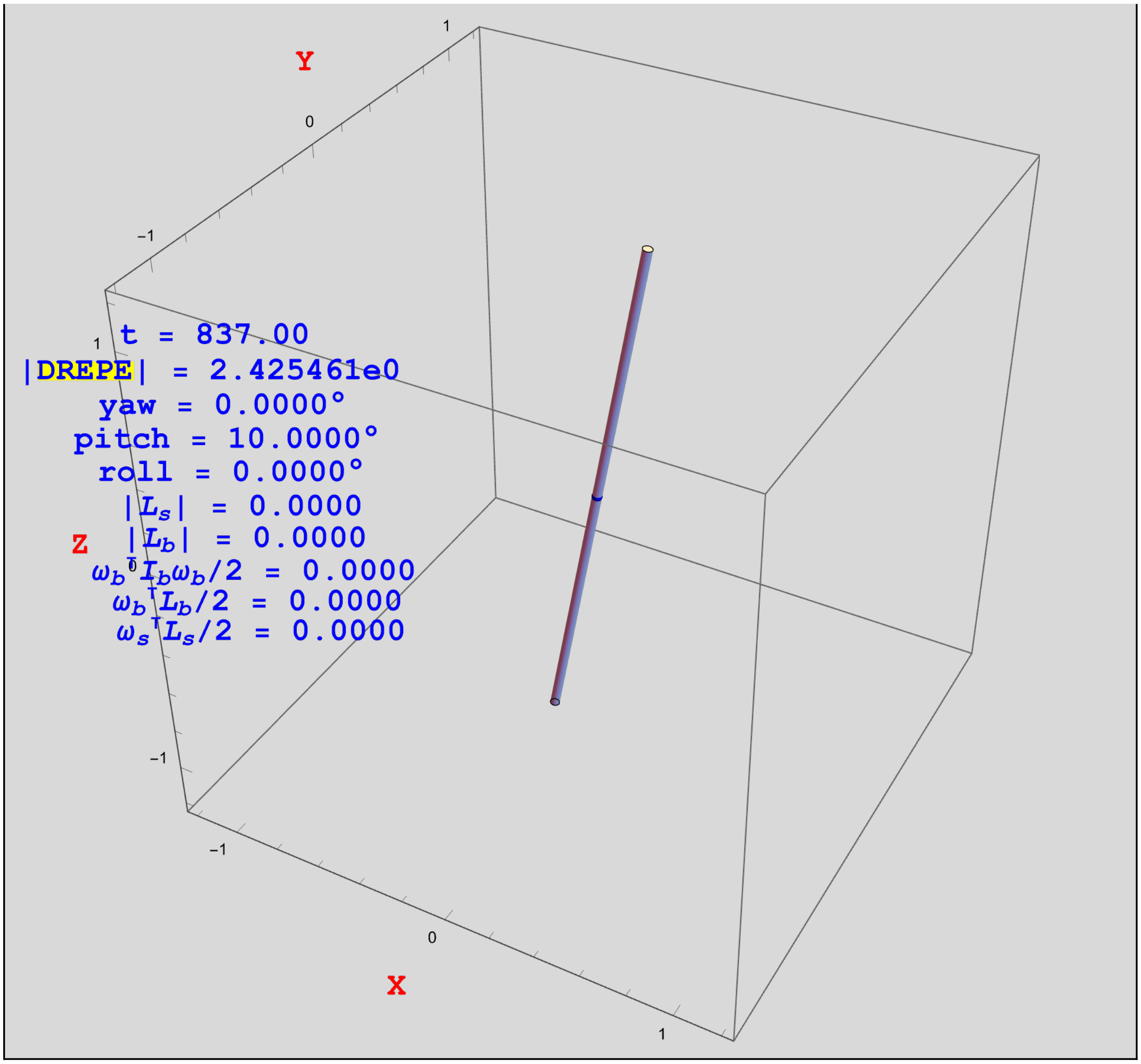

1
Y
0
−1
1
t = 31.14
|DREPE| = 0.000081e0
yaw = -29.5787°
pitch = 14.1857°
roll = -71.8459°
$|L_s|$ = 0.4810
Z
$|L_b|$ = 0.4810
0
$\omega_b{}^{\mathsf{T}} I_b \omega_b/2$ = 1.2012
$\omega_b{}^{\mathsf{T}} L_b/2$ = 1.2012
$\omega_s{}^{\mathsf{T}} L_s/2$ = 1.2012
−1
−1
0
X
1

We can’t do 4DISP yet because we don’t have a force model.

In[546]:=

```
ClearAll[runSimForcedRotationalMotionWithDREPErig];
runSimForcedRotationalMotionWithDREPErig[
   apparatus_,
   h_,
   ωbIn_ : {6., .01, 0},
   rqIn_ : rq[π / 4.0, {0, 1., 0}],
   fs_ : {0, 0, 0},
   τs_ : {0, 0, 0}] :=
  With[{ibibi = apparatus["moment of inertia"]},
   With[{Ib = ibibi〚1〛, Ibi = ibibi〚2〛},
    DynamicModule[
     {t = 0, ωb = ωbIn, ωs, Lb, Ls, rotMatSO3 = rotMatFromQ[rqIn],
      Tkm, Tk, Tkp, DREPENorm, qk, csk, τsk, Fk},
     With[{
        init = Function[(* of no arguments *)
          Tkm = Tk = TSE3[rqIn, o3];
          {ωb, rotMatSO3, ωs, Lb, Ls} =
           initialConditionsSO3ForcedRotationalMotion[
            ωbIn, rotMatFromQ[rqIn], Ib, Ibi]; t = 0;
          Tkp = TSE3[rotMatSO3, o3];
          qk = qFromRotMat[rotMatSO3];
          csk = rv[qk, rig["cb"]];
          τsk = (rig["mass"] Abs[g] e3) × csk;
          Fk = ConstantArray[-1., 6];
          DREPENorm = Norm[DREPE[Tkm, Tk, Tkp, Gdzhany6R6, PrigR, Fk, h]];
          showSO3ApparatusWithDREPE[
           t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, DREPENorm, apparatus]],
        step = Function[(* of no arguments *)
          Tkm = Tk; Tk = Tkp;
          {ωb, rotMatSO3, ωs, Lb, Ls} =
           oneStepSO3ForcedRotationalMotion[ωb, rotMatSO3, Ib, Ibi, h, τs];
          Tkp = TSE3[rotMatSO3, o3];
          (* still a quat in the sim *)
          qk = qFromRotMat[rotMatSO3];
          csk = rv[qk, rig["cb"]];
          τsk = (rig["mass"] Abs[g] e3) × csk;
          Fk = ConstantArray[-1., 6];
          DREPENorm = Norm[DREPE[Tkm, Tk, Tkp, Gdzhany6R6, PrigR, Fk, h]];
          t += h;
          showSO3ApparatusWithDREPE[
           t, ωb, Lb, ωs, Ls, Ib, rotMatSO3, DREPENorm, apparatus]]},
      init[];
      Dynamic[t += h;
       step[]]]]]];
runSimForcedRotationalMotionWithDREPErig[rig, 0.03, o, Q[0, 10 °, 0 ∗ -0.5 °], o]
```

Out[548]=


1
Y
0
−1
t = 837.00
|DREPE| = 2.425461e0
yaw = 0.0000°
pitch = 10.0000°
roll = 0.0000°
$|L_s|$ = 0.0000
Z
$|L_b|$ = 0.0000
$\omega_b{}^{T} I_b \omega_b/2$ = 0.0000
$\omega_b{}^{T} L_b/2$ = 0.0000
$\omega_s{}^{T} L_s/2$ = 0.0000
1
0
−1
−1
0
X
1