Discrete Variational Spherical Pendulum: Free Motion

Brian Beckman
January 2025

Abstract

We compare three numerical solutions for free motion of an inverted spherical pendulum. Our contribution is to find that only the discrete variational solution exhibits long-term, secular energy conservation at acceptably large time steps. By contrast, numerical integration of the continuous equations of motion exhibits secular energy growth, even at unacceptably small time steps.

The discrete variational approach begins by discretizing dynamical variables and the Lagrangian, replacing continuous functions of time with sequences of discrete quantities. We call this ***early discretization***. Appropriate derivatives of the discrete Lagrangian produce algebraic recurrences that we solve by root-finding. The standard approach delays discretizing continuous functions as late in the process as possible, typically inside opaque solvers for differential equations. We call this ***late discretization***. We find two late-discretized solutions with undesirable secular energy growth. We speculate that all late-discretized solutions will have incorrect secular trends of conserved quantities.

Details

When dropped from rest at co-latitude, say, 5 degrees, the expected behavior is for the pendulum to swing through the nadir, rise to 355 degrees in the back hemisphere, then fall back around again and repeat. This large-swing motion is convenient for visually checking energy conservation. If the pendulum swings up too far in the back, the solution is gaining energy through secular numerical noise.

Runge-Kutta integration of quaternions at a 1-ms time step immediately gains enough energy to swing all the way through zenith on the first drop. Only by going to 100-μs time step is there even short-term energy conservation. At 100-μs time step, the solution is far slower than real-time, unacceptable for animation.

In pitch/cone-roll coordinates, and with *Mathematica*'s automatic (and opaque) choices for time step, secular energy conservation is much better, plus we get animation speed. However, there is still secular energy growth. The solution will eventually require rebooting.

By contrast, the early-discretized solution—the discrete variational solution—has no discernable secular energy growth, even at 1-ms time step. There is *periodic* energy noise in the fourth significant

figure, but this is not visible in plots or animations. The periodic noise grows in amplitude, so the solution will eventually require rebooting, but for a different reason. Solution time is much longer than we would like. However, the attractiveness of secular energy conservation warrants further optimization and study of this method.

This notebook is part of a series on discrete variational dynamics. Previous notebooks illustrated harmonic oscillators and the spinning symmetrical top.

Introduction

Our spherical pendulum is a massive baton of uniform density resting on a massless, sliding, point-like base. The base slides on a ghostly floor: the baton falls through the floor in either direction. When dropped from near upright, the baton falls under its own weight, sliding the base back and then forth. The baton then swings through nadir and back up again via its inertia, sliding the base forth and then back.

In this notebook, we consider only free motion. Later notebooks will address controlling the baton by applying forces to the base.

The spherical pendulum is surprisingly difficult given that the circular pendulum, with one angle and one position coordinate, is an undergraduate exercise with easily solvable optimal control.[15] The spherical pendulum is the subject of at least one doctoral thesis.[7] An internet search for "control inverted spherical pendulum" produces many references.

For the spherical problem:

- Standard spherical coordinates are ruled out because they are singular at the North pole, exactly where we want to balance and control the baton. Control models diverge in these coordinates.
- Quaternions have no singularity, but with fourth-order Runge-Kutta (RK4) or Runge-Kutta-Fehlberg (RK45), solutions fail spectacularly, even with 1-ms time step. In this notebook, the integrators are conclusively shown at fault.
- Pitch/cone-roll coordinates, developed in this notebook, avoid singularity at the North pole (the control set-point). Continuous equations in these coordinates behave better under late discretization, i.e., standard numerical methods. Numerical solutions in these coordinates have adequate but imperfect secular energy conservation. While small, secular energy growth is a signal that late discretization is fundamentally flawed. Also, while *Mathematica*'s `NDSolve` works well, it does not export out of *Mathematica*, so we can't use it for deployed solutions.
- Finally, we consider a discrete Lagrangian integrator in pitch/cone-roll coordinates. This has great long-term energy conservation. Its periodic energy noise—in the fourth decimal place—is much larger than the secular-growth noise of the continuous case—in the sixth or seventh decimal place—but still not noticeable visually.

Outline

1. **late discretization**: Integrate continuous Euler-Lagrange equations of motion in quaternions via RK4 and RK45.
2. Demonstrate fast energy divergence.
3. **late discretization**: Integrate continuous Euler-Lagrange equations in pitch/cone-roll coordinates.
4. Demonstrate slow energy divergence.
5. **early discretization**: Develop discrete Euler-Lagrange equations in pitch/cone-roll coordinates. Solve resulting recurrences via root-finding.
6. Demonstrate secular energy conservation despite periodic noise.

References

This is a list of general references on discrete variational dynamics and the background mathematics. Not all these references are cited directly in this notebook.

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, 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).
27. F. C. Park, J. E. Bobrow, S. R. Ploen, *A Lie Group Formulation of Robot Dynamics.*
28. George W. Hart, *Multidimensional Analysis, https://a.co/d/8c5RqOX*
29. Evandro Bernardes, Stephane Viollet, *Quaternion to Euler angles conversion: A direct, general and computationally efficient method,* PLoS ONE 17(11) e0276302
30. Linear Quadratic Regulator, from *Underactuated Robotics*, https://underactuated.mit.edu/lqr.html

Quaternions, RK4, RK45

Quaternions do not have the gimbal lock inherent to Euler angles, which are singular at the North pole. With fourth-order Runge-Kutta (RK4) or Runge-Kutta-Fehlberg integration (RK45), we get plausible results on a unit test of rigid-body free rotation, spontaneously exhibiting the intermediate-axis (Dzhanibekov) effect.[8] Thus, we trust the integration scheme.

However, RK4 and RK45 fail dramatically on the spherical pendulum, even with a 1-ms time step. They require 100-μs granularity to conserve energy and momentum, even in the short run. Such failure is

not unexpected, but the drama is surprising—despite doing a good job on rigid-body rotation, Runge-Kutta integration is not even close for the spherical pendulum.

Tiny Quaternionic Dynamics Library (TQDL)

See Reference [1].

Primitives

Most items have two names: one long and one short. The long names bulk up code, so we use the short ones most of the time. The long names explain and remind us of the short ones.

In[74]:=
```
<< 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];
```

A *versor* is a *rotation quaternion*, representing an angular twist around a unit 3-vector axis of rotation. `rq` is short for *versor*.

In[76]:=
```
versor := rq;
```

`rq` can take a zero vector as input. `Normalize` does not fail on zero input. Results are not valid versors, though this library improperly allows them. It does not seem to be a problem in practice.

Ignore the error caret in the following definition because `Sequence` expands correctly to three arguments.

In[77]:=
```
rq[θ_?NumberQ, v_List] :=
  Quaternion[Cos[θ/2], Sequence @@ (Sin[θ/2] Normalize[v])];
```

Overload of `rq` for four numerical arguments:

In[78]:=
```
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.

In[79]:=
```
rq0 = rq[0, {1, 0, 0}];
```

In[80]:=
```
random3Vector := ranv;
ranv[] := RandomReal[{-1, 1}, 3];
```

In[82]:=
```
randomAngleRad := ranθ;
ranθ[] := RandomReal[{0, 2 π}];
```

In[84]:=
```
randomVersor := ranrq;
ranrq[] := rq[RandomReal[2 π], ranv[]];
```

In[86]:=
```
versorFromTwistVector := rqw;
rqw[w_List] := rq[Norm[w], w];
```

Every quaternion, not just a versor, has a *real part*, a 3-vector:

In[88]:=
```
realPart := vq;
vrq := vq;
vq[q_Quaternion] := List @@ q[[2 ;; 4]];
```

In[91]:=
```
twistVectorFromVersor := wrq;
wrq[rq_Quaternion] := 2 ArcCos[rq[[1]]] Normalize @ vq @ rq;
```

In[93]:=
```
twistAngleFromVersor := θrq;
θrq[rq_Quaternion] := 2 ArcCos[rq[[1]]];
```

In[95]:=
```
twistAngleAndRealPartFromVersor := θvrq;
θvrq[rq_Quaternion] := {θrq[rq], vrq[rq]};
```

A *pure quaternion* has no imaginary part. It represents an arbitrary 3-vector in the space of quaternions.

In[97]:=
```
pureQuaternion := qv;
```

... another incorrect error caret because `Sequence` expands to three arguments ...

In[98]:=
```
qv[v_List] := Quaternion[0, Sequence @@ v];
```

Rotate a vector-as-pure-quaternion by acting on it with versors.

In[99]:=
```
rotatedVector := rv;
rv[rq_Quaternion, v_List] := vq[rq ** qv[v] ** rq*];
```

The quaternion-conjugate action rotates the frame of a 3-vector rather than the 3-vector itself.

In[101]:=
```
vectorInRotatedFrame := rf;
rf[rq_Quaternion, v_List] := vq[rq* ** qv[v] ** rq];
```

In[103]:=
```
normalize[q_Quaternion] :=
 With[{n = Chop[Abs[q]]},
  If[n == 0.0, Quaternion[0, 0, 0, 0], (* ? should be "rq0" ? *)
   q / Abs[q]]]
```

Time Derivatives

In[104]:=
```
ClearAll[drqb2sdt, dωbdt, rk4, ωbn, dVersorDt,
  dBodyAngVelDt, stepVersorFromBodyAngVel, stepAngVelBodyFrame];
```

The usual definition of dq/dt in the inertial space frame is $\frac{1}{2}\,\omega_s\,q$. My angular velocity, ω_b, is in the body frame. But $\omega_s = q\,\omega_b\,q^*$ is in the inertial space frame, so $dq/dt = \frac{1}{2}\,\omega_s\,q = \frac{1}{2}\,q\,\omega_b\,q^*\,q = \frac{1}{2}\,q\omega_b$ because $q^*\,q = 1$ for a versor, by definition.

In[105]:=
```
dVersorDt := drqb2sdt; (* ignore t *)
drqb2sdt[t_, qb2s_Quaternion, ωb_List] := (qb2s / 2) ** qv[ωb];
```

In[107]:=
```
dBodyAngVelDt := dωbdt;(* ignore t*)
dωbdt[t_, ωb_List, mi_List, mii_List, τs_, rq_] :=
  mii.(rf[rq, τs] - ωb × (mi.ωb));
```

Quaternion From Euler Angles ψ, θ, ϕ

Mnemonics: θ looks like pitch, ϕ looks like roll, ψ looks like 'y' in 'yaw.'

Via page 206 of Kuipers.[1]

In[109]:=
```
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[112]:=
```
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 $\pi/2$. Numerical roblems occur near the endpoints of those ranges, especially for pitch.

Euler Angles ψ, θ, ϕ from Quaternions

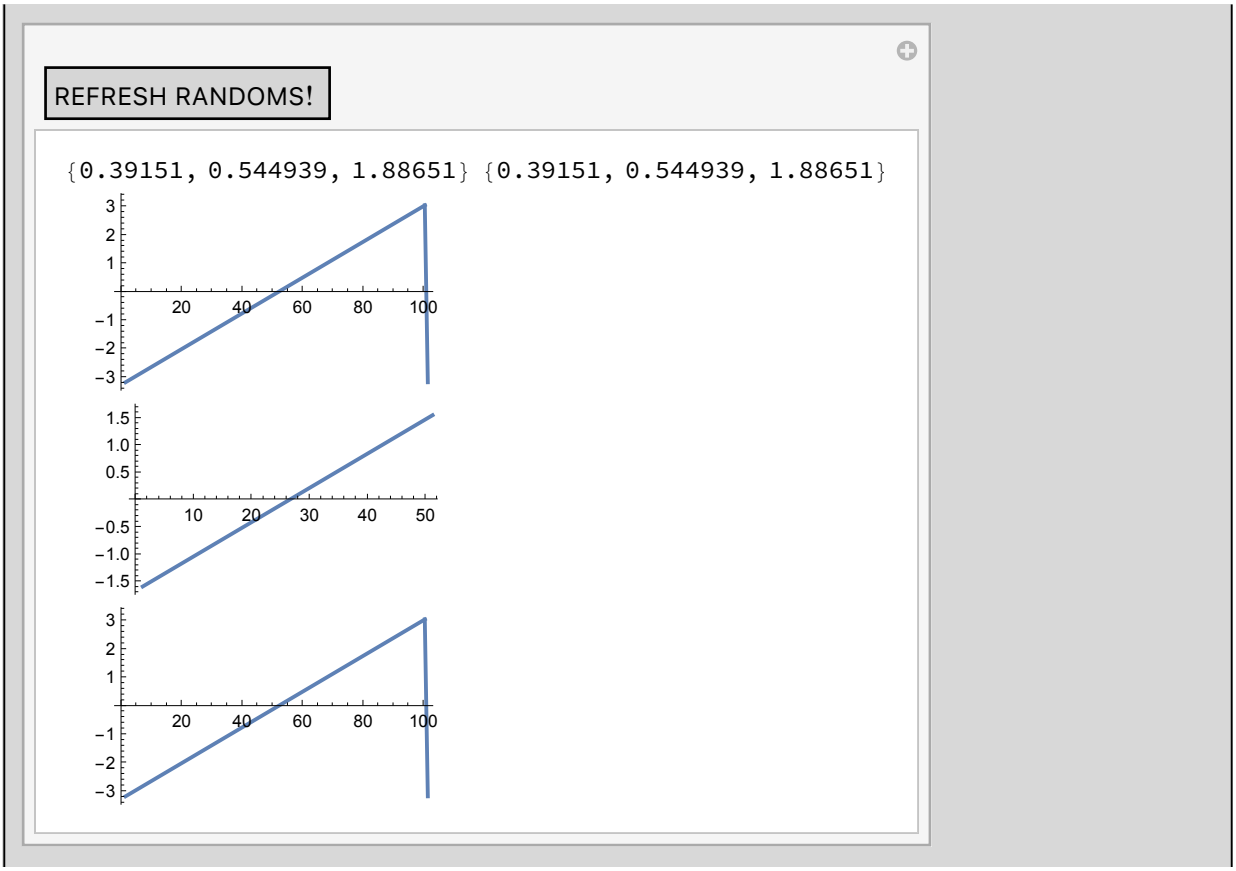
Via Reference 29, tailored (hacked) to our use-cases, with unit test.

In[113]:=

```
ClearAll[eulerAnglesFromQ, ψθϕFromQ];
eulerAnglesFromQ[Quaternion[w_, x_, y_, z_], {i_, j_, k_} /; i ≠ j && j ≠ k] :=
  Module[{a, b, c, d, ε, notProper = True, θ1, θ2, θ3, θPlus, θMinus, ψ, θ, ϕ},
   If[i == k, notProper = False; k = 6 - i - j];
   ε = (i - j) (j - k) (k - i) / 2;
   If[notProper,
    a = w - y;
    b = x + z ε;
    c = y + w;
    d = z ε - x,
    (*else*)
    a = w;
    b = x;
    c = y;
    d = z ε];
   θ2 = ArcCos[2 (a^2 + b^2)/(a^2 + b^2 + c^2 + d^2) - 1];
   θPlus = ArcTan[b, a];
   θMinus = ArcTan[d, c];
   If[θ2 == 0,
    θ1 = 0,
    θ3 = 2 θMinus - θ1;
    (*else*)
    If[θ2 == N[π] / 2,
     θ1 = 0;
     θ3 = 2 θMinus + θ1,
     (*else*)
     θ1 = θPlus - θMinus;
     θ3 = θPlus + θMinus]
   ];
   ψ = Mod[-If[notProper, ε θ3, θ3], 2 π] - π;
   θ = If[notProper, θ2 - π / 2, θ2];
   ϕ = Mod[-θ1 - π, 2 π] - π;
   {ψ, θ, ϕ}];
ψθϕFromQ[q : Quaternion[w_, x_, y_, z_]] := eulerAnglesFromQ[q, {3, 2, 1}];
With[{randomizeAngles =
    {RandomReal[{-π, π}], RandomReal[{-π / 2, π / 2}], RandomReal[{-π, π}]} &},
 DynamicModule[{ψ, θ, ϕ},
```

```
{ψ, θ, ϕ} = randomizeAngles[];
Manipulate[
 Grid[{{{ψ, θ, ϕ}, ψθϕFromQ[Q[ψ, θ, ϕ]]},
   {ListLinePlot[Table[ψθϕFromQ[Q[ψm, θ, ϕ]][[1]], {ψm, -π, π, 0.02 π}]]},
   {ListLinePlot[Table[ψθϕFromQ[Q[ψ, θm, ϕ]][[2]], {θm, -π / 2, π / 2, 0.02 π}]]},
   {ListLinePlot[Table[ψθϕFromQ[Q[ψ, θ, ϕm]][[3]], {ϕm, -π, π, 0.02 π}]]}}],
 Control[Button[ "REFRESH RANDOMS! ", ({ψ, θ, ϕ} = randomizeAngles[])]]]]
```

Out[116]=



Integrators

These integrators are very general, not specialized to TQDL.

Standard Fourth-Order Runge-Kutta

In[117]:=

```
ClearAll[rk4];
rk4[v_, vprime_, t_, dt_?NumberQ, args___] :=
  With[{k1 = dt * vprime[t, v, Sequence @@ {args}]},
   With[{k2 = dt * vprime[t, v + k1 / 2, Sequence @@ {args}]},
    With[{k3 = dt * vprime[t, v + k2 / 2, Sequence @@ {args}]},
     With[{k4 = dt * vprime[t, v + k3, Sequence @@ {args}]},
      v + (k1 + 2 k2 + 2 k3 + k4) / 6]]]];
```

Fourth-Order Runge-Kutta-Fehlberg

https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta%E2%80%93Fehlberg_method

`rk45` is a drop-in replacement for `rk4`. We invite you to check that our spherical pendulum performs no better with `rk45` than with `rk4`.

In[119]:=

```
ClearAll[rk45];
Module[{eps = 1.0 * 10^-10, result, TE = 1.0, rk45step},
  With[{A = {0, 2./9, 1./3, 3./4, 1., 5./6},

       {{Null, Null, Null, Null, Null},
        {2./9, Null, Null, Null, Null},
        {1./12, 1./4, Null, Null, Null},
    B = {69./128, -243./128, 135./64, Null, Null},
        {-17./12, 27./4, -27./5, 16./15, Null},
        {65./432, -5./16, 13./16, 4./27, 5./144}},

    C = {1./9, 0, 9./20, 16./45, 1./12, Null}, (*unused?*)
    CH = {47./450, 0, 12./25, 32./225, 1./30, 6./25},
    CT = {1./150, 0, -3./100, 16./75, 1./20, -6./25}},
   rk45step[v_, vprime_, t_, h_, args___] :=
    Module[{},
     With[{k1 = h vprime[t + A[[1]] h, v, Sequence @@ {args}]},
      With[{k2 = h vprime[t + A[[2]] h, v + B[[2, 1]] k1, Sequence @@ {args}]},
       With[
        {k3 = h vprime[t + A[[3]] h, v + B[[3, 1]] k1 + B[[3, 2]] k2, Sequence @@ {args}]},
        With[{k4 = h vprime[t + A[[4]] h,
              v + B[[4, 1]] k1 + B[[4, 2]] k2 + B[[4, 3]] k3, Sequence @@ {args}]},
         With[{k5 = h vprime[t + A[[5]] h, v + B[[5, 1]] k1 + B[[5, 2]]
                 k2 + B[[5, 3]] k3 + B[[5, 4]] k4, Sequence @@ {args}]},
          With[{k6 = h vprime[t + A[[6]] h, v + B[[6, 1]] k1 + B[[6, 2]] k2 +
                 B[[6, 3]] k3 + B[[6, 4]] k4 + B[[6, 5]] k5, Sequence @@ {args}]},
           result = v + CH[[1]] k1 + CH[[2]] k2 + CH[[3]] k3 + CH[[4]] k4 + CH[[5]] k5 + CH[[6]] k6;
           TE = Norm[CT[[1]] k1 + CT[[2]] k2 + CT[[3]] k3 + CT[[4]] k4 + CT[[5]] k5 + CT[[6]] k6];
          ]]]]]]];
  rk45[v_, vprime_, t_, h_, args___] :=
   Module[{hnew = h},
```

```
    TE = 1.0; (* ensure at least one iteration *)
    While[TE > eps,
     Module[{},
      rk45step[v, vprime, t, hnew, args];
      hnew = 0.9 hnew (eps/TE)^(1./5)]];
    result]];
```

The following stepper functions don't depend on time, so we just pass 0 in the time slot.

In[121]:=

```
stepVersorFromBodyAngVel := rqb2sn;
rqb2sn[rqb2snm1_Quaternion, ωbnm1_List, dt_?NumberQ, integrator_] :=
  normalize@integrator[rqb2snm1, drqb2sdt, 0, dt, ωbnm1];
```

In[123]:=

```
stepAngVelBodyFrame := ωbn;
ωbn[ωbnm1_List, mib_List, miib_List, dt_?NumberQ, τs_, rq_, integrator_] :=
  integrator[ωbnm1, dωbdt, 0, dt, mib, miib, τs, rq];
```

Global Constants

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[125]:=

```
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};
```

Free Rotation Benchmark for TQDL & RK4

This section contains unit tests for TQDL & RK4.

Show Apparatus

`showApparatus` is for graphics only, embodying no interesting physics or numerics. Most of the bulk concerns a head's-up display that requires frequent tweaking. We did not take the effort to properly abstract the head's-up display.

In[128]:=

```
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][
           "|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][
           "ωbᵀIbωb/2 = " <> ToString[NumberForm[Sqrt[1/2 ωb.Ib.ωb], {10, 4}]]],
          {-.95, -1, placeZ}],

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

         placeZ += displaceZ;
         Text[myFont[Blue][
           "ωsᵀLs/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 the physicist's 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[132]:=

```
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[132]=

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

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

Run Sim Forced Motion (TQDL & RK4)

- **Programming Note**—`runSimForcedRotationalMotion` illustrates an idiom for implicit animation looping via the combination of Dynamic and DynamicModule. Here is a tiny example:

In[133]:=

```
DynamicModule[{t = 0, dt = 0.03},
 Dynamic[t += dt; t]]
```

Out[133]=

```
0.78
```

In[134]:=

```
ClearAll[oneStepForcedRotationalMotion];
oneStepForcedRotationalMotion[ωb_, rq_, Ib_, Ibi_, dt_, τs_] :=
  With[{
    ωbNew = ωbn[ωb, Ib, Ibi, dt, τs, rq, rk4], (* changed to rk45 -- no help *)
    rqNew = rqb2sn[rq, ωb, dt, rk4]},
   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]]]]];
```

Dzhanybekhov

A benchmark and standard test for rigid-body rotation (search the web for "*dzhanibekov [sic] effect youtube*," taking care to spell with the (correct) 'k' rather than our (incorrect) 'kh.').

In[138]:=

```
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 ms, this slowly leaks energy and angular momentum.

Energy is the same in the non-rotating body frame b and in the inertial or space frame s because there is no translational motion. Ditto for magnitudes of angular momentum. There are three independent computations of energy: $\frac{1}{2}\,\omega_b{}^\intercal\, I_b\, \omega_b$ and $\frac{1}{2}\,\omega_b{}^\intercal L_b$ in the body frame, and $\frac{1}{2}\,\omega_b{}^\intercal L_b$ in the inertial space frame.

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.

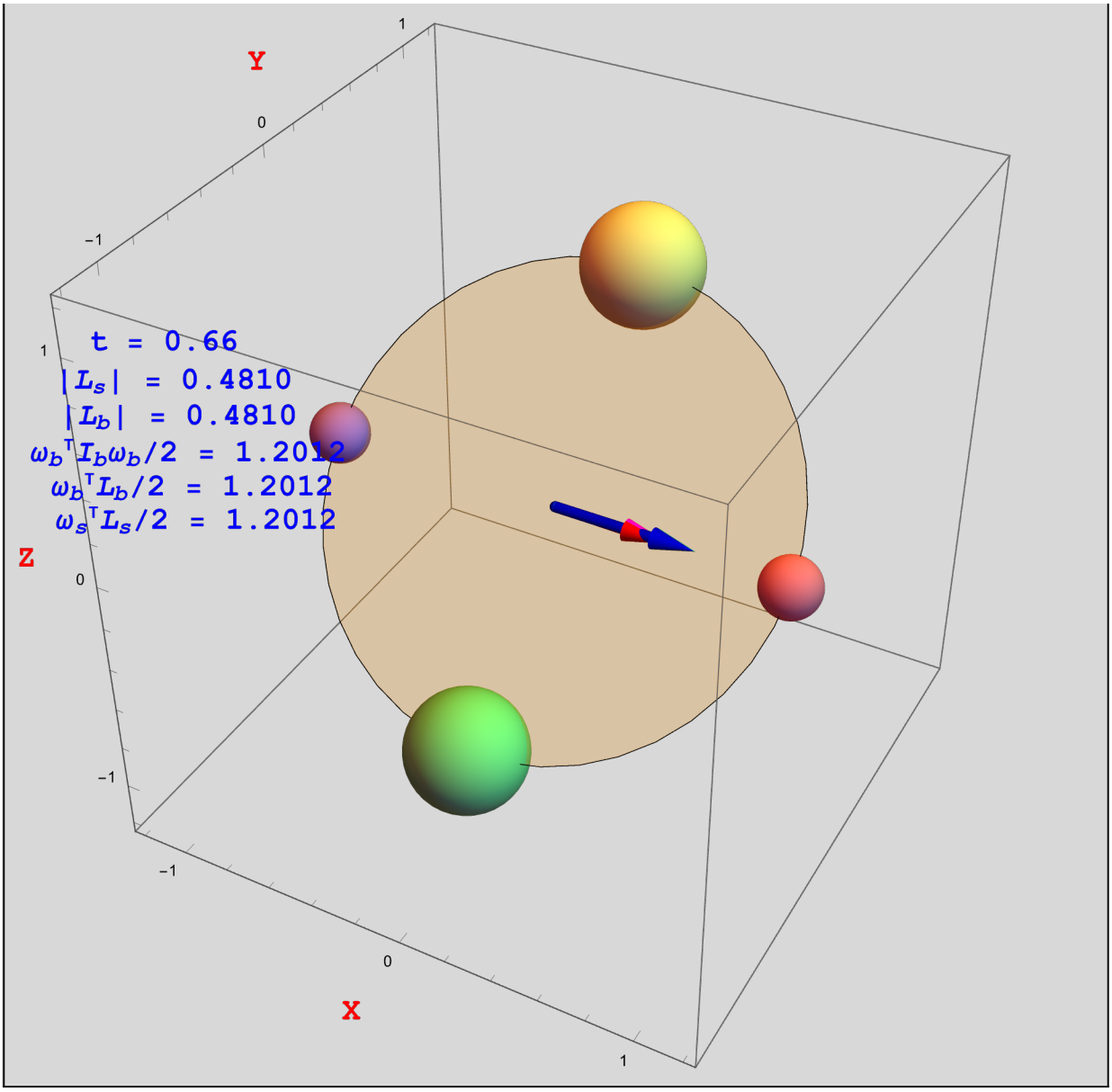
See the magenta arrow wiggle a little when the apparatus flips. This is a bad sign. It means that the computation of angular momentum in the inertial frame has numerical trouble. It spontaneously corrects, but this trouble needs deeper investigation.

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

In[142]:=

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

Out[142]=

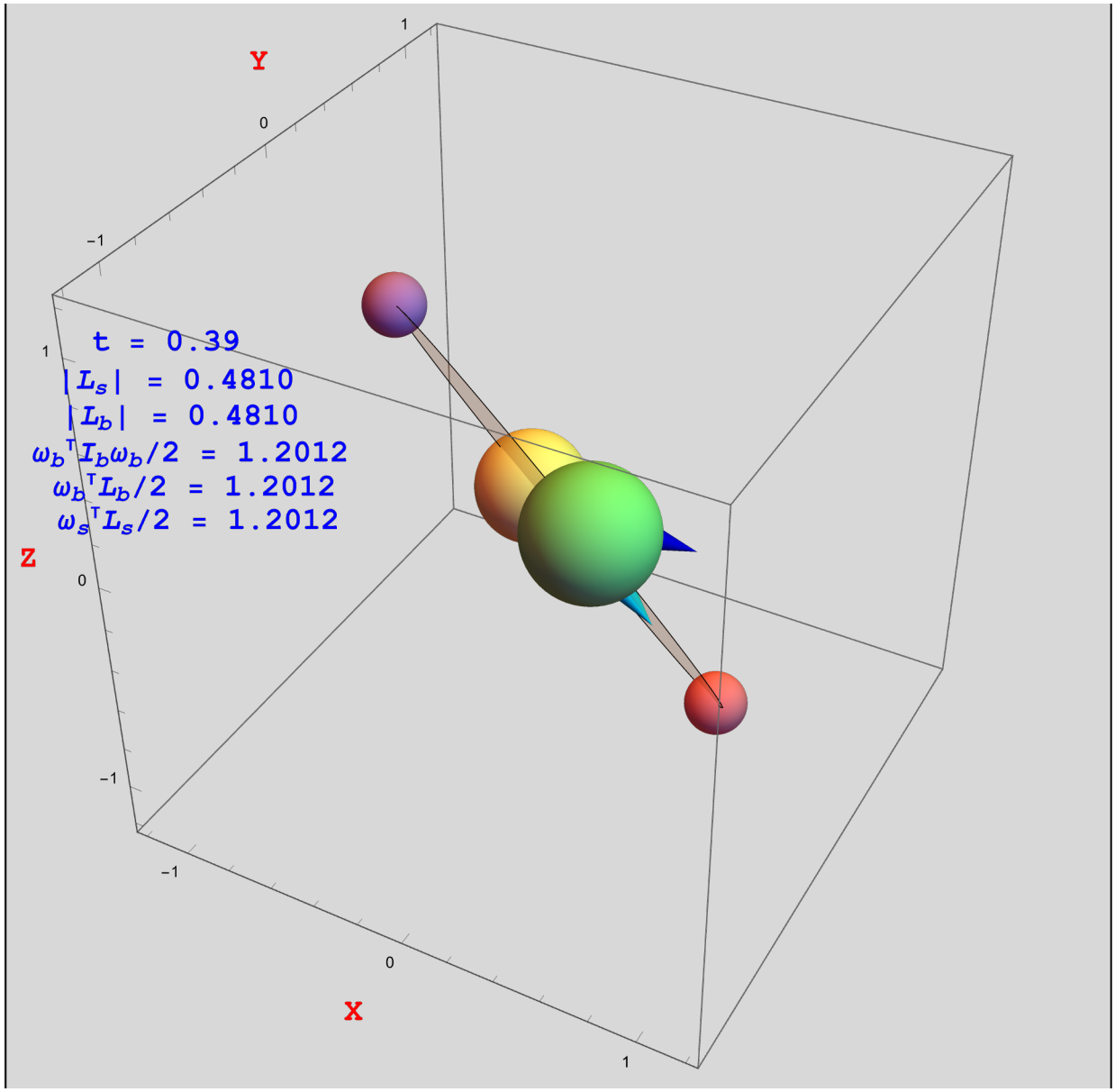


Notice that the values of kinetic energy and magnitude of angular momentum do not depend on initial orientation.

In[143]:=

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

Out[143]=



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 or RK45.

Rig (apparatus)

In[144]:=

```
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[146]:=

```
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[150]:=

```
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];
```

Accumulating Energy and Angular Momentum

In the body frame, reaction to gravitation points upward with a point of application at the base. The vector from the center of gravity to the point of contact in the inertial space frame is $-c_s$. The torque is

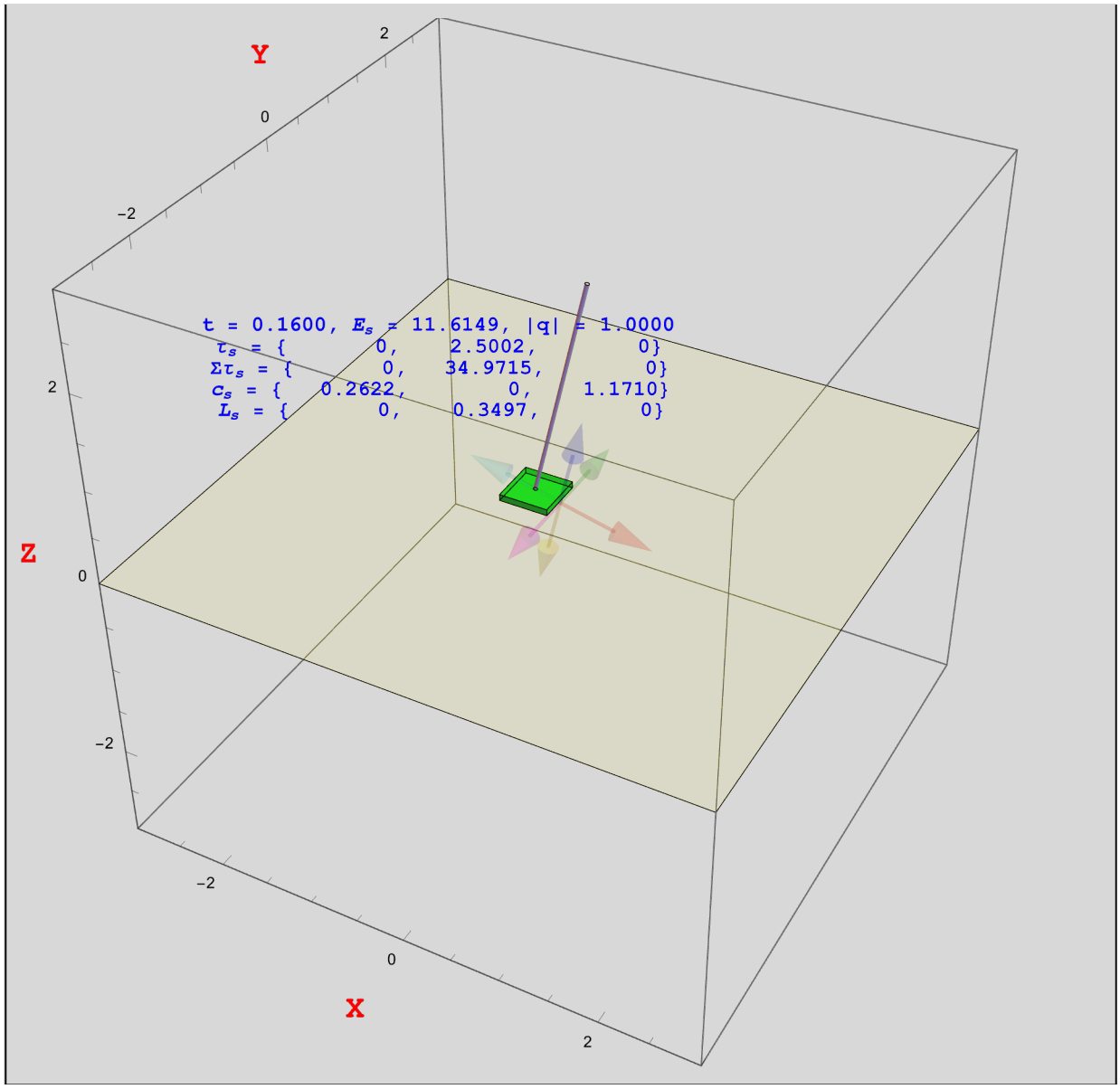
$(-c_s \times m\ |g|\ e_3) = (m\ |g|\ e_3 \times c_s)$, with × the 3D vector cross product. The gravitational force vector is applied at the same point with the same signs. The integration step size is 10 ms, as before.

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 ms, an unphysical result. The integration also diverges with a time step of 1 ms, as you may check by changing `dt` below. It is unpleasantly slow to watch, but we invite you to try it. The time step must be reduced by a factor of 100 to 100 μs to prevent swing-around. At that time step, the animation is *intolerably* slow, taking 10 minutes or so to swing up. In any event, a real pendulum with a frictionless mount would not accumulate energy or angular momentum. We shall see that the discrete integrator has no such energy accumulation.

In[155]:=

```
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[155]=



Addressing the Mystery

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

In[156]:=

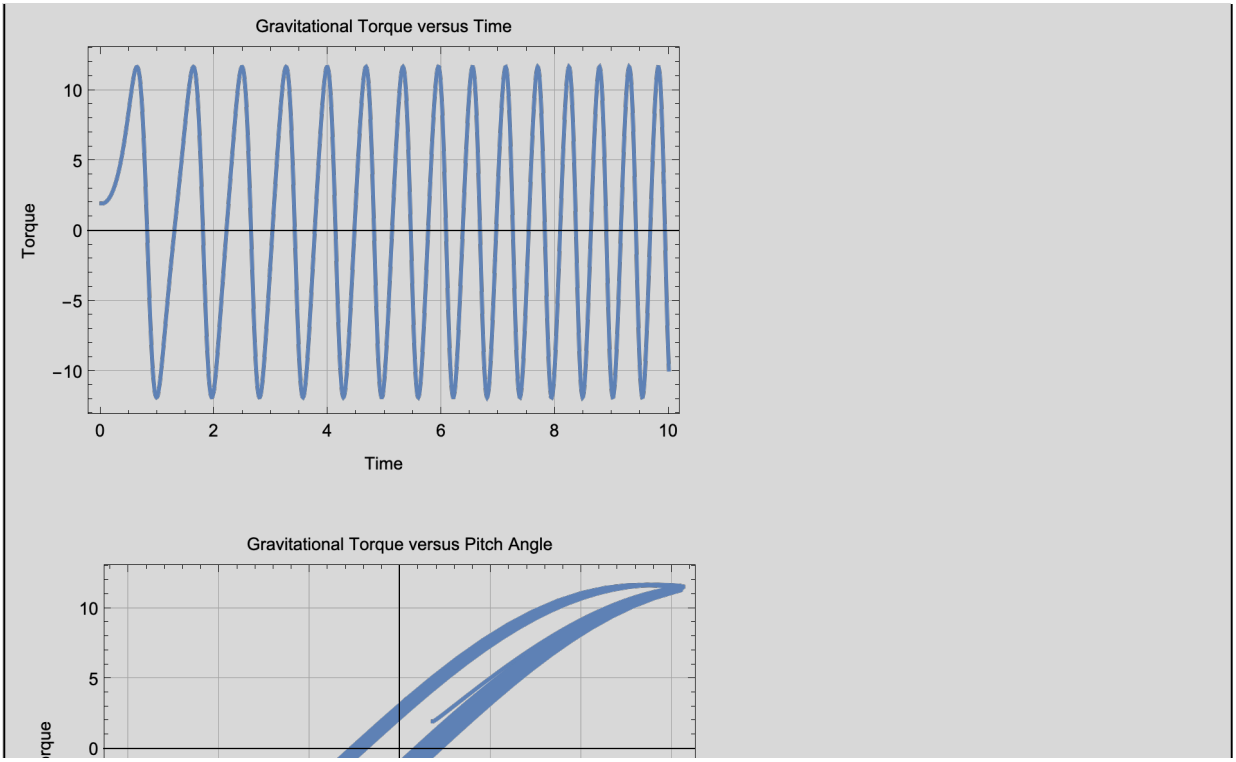
```
ClearAll[ψs, θs, ϕs, τss];
With[{epsilon = 0.01,
  (* Adjust here ~~> *)dt = 0.01, nMax = 1000,
  h = 1 / 15., w = 1 / 4., vu = 3.},
 With[{
   Ib = rig["moment of inertia"][[1]], Ibi = rig["moment of inertia"][[2]],
```

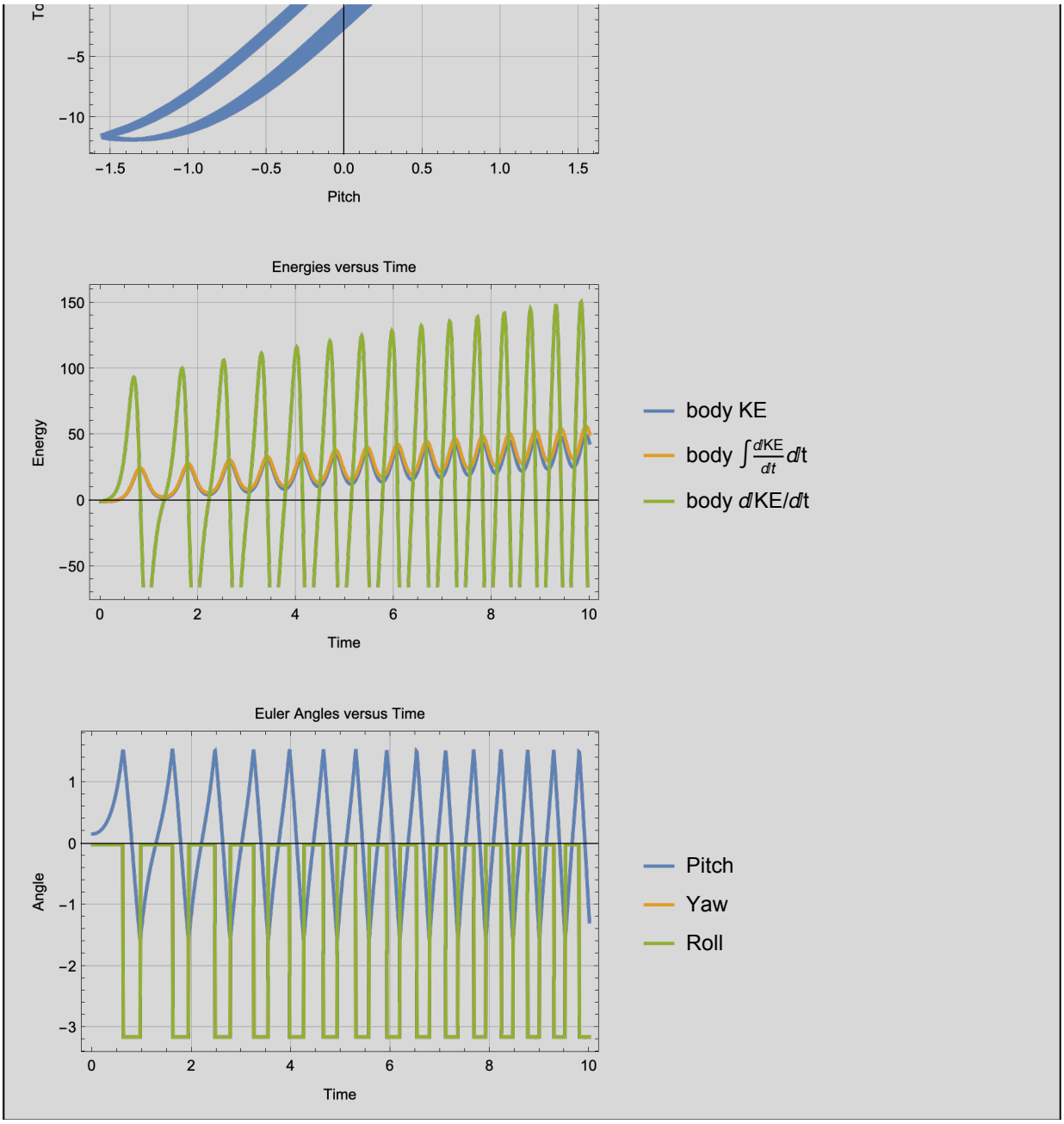
```
  cb = rig["cb"], rq0 = Q[0, 10 °, 0 * -0.5 °]},
 Module[
  {rq = rq0, ωb = o, ωs = o, Lb = o, Ls = o, pos = o, ps = o, force = o, τs = o, t = 0,
   cs = rv[rq0, cb], θv = θvrq[rq0], Στs = o},
  Module[
   {n = 0, τn = ConstantArray[0, nMax], Στsn = ConstantArray[0, nMax], ψ, θ, ϕ,
    ωbLast = ωb, 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]},
   tn = ConstantArray[0, nMax];
   ψs = ConstantArray[0, nMax];
   θs = ConstantArray[0, nMax];
   ϕs = ConstantArray[0, nMax];
   τss = ConstantArray[0, nMax];
   times = ConstantArray[0, nMax];
   For[(n = 1; t = 0), (n ≤ nMax), (n++),
    τs = ((rig["mass"] Abs[g] e3 + force) × (cs));
    τss[[n]] = τs[[2]];
    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];
    {ψ, θ, ϕ} = ψθϕFromQ[rq];
    ψs[[n]] = ψ; θs[[n]] = θ; ϕs[[n]] = ϕ;
    (* ---------------- 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.Ls;
    dEsrotdtn[[n]] = dEsrotdt;
    Esrotn[[n]] = ωs.Ls / 2;
    ΣdtdEsrotdt += (ωs - ωsLast).Ls;
```

```
    ΣdtdEsrotdtn[[n]] = ΣdtdEsrotdt;
    ωsLast = ωs;
    (* ------------------------------------------------ *)
    t += dt];
  ListLinePlot[{{tn, τss}ᵀ}, Frame → True,
    GridLines → Automatic, FrameLabel → {{"Torque", ""},
      {"Time", "Gravitational Torque versus Time"}}, ImageSize → Medium] ×
   ListLinePlot[{{tn, θs}ᵀ, {tn, ψs}ᵀ, {tn, ϕs}ᵀ},
    PlotLegends → {"Pitch", "Yaw", "Roll"}, Frame → True, GridLines → Automatic,
    FrameLabel → {{"Angle", ""}, {"Time", "Euler Angles versus Time"}},
    ImageSize → Medium] ×
   ListLinePlot[{{θs, τss}ᵀ}, Frame → True,
    GridLines → Automatic, FrameLabel → {{"Torque", ""}, {"Pitch",
       "Gravitational Torque versus Pitch Angle"}}, ImageSize → Medium] ×
   ListLinePlot[{{tn, Ebrotn}ᵀ, {tn, ΣdtdEbrotdtn}ᵀ, {tn, dEbrotdtn}ᵀ},
    PlotLegends → {"body KE", "body ∫dKE/dt dt", "body dKE/dt"},
    Frame → True, GridLines → Automatic, FrameLabel →
     {{"Energy", ""}, {"Time", "Energies versus Time"}}, ImageSize → Medium]
  (*,ListLinePlot[{{tn,Esrotn}ᵀ,{tn,ΣdtdEsrotdtn}ᵀ,{tn,dEsrotdtn}ᵀ}]*)

  (*Quiet@ListLinePlot[{{tn,#[[2]]&/@Στsn}ᵀ,{tn,#[[2]]&/@τn}ᵀ}]*)
 ]] ]]
```

Out[157]=





First, the gravitational torque is qualitatively wrong. It should be zero when pitch is 0, with the baton upright or dangling. This is the first sign of trouble.

The plots 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.

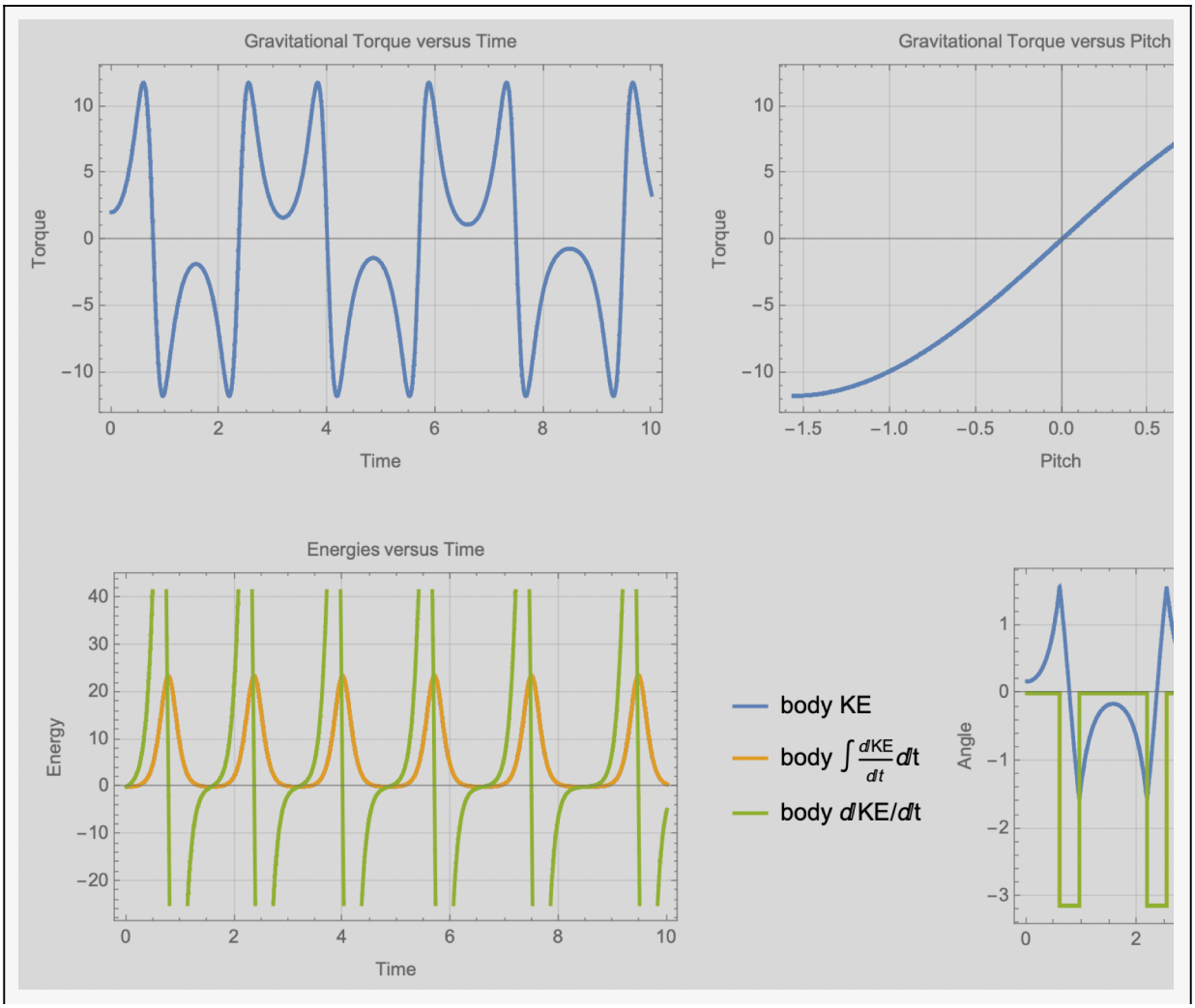
The plots for yaw and roll coincide. All angles exhibit discontinuities due merely to our choices for branch cuts in conversion from quaternions to Euler angles. The quaternions vary smoothly, as shown by the smoothness of the torque-versus-pitch graph and by the animation.

If we decrease dt 100 μs, we get something much more sensible. It's visibly not conservative over ten

simulated seconds, but it shows the expected torque versus pitch, plus the body KE and the integral of its derivative coincide within the resolution of the plot.

The “whiplash” in torque and pitch below are artificial, due to discontinuities in converting quaternions to Euler angles. We’ll see the same phenomenon in the discrete solution. We could change the conversions, but it’s just as easy to interpret the angles properly. One should imagine the concave portions of the pitch and torque plots flipped upside down, resulting in a smooth, sinusoidal signal.

With timestamp reduced to 100 μs, it takes several minutes to generate the plot, so I save only a screenshot.



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.

Discrete Dynamical Integrator

What are good generalized coordinates and velocities? Spherical coordinates have a singularity at the North pole, right where we want to control the baton. A better choice are pitch and cone roll (not axis roll!): co-latitude δ and 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 singularity on the Equator. The demonstration below makes this visually clear. Both δ and η are non-singular 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 singularities, not linger at them.

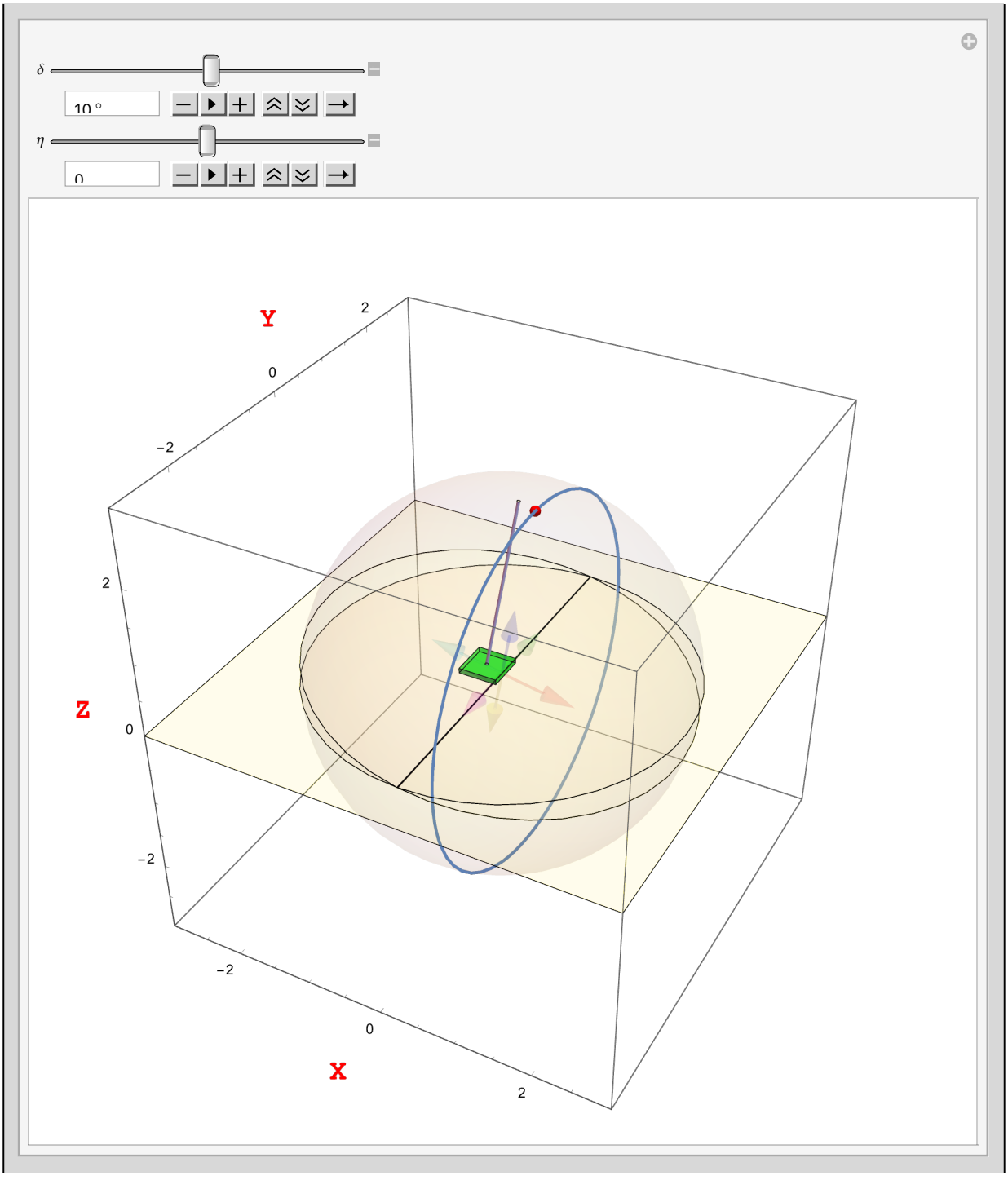
The actual physical system is the center of mass of the baton, swinging around the origin. The sliding cart in the animations is a visual fiction. The center of mass of the baton goes up and down.

Demonstration of the Coordinate System

In[158]:=

```
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 °}, -360 °, 360 °, 1 °, Appearance → {"Open"}},
    {{η, 0 °}, -360 °, 360 °, 1 °, Appearance → {"Open"}}]]]]
```

Out[158]=



Dynamical Set-Up

Center of mass

In[159]:=

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

Out[161]//MatrixForm=

$$\begin{pmatrix} \text{Sin}[\delta]\, c_z \\ \text{Cos}[\delta]\, \text{Sin}[\eta]\, c_z \\ \text{Cos}[\delta]\, \text{Cos}[\eta]\, c_z \end{pmatrix}$$

z-Height

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

In[162]:=

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

Out[164]=

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

Potential Energy

We'll need symbolical and numerical functions for energy. The symbolical functions support *Mathematica*'s calculus with functions like $\delta[t]$, $\eta[t]$, $\delta'[t]$, and $\eta'[t]$. That calculus yields the equations of motion.[15] The numerical energy functions support graphics and discrete calculations.

numRules

For numerical functions, we replace parametric functions with unbound symbols.

In[165]:=

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

In[167]:=
```
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[168]=
$$9.81\, m \cos[\delta] \cos[\eta]\, c_z$$

Out[170]=
$$11.772 \cos[\delta[t]] \cos[\eta[t]]$$

Velocity Vector

The kinetic energy depends on the velocity vector.

In[171]:=
```
ClearAll[v];
(v[{δ_, η_}] := D[cm[{δ, η}], t]);
v[{δ[t], η[t]}] // FullSimplify // MatrixForm
```

Out[173]//MatrixForm=
$$\begin{pmatrix} \cos[\delta[t]]\, c_z\, \delta'[t] \\ c_z\,(-\sin[\delta[t]] \sin[\eta[t]]\, \delta'[t] + \cos[\delta[t]] \cos[\eta[t]]\, \eta'[t]) \\ -c_z\,(\cos[\eta[t]] \sin[\delta[t]]\, \delta'[t] + \cos[\delta[t]] \sin[\eta[t]]\, \eta'[t]) \end{pmatrix}$$

Kinetic Energy

In[174]:=

```
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[176]=

$$\frac{1}{2} m c_z^2 \left(\delta'[t]^2 + \text{Cos}[\delta[t]]^2 \eta'[t]^2\right)$$

Out[180]=

$$0.72\, D\delta^2 + 0.36\, D\eta^2 + 0.36\, D\eta^2\, \text{Cos}[2\,\delta]$$

Lagrangians

`L` is for the late-discretized, standard numerical solution. `Lnum` is for the early-discretized Lagrangian, L_d, below.

In[181]:=

```
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[182]=

$$0.25\, m\, c_z \left(-39.24\, \text{Cos}[\delta[t]]\, \text{Cos}[\eta[t]] + c_z \left(2.\, \delta'[t]^2 + (1. + 1.\, \text{Cos}[2\,\delta[t]])\, \eta'[t]^2\right)\right)$$

Out[184]=

$$0.72\, D\delta^2 + 0.36\, D\eta^2 + 0.36\, D\eta^2\, \text{Cos}[2\,\delta] - 11.772\, \text{Cos}[\delta]\, \text{Cos}[\eta]$$

Leqns

Mathematica has symbolical magic for the Euler-Lagrange equations. See Reference [15].

In[185]:=
```
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[185]=
$$\left\{m\, c_z \left(-9.81 \text{Cos}[\eta[t]]\, \text{Sin}[\delta[t]] + c_z \left(\text{Cos}[\delta[t]]\, \text{Sin}[\delta[t]]\, \eta'[t]^2 + \delta''[t]\right)\right) == 0,\right.$$
$$\left. m\, \text{Cos}[\delta[t]]\, c_z \left(-9.81 \text{Sin}[\eta[t]] + c_z \left(-2.\, \text{Sin}[\delta[t]]\, \delta'[t]\, \eta'[t] + \text{Cos}[\delta[t]]\, \eta''[t]\right)\right) == 0\right\}$$

State-Space Form

In[186]:=
```
ClearAll[stEqns];
stEqns =
 Equal @@@ (Solve[Leqns, {δ''[t], η''[t]}] // FullSimplify // Chop // Part[#, 1] &)
```

Out[187]=
$$\left\{\delta''[t] == \text{Sin}[\delta[t]] \left(\frac{9.81\, \text{Cos}[\eta[t]]}{c_z} - 1.\, \text{Cos}[\delta[t]]\, \eta'[t]^2\right),\right.$$
$$\left.\eta''[t] == \frac{9.81\, \text{Sec}[\delta[t]]\, \text{Sin}[\eta[t]]}{c_z} + 2.\, \text{Tan}[\delta[t]]\, \delta'[t]\, \eta'[t]\right\}$$

Late Discretization: Numerical Solution of Continuous Equations

Solve the system for tLim = 25 simulated seconds.

In[188]:=

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

Out[191]=

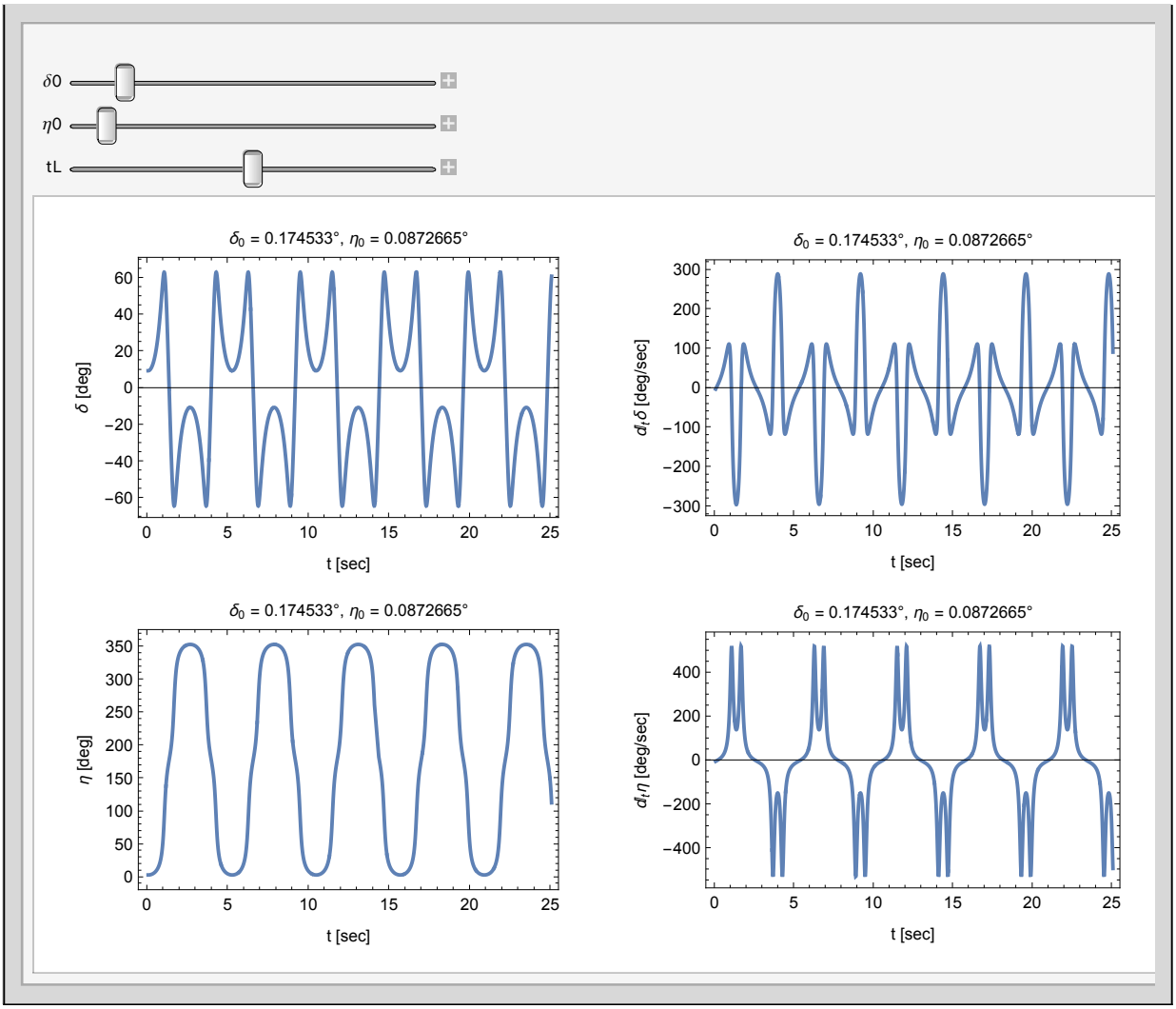
$$\{\{\delta[t] \to \text{InterpolatingFunction}[\langle\text{Domain: } \{\{0., 25.\}\} \text{ Output: scalar}\rangle][t],$$
$$\eta[t] \to \text{InterpolatingFunction}[\langle\text{Domain: } \{\{0., 25.\}\} \text{ Output: scalar}\rangle][t],$$
$$\delta'[t] \to \text{InterpolatingFunction}[\langle\text{Domain: } \{\{0., 25.\}\} \text{ Output: scalar}\rangle][t],$$
$$\eta'[t] \to \text{InterpolatingFunction}[\langle\text{Domain: } \{\{0., 25.\}\} \text{ Output: scalar}\rangle][t]\}\}$$

Plot the Solutions

In[192]:=

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

Out[193]=



The angles and their derivatives exhibit discontinuities as we saw with quaternions, due merely to choice of branch cuts. The animation below shows smooth motion, plus the pitch angle, here, is visually identical to the pitch-angle plot for the 100-μs quaternion solution.

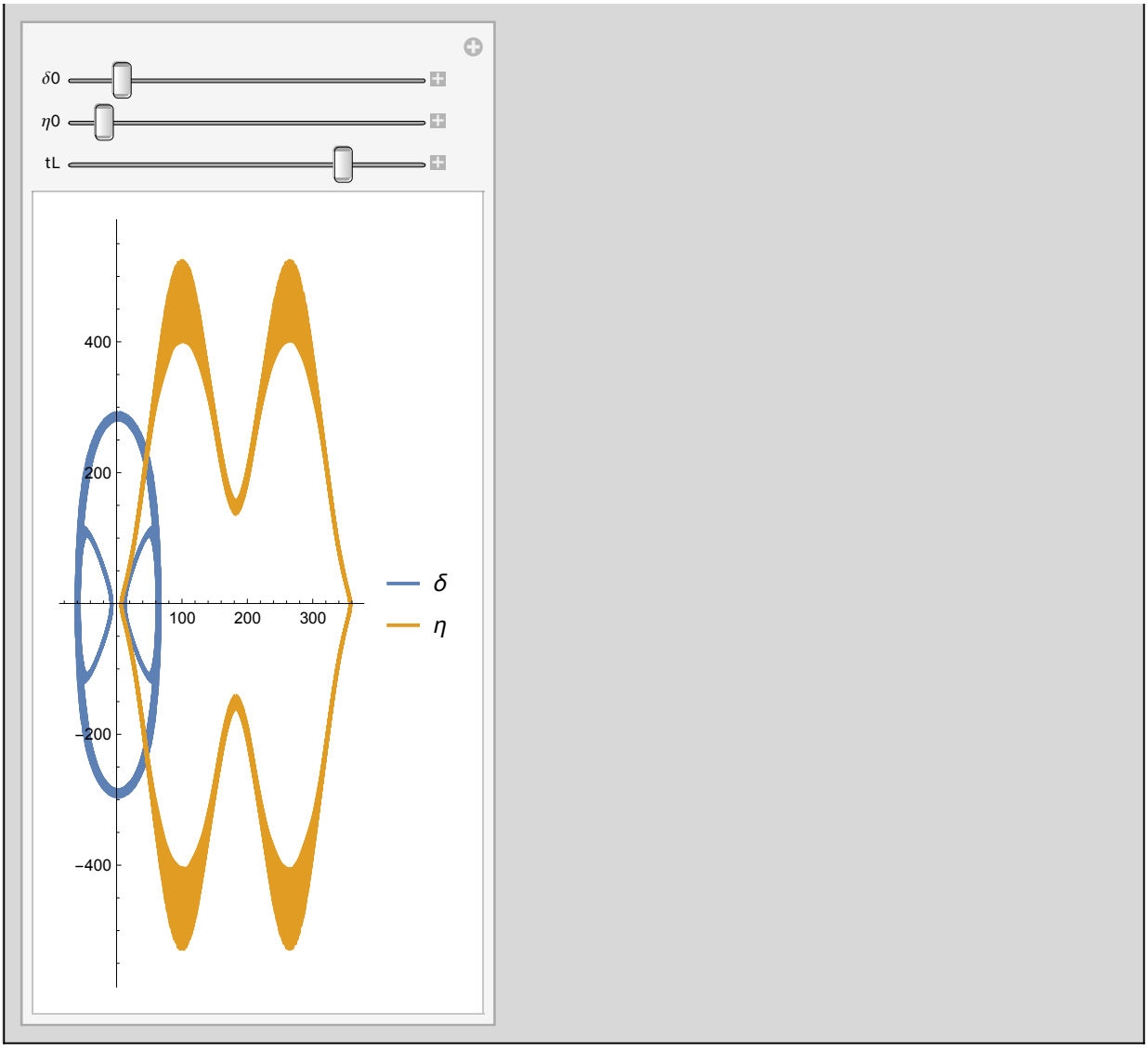
Phase Portrait

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 Runge-Kutta quaternions. This is going to be a good ground-truth.

In[194]:=

```
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[194]=



Ground-Truth Animation

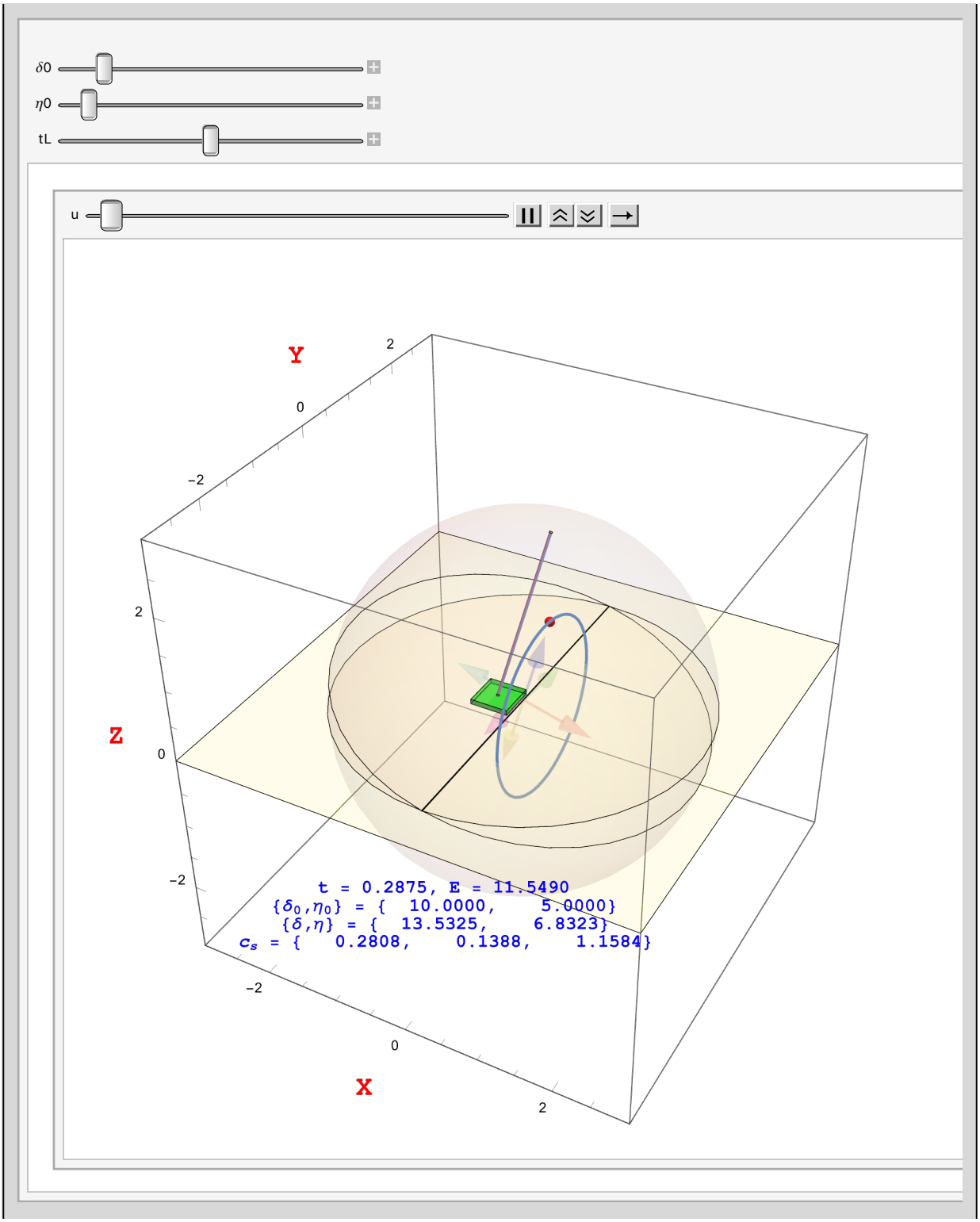
This is pretty convincing. Energy is well conserved. We'd only have to reboot it after a long time. If we were satisfied with an integrator that works only inside *Mathematica*, we'd be just about done. But we want an integrator we can code up in any programming language.

In[195]:=

```
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["{δ0,η0} = " <> 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[195]=



Early Discretization

Discretize the dynamical variables and the Lagrangian, itself, via Equation 7 of Reference [10]. First, a reminder:

In[196]:=
```
?? Lnum
```

Out[196]=

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 ***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 actually discretized ***action*** (units of energy × time), being the product of `LNum` (units of energy) and the finite (constant) time increment `dt`.

In[197]:=
```
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***, without generalized force terms, are

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

In[199]:=

```
ClearAll[d1ld, d2ld];
(* D1Ld(qk,qk+1) *)
d1ld[{δk_, ηk_}, {δkp1_, ηkp1_}, dt_] :=
  (* {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])};
(* +D2Ld(qk-1,qk) *)
d2ld[{δkm1_, ηkm1_}, {δk_, ηk_}, dt_] :=
  (* +{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 - δkm1) +
      5.886 dt^2 Cos[(ηk + ηkm1)/2] Sin[(δk + δkm1)/2] - 0.36 (ηk - ηkm1)^2 Sin[δk + δkm1]),
   1/dt (0.72 ηk - 0.72 ηkm1 + (0.72 ηk - 0.72 ηkm1) Cos[δk + δkm1] +
      5.886 dt^2 Cos[(δk + δkm1)/2] Sin[(ηk + ηkm1)/2])};
```

Numerical Solution for δ_1 and η_1

All we need is `FindRoot`, which we could replace with an explicit Newton-Raphson solver in C, C++, Fortran, or Python, for instance. We bootstrap the integrator by taking the first two positions from the late-discretized solution

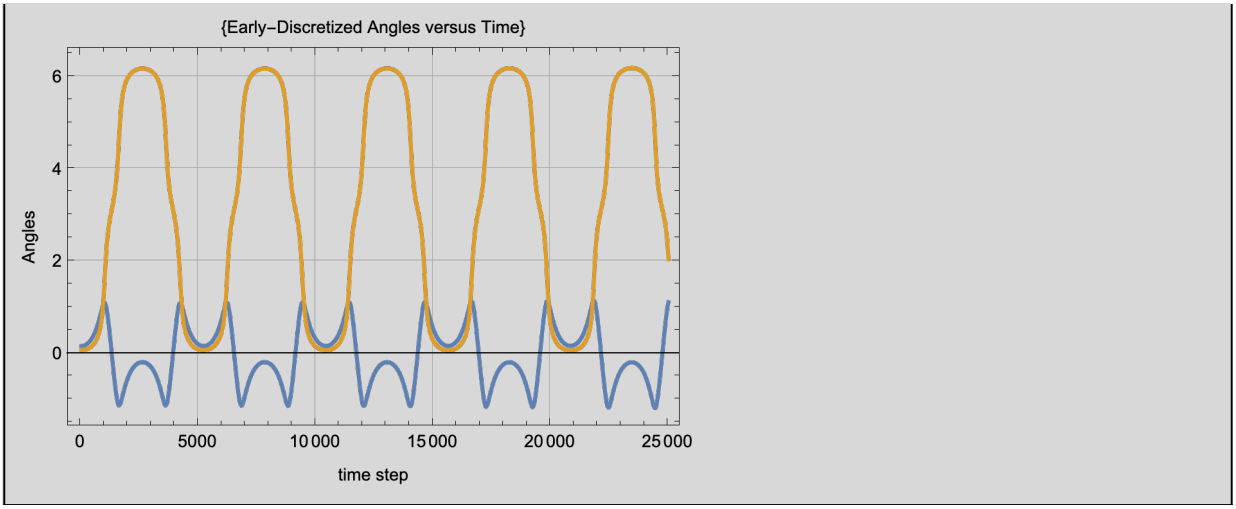
Notice that the angles are qualitatively the same as for the late-discretized solution.

In[202]:=

```
ClearAll[δt, ηt, dt, fh, δ0, η0, δ1, η1,
  δkm1, ηkm1, δk, ηk, δkp1, ηkp1, tn, len, nvars, dsoln];
δt = nsolns[[1, 1, 2]]; ηt = nsolns[[1, 2, 2]];
dt = 1 / 1000; (* sec *) fh = 1.0 * dt;
δ0 = δt /. {t → 0}; η0 = ηt /. {t → 0};
δ1 = δt /. {t → dt}; η1 = ηt /. {t → dt};
tn = 25; (* sec *)
len = tn / dt;
nvars = 2;
dsoln = ConstantArray[0, {2 nvars, len}];
dsoln[[1, 1]] = δ0; dsoln[[2, 1]] = η0;
dsoln[[1, 2]] = δ1; dsoln[[2, 2]] = η1;
Echo@Quiet@AbsoluteTiming@
    For[j = 2, j < len, j += 1,
     Module[{pj, δkm1, ηkm1, δk, ηk, δkp1, ηkp1, r},
      δkm1 = dsoln[[1, j - 1]]; ηkm1 = dsoln[[2, j - 1]];
      δk = dsoln[[1, j]]; ηk = dsoln[[2, j]];
      pj = d1ld[{δk, ηk}, {δkp1, ηkp1}, dt] + d2ld[{δkm1, ηkm1}, {δk, ηk}, dt];
      r = FindRoot[pj, {{δkp1, δk}, {ηkp1, ηk}}];
      dsoln[[1 ;; 2, j + 1]] = r[[ ;; , 2]];
      dsoln[[3 ;; 4, j + 1]] = (dsoln[[1 ;; 2, j + 1]] - dsoln[[1 ;; 2, j]]) / fh;  ]];
ListLinePlot[dsoln[[1 ;; 2,  ;;]], Frame → True, GridLines → Automatic, FrameLabel →
  {{"Angles", ""}, {"time step", {"Early-Discretized Angles versus Time"}}}]
```

» {7.01195, Null}

Out[214]=



Energies of the Late Discretization

In[215]:=

```
ClearAll[TnumFlat, VnumFlat];
TnumFlat[δ_, η_, dδ_, dη_] := Tnum[{δ, η}, {dδ, dη}];
VnumFlat[δ_, η_] := Vnum[{δ, η}];
```

In[218]:=

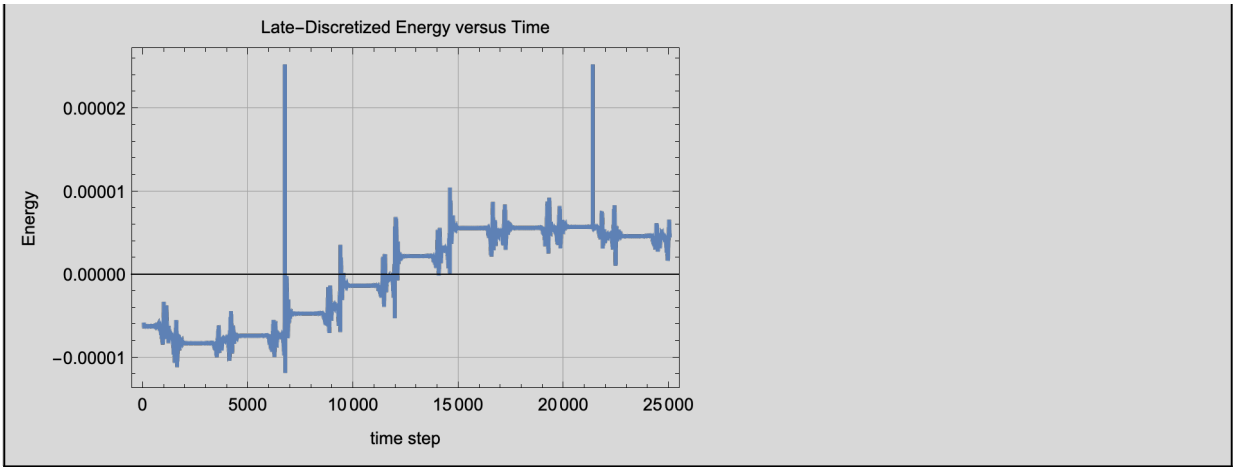
```
dδ = dsoln[[1]];
dη = dsoln[[2]];
dδDot = dsoln[[3]];
dηDot = dsoln[[4]];
dT = MapThread[TnumFlat, {dδ, dη, dδDot, dηDot}];
dV = MapThread[VnumFlat, {dδ, dη}];
dE = dT + dV;
mdE = Mean[dE];
nδ = Table[nsolns[[1, 1, 2]] /. t → u, {u, 0 dt, 24 999 dt, dt}];
nδDot = Table[nsolns[[1, 3, 2]] /. t → u, {u, 0 dt, 24 999 dt, dt}];
nη = Table[nsolns[[1, 2, 2]] /. t → u, {u, 0 dt, 24 999 dt, dt}];
nηDot = Table[nsolns[[1, 4, 2]] /. t → u, {u, 0 dt, 24 999 dt, dt}];
nT = MapThread[TnumFlat, {nδ, nη, nδDot, nηDot}];
nV = MapThread[VnumFlat, {nδ, nη}];
nE = nT + nV;
mnE = Mean[nE];
```

From the late-discretized solution, we calculate an energy of 11.5490. It slowly drifts upward in the fifth decimal place:

In[234]:=

```
ListLinePlot[nE - mnE, Frame → True, GridLines → Automatic, FrameLabel →
  {{"Energy", ""}, {"time step", "Late-Discretized Energy versus Time"}}]
```

Out[234]=



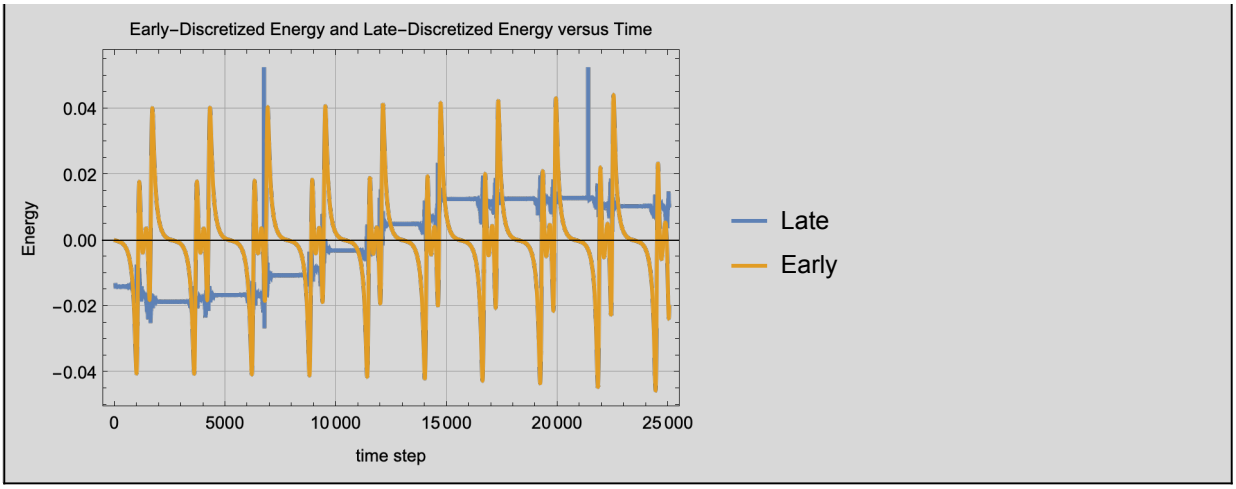
The late-discretized solution has a mean energy of 11.5486+/-0.014, differing from early-discretized energy in the 5-th figure and with a standard deviation consistent with that difference. The early-

discretized solution has a periodic error of about that amplitude, about 2250 times larger than the standard deviation of the energy of the early-discretized solution. However, the big difference is that the late-discretized solution has no discernible secular (linear) trend, whereas the early-discretized solution, when amplified by 2250, has a visually obvious trend.

In[235]:=

```
ListLinePlot[{(nE - mnE) * 2250, (dE - mdE)}, Frame → True,
 GridLines → Automatic, FrameLabel → {{"Energy", ""}, {"time step",
    "Early-Discretized Energy and Late-Discretized Energy versus Time"}},
 PlotLegends → {"Late", "Early"}]
```

Out[235]=



Conclusion

Eventually, the late-discretized solution would accumulate energy, just like the disastrous quaternion-and-Runge-Kutta solution, and would have to be rebooted. We see, again, that early discretization conserves energy much better than late discretization, though with a much larger periodic error. That periodic error is obviously growing in amplitude in the plot, and would eventually require a reboot, too, but for a very different reason.