Table of Contents

DASHPOT IN SPINNING FREE FALL: EXTENDED KALMAN FILTER

Brian Beckman

6 Nov 2017

ABSTRACT

A *dashpot* is a mechanical model of a dipole damped harmonic oscilator, thus a fundamental component of mechanical systems. It has a mass, spring, damper, and non-zero resting length. We consider the problem of estimating the four states and four constant parameters of a dashpot spinning in free fall in a vertical plane. We can get credible ***tracking*** (state estimation) and ***system identification*** (parameter estimation) by observing only the angle because of coupling in the equations of motion. An ***Extended Kalman filter*** (EKF) achieves these estimates in constant memory. The particular form of the EKF here exploits ***functional fold***, enhancing clarity, modularity and testability of the software.

INTRODUCTION

Dynamical simulators of very high fidelity can be built entirely of dashpots (search "Rigs of Rods" and "TORCS"). A dashpot is a sliding rod with mass m, spring constant k in force per unit length, damper, and resting length l. The rod thus forms a damped harmonic oscillator of natural frequency $\sqrt{k/ml}$. Optionally, the rod can have:

1. masses only at the ends or distributed along the rod
2. plastic deformation constants (length or speed beyond which its rest length changes)
3. fracture properties (length or speed beyond which it permanently breaks)
4. static and dynamic friction for interacting with the environment
5. spin degrees of freedom for gyroscopic effects

It is easy to create a convincing simulation of tires, one of the most complicated objects in common use, as a truss of dashpots with these optional properties. We do not consider these optional properties further in this study. We also ignore aerodynamical forces like drag.

We estimate the state of a dashpot spinning in free fall in two dimensions observing only the angle of the rod with respect to horizontal. The Lagrangian for this system has four states:

- q, the extension of the rod
- $\dot{q}$, the time rate of change of the extension
- θ, the angle of the rod with respect to horizontal
- $\dot{\theta}$, the rate of change of the angle, or the *spin rate*

There are also four constant parameters:

- m - the total mass consisting of $m/2$ at each end; the connecting rod is massless
- k - the spring constant in units of force per unit length
- ν - the damping constant in units of force per unit speed
- l - the resting, zero-force extension of the spring

Given an initial, non-zero spin rate, we expect the rod to lengthen under centrifugal force. When the rod lengthens, its moment of inertia increases. Spin rate goes down immediately because the product of spin rate and moment of inertia --- angular momentum --- is conserved in free fall. The system will rebound under the spring and spin rate will go back up again. Eventually, under action of the damper, the system will equilibrate the spin rate. We observe these effects numerically below.

PRELIMINARIES

PACKAGES AND JUPYTER MAGIC

The following are a superset of the package dependencies for this notebook.

In [1]:
```
%%bash
pip freeze
```

```
alabaster==0.7.10
anaconda-client==1.6.3
anaconda-navigator==1.6.2
anaconda-project==0.6.0
appdirs==1.4.0
appnope==0.1.0
appscript==1.0.1
apptools==4.4.0
args==0.1.0
asn1crypto==0.22.0
astor==0.5
astroid==1.4.9
astropy==1.3.2
attrs==17.2.0
Babel==2.5.1
backports-abc==0.4
backports.shutil-get-terminal-size==1.0.0
backports.ssl-match-hostname==3.5.0.1
backports.weakref==1.0.post1
beautifulsoup4==4.6.0
bitarray==0.8.1
blaze==0.10.1
bleach==1.5.0
bokeh==0.12.4
boto==2.46.1
bottle==0.12.10
Bottleneck==1.2.1
catkin-pkg==0.3.5
cdecimal==2.3
certifi==2017.11.5
cffi==1.10.0
chardet==3.0.4
click==5.1
clint==0.5.1
cloudpickle==0.2.2
clyent==1.2.2
colorama==0.3.7
conda==4.3.30
configobj==5.0.6
configparser==3.5.0
contextlib2==0.5.5
cryptography==1.8.1
cycler==0.10.0
Cython==0.25.2
cytoolz==0.8.2
dask==0.10.0
datashape==0.5.4
decorator==4.0.11
distributed==1.16.3
docutils==0.14
entrypoints==0.2.2
enum34==1.1.6
envisage==4.6.0
et-xmlfile==1.0.1
fastcache==1.0.2
filterpy==0.1.5
Flask==0.12.2
Flask-Cors==3.0.2
funcsigs==1.0.2
functools32==3.2.3.post2
futures==3.1.1
gevent==1.2.1
gnureadline==6.3.3
greenlet==0.4.12
grin==1.2.1
gym==0.9.3
h5py==2.7.0
HeapDict==1.0.0
html5lib==0.9999999
hy==0.11.0
hy-kernel==0.3.0
hypothesis==3.6.1
idna==2.6
imagesize==0.7.1
inflection==0.3.1
ipaddress==1.0.18
```

```
ipykernel==4.5.2
ipython==4.2.0
ipython-genutils==0.1.0
ipython-tikzmagic==0.1.0
ipywidgets==5.2.2
isort==4.2.5
itsdangerous==0.24
jdcal==1.3
jedi==0.10.2
Jinja2==2.10
jsonschema==2.5.1
jupyter==1.0.0
jupyter-client==4.4.0
jupyter-console==5.0.0
jupyter-contrib-core==0.3.3
jupyter-contrib-nbextensions==0.3.3
jupyter-core==4.2.1
jupyter-highlight-selected-word==0.1.0
jupyter-latex-envs==1.3.8.4
jupyter-nbextensions-configurator==0.2.8
kerberos==1.2.4
lazy-object-proxy==1.2.2
llvmlite==0.18.0
locket==0.2.0
lockfile==0.12.2
lxml==4.1.1
Markdown==2.6.9
MarkupSafe==1.0
matplotlib==1.5.1
mayavi==4.5.0
mistune==0.7.3
mock==2.0.0
more-itertools==2.5.0
mpi4py==2.0.0
mpmath==1.0.0
msgpack-python==0.4.8
multipledispatch==0.4.9
navigator-updater==0.1.0
nbconvert==5.0.0
nbformat==4.2.0
networkx==1.11
nltk==3.2.3
nose==1.3.7
notebook==4.3.1
numba==0.33.0+0.ge79330a.dirty
numexpr==2.6.2
numpy==1.13.3
numpydoc==0.6.0
odo==0.5.0
olefile==0.44
openpyxl==2.4.7
packaging==16.8
pandas==0.18.1
pandocfilters==1.4.1
partd==0.3.8
pathlib==1.0.1
pathlib2==2.1.0
patsy==0.4.1
pbr==3.1.1
pep8==1.7.0
pexpect==4.1.0
pickleshare==0.7.2
Pillow==3.2.0
Pint==0.8.1
platformio==3.2.1
plotly==2.0.1
ply==3.10
prompt-toolkit==1.0.9
protobuf==3.4.1
psutil==5.2.2
ptyprocess==0.5.1
py==1.4.33
PyAudio==0.2.7
pybullet==1.5.8
pyconfig==3.1.1
pycosat==0.6.2
```

```
pycparser==2.17
pycrypto==2.6.1
pycurl==7.43.0
pyface==5.1.0
pyflakes==1.5.0
pygame==1.9.3
pyglet==1.2.4
Pygments==2.2.0
pylab==0.1.3
pylint==1.6.4
pyodbc==4.0.16
PyOpenGL==3.1.0
PyOpenGL-accelerate==3.1.0
pyOpenSSL==17.0.0
pyparsing==2.1.5
pyrsistent==0.12.0
pyserial==3.2.1
pytest==3.0.7
python-dateutil==2.5.3
pythreejs==0.2.3
pytool==3.4.1
pytz==2017.3
PyWavelets==0.5.2
PyYAML==3.12
pyzmq==15.2.0
QtAwesome==0.4.4
qtconsole==4.2.1
QtPy==1.2.1
## FIXME: could not find svn URL in dependency_links for this package:
RBTools===0.5Amazon1.dev-r0
requests==2.18.4
rope==0.9.4
rosdep==0.11.7
rosdistro==0.6.2
rosinstall==0.7.8
rosinstall-generator==0.1.13
rospkg==1.1.1
rply==0.7.4
Rx==1.6.0
scandir==1.5
scikit-image==0.12.3
scikit-learn==0.17.1
scipy==0.19.1
seaborn==0.7.1
semantic-version==2.6.0
simplegeneric==0.8.1
simplejson==3.8.2
singledispatch==3.4.0.3
six==1.11.0
snowballstemmer==1.2.1
sortedcollections==0.5.3
sortedcontainers==1.5.7
Sphinx==1.6.5
sphinxcontrib-websupport==1.0.1
spyder==3.1.4
SQLAlchemy==1.1.9
statsmodels==0.8.0
subprocess32==3.2.7
sympy==1.1.1
tables==3.3.0
TBB==0.1
tblib==1.3.2
tensorflow==1.4.0
tensorflow-tensorboard==0.4.0rc3
terminado==0.6
testpath==0.3
toolz==0.8.0
tornado==4.3
traitlets==4.2.1
traits==4.6.0
traitsui==5.1.0
transducers==0.4.dev0
trimesh==2.13.11
typing==3.6.2
unicodecsv==0.14.1
urllib2-kerberos==0.1.6
```

```
urllib3==1.22
vcstools==0.1.39
virtualenv==15.0.1
wcwidth==0.1.7
Werkzeug==0.12.2
widgetsnbextension==1.2.6
wrapt==1.10.10
wstool==0.1.13
xlrd==1.0.0
XlsxWriter==0.9.6
xlwings==0.10.4
xlwt==1.2.0
zict==0.1.2

Warning: cannot find svn location for RBTools===0.5Amazon1.dev-r0
```

Here are the Jupyter magic commands we may use:

In [2]:
```
%lsmagic
```

Out[2]:
```
Available line magics:
%alias  %alias_magic  %autocall  %automagic  %autosave  %bookmark  %cat  %cd  %clear  %colors  %conf
ig  %connect_info  %cp  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui  %hist  %histor
y  %install_default_config  %install_ext  %install_profiles  %killbgscripts  %ldir  %less  %lf  %lk
%ll  %load  %load_ext  %loadpy  %logoff  %logon  %logstart  %logstate  %logstop  %ls  %lsmagic  %lx
%macro  %magic  %man  %matplotlib  %mkdir  %more  %mv  %notebook  %page  %pastebin  %pdb  %pdef  %pd
oc  %pfile  %pinfo  %pinfo2  %popd  %pprint  %precision  %profile  %prun  %psearch  %psource  %pushd
  %pwd  %pycat  %pylab  %qtconsole  %quickref  %recall  %rehashx  %reload_ext  %rep  %rerun  %reset
%reset_selective  %rm  %rmdir  %run  %save  %sc  %set_env  %store  %sx  %system  %tb  %time  %timeit
  %unalias  %unload_ext  %who  %who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%latex  %%perl  %%pru
n  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time  %%
timeit  %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

UNITS OF MEASURE

The `Pint` library has excellent support for units of measure (better than Mathematica's!). It is so good, it's almost sufficient reason to adopt Python. The support is integrated across `numpy`, `scipy`, and `sympy`.

In [3]:
```
%%bash
pip install pint -U
```

```
Requirement already up-to-date: pint in /anaconda/lib/python2.7/site-packages
```

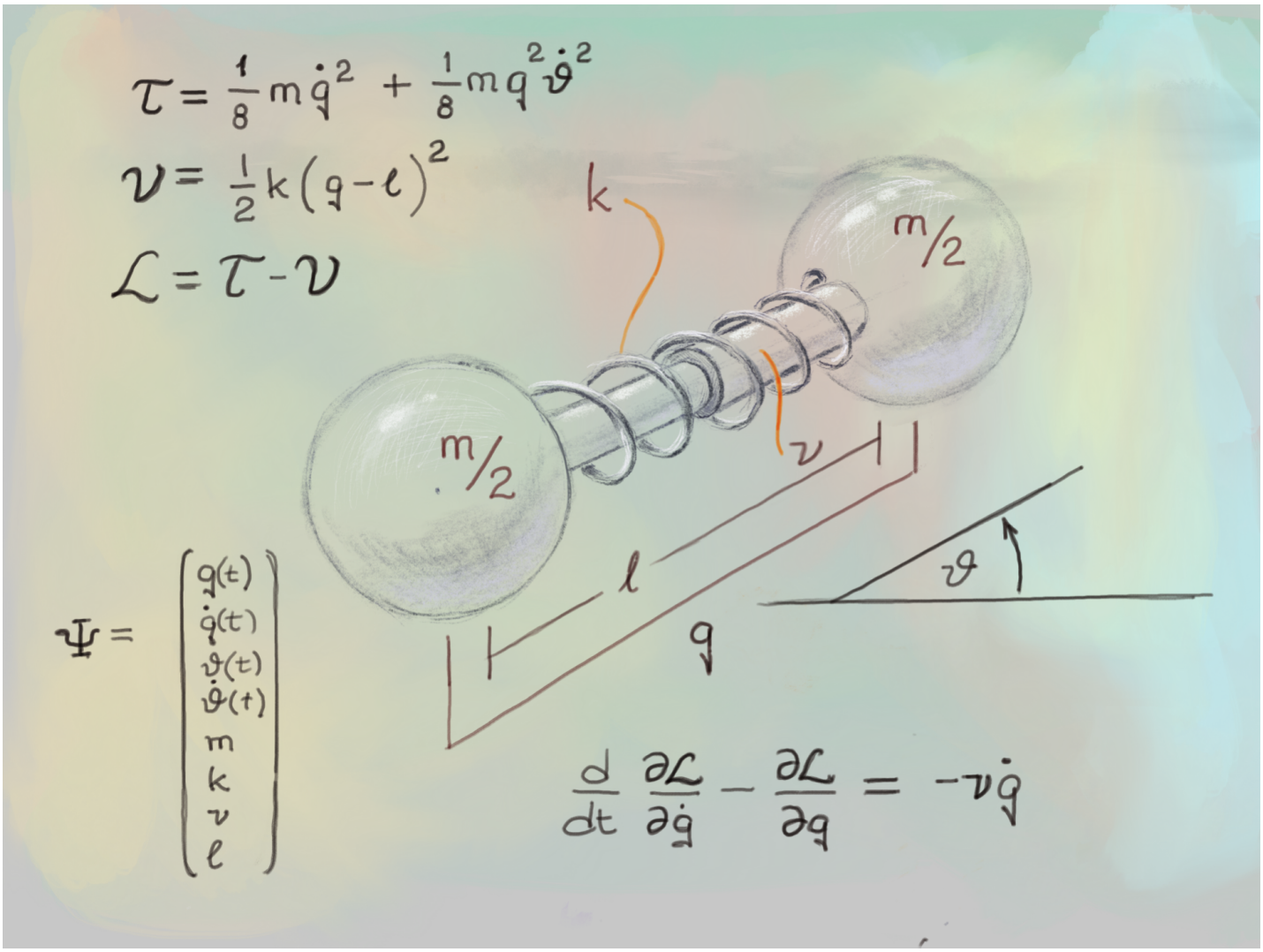
In [4]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import pint # for units of measure
units = pint.UnitRegistry()
```

In [5]:
```
units.default_system
```

Out[5]:
```
u'mks'
```

EQUATIONS OF MOTION

The following figure illustrates the system, an eight-dimensional state vector, the Lagrangian, and one of the two equations of motion in the familiar Euler-Lagrange symbolic form:



We want equations of motion in *first-order, state-space* form

$$\frac{dx}{dt}(t_0) = \text{func}\,(x, t_0, \ldots \langle \textit{optional arguments} \rangle \ldots)$$

`func` need not be linear, so this form is very general.

We get there by symbolic differentiation of the Lagrangian, then symbolic rearrangement to express d^2q/dt^2 and $d^2\theta/dt^2$ explicitly in terms of other quantities.

LAGRANGIAN FORMULATION

In [6]:
```
%%bash
pip install sympy -U
```

```
Requirement already up-to-date: sympy in /anaconda/lib/python2.7/site-packages
Requirement already up-to-date: mpmath>=0.19 in /anaconda/lib/python2.7/site-packages (from sympy)
```

The *Lagrangian* is kinetic energy minus potential energy.

The kinetic energy is often expressed in the center-of-mass frame, where it decomposes into pure linear motion and pure rotational motion. But kinetic energy must be measured in an inertial frame (many mistakes I see in published papers and books can be traced to misunderstanding subtleties of this fact.)

In [7]:
```
from sympy import Symbol, Function
from sympy.calculus.euler import euler_equations
from sympy import init_printing
init_printing()

q     = Function('q')
theta = Function('theta')
t     = Symbol('t')
m     = Symbol('m')
k     = Symbol('k')
nu    = Symbol('nu')
l     = Symbol('l')

qdot  = q(t).diff(t)
omega = theta(t).diff(t)

Tlin  = m * qdot**2 / 8           # Kinetic Energy of Linear Motion
Trot  = m * (q(t) * omega)**2 / 8 # Kinetic Energy of Rotational Motion
P     = - k*(q(t) - l)**2 / 2     # Potential Energy of Spring
L     = Tlin + Trot - P

Leqns = euler_equations(L, [q(t), theta(t)], t)
Leqns
```

Out[7]:
$$\left[\frac{k}{2}(-2l + 2q(t)) + \frac{m}{4}q(t)\left(\frac{d}{dt}\theta(t)\right)^2 - \frac{m}{4}\frac{d^2}{dt^2}q(t) = 0, \quad -\frac{m}{4}\left(q(t)\frac{d^2}{dt^2}\theta(t) + 2\frac{d}{dt}q(t)\frac{d}{dt}\theta(t)\right)q(t) = 0\right]$$

Solve algebraically for the required functions:

In [8]:
```
from sympy.solvers import solve
LeqnsExplicit = solve(Leqns, [q(t).diff(t, t), theta(t).diff(t, t)])
LeqnsExplicit
```

Out[8]:
$$\left\{\frac{d^2}{dt^2}q(t) : \frac{1}{m}\left(-4kl + 4kq(t) + mq(t)\left(\frac{d}{dt}\theta(t)\right)^2\right), \quad \frac{d^2}{dt^2}\theta(t) : -\frac{2\frac{d}{dt}q(t)}{q(t)}\frac{d}{dt}\theta(t)\right\}$$

STATE-SPACE FORM

State-space form expresses any system as a first-order system: using only first derivatives.

Explicitly add in the generalized damping force $-\nu\dot{q}$. Scipy did not furnish a convenient way to do that above.

In [9]:
```
def Dx(x, t):                       # unpack the elements of the state x
    q, qdot, theta, omega, m, k, nu, l = x
    return [qdot,                   # list of dy/dt=f functions
            4 * (k*l - k*q - nu*qdot) / m + q * omega * omega, # q double dot
            omega,                  # theta dot
            - 2 * qdot * omega / q, # theta double dot
            0, 0, 0, 0]             # derivatives of the constants m, k, nu, l
```

We can call this function symbolically as well as numerically.

In [10]:
```
Dx([q(t),qdot,theta(t),omega,m,k,nu,l], t)
```

Out[10]:

$$\left[\frac{d}{dt}q(t),\quad q(t)\left(\frac{d}{dt}\theta(t)\right)^2+\frac{1}{m}\left(4kl-4kq(t)-4\nu\frac{d}{dt}q(t)\right),\quad \frac{d}{dt}\theta(t),\quad -\frac{2\frac{d}{dt}q(t)}{q(t)}\frac{d}{dt}\theta(t),\quad 0,\quad 0,\quad 0,\quad 0\right]$$

Compare the results against Mathematica:

```
Dx[x_, t_] := {{0, 1, 0, 0, 0, 0, 0, 0},
               {-4 k/m, -4 ν/m, 0, 0, 0, 0, 0, 0},
               {0, 0, 0, 1, 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, 0, 0, 0, 0, 0, 0, 0}}.Ψ + {0, 4 k l/m + q θdot^2, 0, -(2 qdot θdot)/q, 0, 0, 0, 0} //. symRules[Ψ, x];

Dx[Ψ, τ] // MatrixForm
```

trixForm=

$$\begin{pmatrix} \text{qdot} \\ \frac{4\,k\,l}{m}-\frac{4\,k\,q}{m}+q\,\theta\text{dot}^2-\frac{4\,\text{qdot}\,\nu}{m} \\ \theta\text{dot} \\ -\frac{2\,\text{qdot}\,\theta\text{dot}}{q} \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

SAMPLE INTEGRATION

INITIAL CONDITIONS

Pick plausible initial conditions.

In [11]:
```
q0     = (1.0 * units.inch)                       .to_base_units().magnitude
qdot0  = (0.0 * units.inch / units.second)        .to_base_units().magnitude
theta0 = (0.0 * units.radian)                     .to_base_units().magnitude
omega0 = (1440.0 * units.arcdegree / units.second).to_base_units().magnitude

m0  = (10. * units.ounce)                         .to_base_units().magnitude
k0  = (0.0057101471547 * units.lbf / units.inch)  .to_base_units().magnitude
nu0 = (0.0003 * units.lbf * units.sec / units.inch).to_base_units().magnitude
l0  = (1.0 * units.inch)                          .to_base_units().magnitude

x0 = [q0, qdot0, theta0, omega0, m0, k0, nu0, l0]
print (x0)
```

```
[0.0254, 0.0, 0.0, 25.132741228718345, 0.28349523125000003, 0.9999999999942827, 0.0525380505739429,
0.0254]
```

TIME: INDEPENDENT VARIABLE

Track the system for 1.5 seconds at 1,000 observations per second.

In [12]:
```
tStop       = (1.5 * units.sec).magnitude
stepsPerSec = 1000.
h           = 1. / stepsPerSec
ts          = np.arange(0., tStop, h)
# print tStop, tInc, ts
```

CALL THE SOLVER

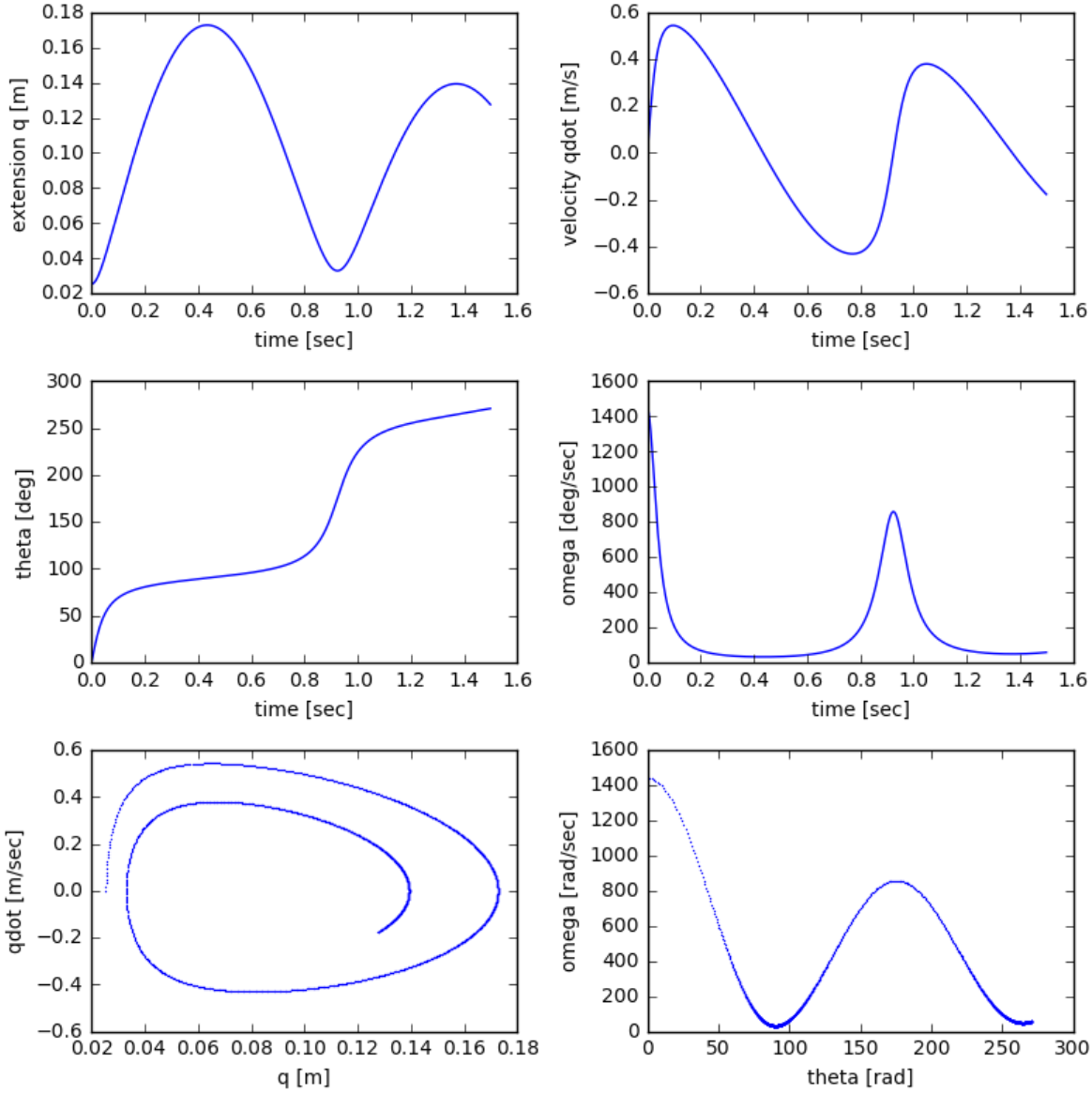
https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.integrate.odeint.html (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.integrate.odeint.html)

In [13]:
```
from scipy import integrate
psoln = integrate.odeint(Dx, x0, ts)
```

PLOT RESULTS

We are unable to break up the following input block for unknown reasons. The side effects on the figures, i.e., the internal state variables of *matplotlib*, do not propagate properly through the notebook.

In [14]:
```
fig = plt.figure(figsize=(8,8))

ax0 = fig.add_subplot(321)
ax0.plot(ts, psoln[:,0])
ax0.set_xlabel('time [sec]')
ax0.set_ylabel('extension q [m]')

ax01 = fig.add_subplot(322)
ax01.plot(ts, psoln[:,1])
ax01.set_xlabel('time [sec]')
ax01.set_ylabel('velocity qdot [m/s]')

ax1 = fig.add_subplot(323)
ax1.plot(ts, psoln[:,2]*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

ax2 = fig.add_subplot(324)
ax2.plot(ts, psoln[:,3]*180/np.pi)
ax2.set_xlabel('time [sec]')
ax2.set_ylabel('omega [deg/sec]')

ax03 = fig.add_subplot(325)
ax03.plot(psoln[:,0], psoln[:,1], '.', ms=1)
ax03.set_xlabel('q [m]')
ax03.set_ylabel('qdot [m/sec]')

ax3 = fig.add_subplot(326)
ax3.plot(psoln[:,2]*180/np.pi, psoln[:,3]*180/np.pi, '.', ms=1)
ax3.set_xlabel('theta [rad]')
ax3.set_ylabel('omega [rad/sec]')

plt.tight_layout()
plt.show()
```



Notice that $\omega \stackrel{\text{def}}{=} \dot{\theta}$, the spin rate, decreases and rebounds, as predicted intuitively.

INTEGRATION BY FOLD

Inside the EKF, we need to integrate the equations of motion. To eliminate the dependency on `scipy.ode`, a black-box integrator, let's write our own, *foldable* (explanation below), Runge-Kutta integrator and test it externally to the EKF.

Note that `numpy`'s integrator automatically handles stiff systems, where the integration rate is below the Nyquist rate of the system, $0.5\sqrt{k/ml}$. Our hand-written integrator is not that smart, but our system is not stiff, as we see below.

A *stiff* system has discretized time step larger than the natural period of the system. Stiff systems require special integrators.

In [15]:
```
def rk4step (Dx, t_x_tuple, dt_t_tuple):
    t_ignore, x = t_x_tuple
    dt, t       = dt_t_tuple
    dx1 = dt * np.array(Dx(x,           t         ))
    dx2 = dt * np.array(Dx(x + dx1 / 2, t + dt / 2))
    dx3 = dt * np.array(Dx(x + dx2 / 2, t + dt / 2))
    dx4 = dt * np.array(Dx(x + dx3,     t + dt    ))
    return (t + dt, x + (dx1 + 2 * dx2 + 2 * dx3 + dx4) / 6)
```

In [16]:
```
%%bash
pip install functools32 -U
```

```
Requirement already up-to-date: functools32 in /usr/local/lib/python2.7/site-packages
```

In [17]:
```
import functools as fn
```

In [18]:
```
duration = 1.5 # seconds
```

In [19]:
```
import toolz
from toolz import accumulate
```

Accumulate is Python's version of *fold* that keeps intermediate results.

In [20]:
```
soln = list (accumulate(fn.partial(rk4step, Dx), # specialized, foldable
                        [(h, h*i) for i in np.arange(int(duration//h))], # data
                        (0.0, x0))) # initial result
```

Notice the call of *partial*, which produces an `rk4` specialized for this `Dx`. The specialized integrator is foldable over the data.

FOLD IS FUNDAMENTAL

It is difficult to overstate the importance of foldable form. Once you learn to recognize it, you will see it thinly discuised all over the place. It goes by many names and variations, and has doubtless been "discovered" multiple times in various computer languages because it is fundamental. The following names are roughly synonymous.

- `fold`, foldleft, foldright
- `scan`, scanleft, scanright
- `reduce`, reductions
- `transduce`, transductions
- `aggregate`
- `accumulate`

Accumulate (or *fold*) updates state quantities, one observation at a time.

It's therefore ideal for all kinds of Markovian and Bayesian processes, where all information about a quantity is incremental, without entire histories. This property is essential for embedded applications, where we must work in constant memory. *Accumulate* (or *fold*) is also fundamental in the sense that most functional operators can be efficiently expressed as folds.

See Graham's article on the Universality of Fold www.cs.nott.ac.uk/~pszgmh/fold.pdf, https://dl.acm.org/citation.cfm?id=968579 (https://dl.acm.org/citation.cfm?id=968579).

Abstracting it as a mnemonic,

$$\langle \textit{sequence of results} \rangle = \textit{accumulate} \begin{pmatrix} \langle \textit{binary function} \rangle, \\ \langle \textit{sequence of data points} \rangle, \\ \langle \textit{initial value for result} \rangle \end{pmatrix}$$

The binary function takes two arguments: an instance of *result* type and an instance of *data* type. Any such binary function is foldable. We seek to write all data processing as foldable functions.

A *foldable function* takes a prior state (result) and an observation (input) and returns a new state (result).

In our example, the result type is a pair of a time and a state, and a state is an eight-dimensional vector:

$$\textit{result} \in \{\text{pairs of times and states}\} = \{(t, \boldsymbol{x}) \mid t \in \mathbb{R}, \boldsymbol{x} \in \mathbb{R}^8\}$$

Our initial value for the result, then, is a pair of the initial value for the time and the state, namely `(0.0, x0)`. So far, we have

$$\langle \textit{sequence of results} \rangle = \textit{accumulate} \begin{pmatrix} \langle \textit{binary function} \rangle, \\ \langle \textit{sequence of data points} \rangle, \\ (0.0, \boldsymbol{x}_0) = \langle \textit{initial value for result} \rangle \end{pmatrix}$$

Our data points are pairs of delta time and the time epoch.

$$\langle \textit{sequence of results} \rangle = \textit{accumulate} \begin{pmatrix} \langle \textit{binary function} \rangle, \\ [(h, 0.0), (h, h), (h, 2h), \ldots, , (h, nh)] = \langle \textit{sequence of data points} \rangle, \\ (0.0, \boldsymbol{x}_0) = \langle \textit{initial value for result} \rangle \end{pmatrix}$$

Our binary function is the integrator `rk4step` specialized on `Dx`. The specialized integrator takes in the current result, a new data point, and produces a new data point, satisfying the requirements for a foldable:

$$\langle \textit{sequence of results} \rangle = \textit{accumulate} \begin{pmatrix} \text{rk4step}\,(Dx\,((t, \boldsymbol{x}), (\delta t, t))) = \langle \textit{binary function} \rangle, \\ [(h, 0.0), (h, h), (h, 2h), \ldots, , (h, nh)] = \langle \textit{sequence of data points} \rangle, \\ (0.0, \boldsymbol{x}_0) = \langle \textit{initial value for result} \rangle \end{pmatrix}$$

In this case, the sequences are lists, but they could be asynchronous data streams (*Observables*; see https://github.com/ReactiveX/RxPY (https://github.com/ReactiveX/RxPY)) or lazy streams (*iterators*, *generators*; see https://pypi.python.org/pypi/lazy-streams/0.4 (https://pypi.python.org/pypi/lazy-streams/0.4)). Exactly the same binary function works on all mechanisms for data delivery. That fact allows us to test and debug the binary function independently of the source of the data and justifies the abstraction into a function.

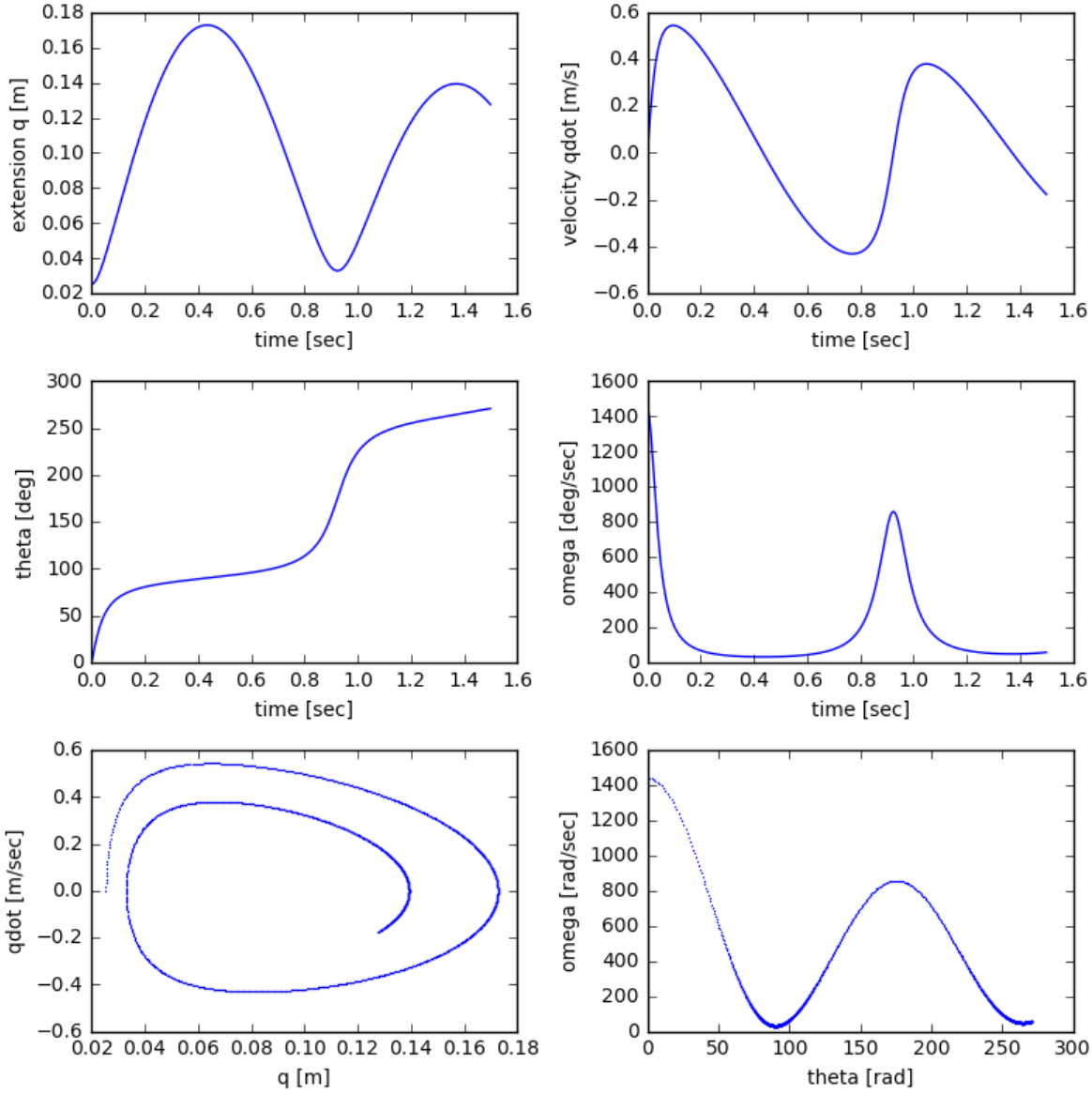
Accumulate abstracts data delivery, separating it from data processing.

We will see exactly this form for the EKF below.

VISUALIZING RESULTS OF RK4STEP

In [21]:
```
times  = np.array([s[0] for s in soln])
qs     = np.array([s[1][0] for s in soln])
qdots  = np.array([s[1][1] for s in soln])
thetas = np.array([s[1][2] for s in soln])
omegas = np.array([s[1][3] for s in soln])
```

In [22]:
```
fig = plt.figure(figsize=(8,8))

ax0 = fig.add_subplot(321)
ax0.plot(times, qs)
ax0.set_xlabel('time [sec]')
ax0.set_ylabel('extension q [m]')

ax01 = fig.add_subplot(322)
ax01.plot(times, qdots)
ax01.set_xlabel('time [sec]')
ax01.set_ylabel('velocity qdot [m/s]')

ax1 = fig.add_subplot(323)
ax1.plot(times, thetas*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

ax2 = fig.add_subplot(324)
ax2.plot(times, omegas*180/np.pi)
ax2.set_xlabel('time [sec]')
ax2.set_ylabel('omega [deg/sec]')

ax03 = fig.add_subplot(325)
ax03.plot(qs, qdots, '.', ms=1)
ax03.set_xlabel('q [m]')
ax03.set_ylabel('qdot [m/sec]')

ax3 = fig.add_subplot(326)
ax3.plot(thetas*180/np.pi, omegas*180/np.pi, '.', ms=1)
ax3.set_xlabel('theta [rad]')
ax3.set_ylabel('omega [rad/sec]')

plt.tight_layout()
plt.show()
```



The results are visually indistinguishable from those produced by numpy.

EKF

Given a model that predicts observations from states, the Kalman Filter gives states from observations.

Kalman optimally inverts the observation-from-states model. "Optimally" means that it minimizes the mean square error, and that's the connection to optimal control, which does the same thing. Kalman is least-squares regression for linear models with Gaussian noise. Extended Kalman Filtering handles non-linear systems by linearizing the observation model. There are many more variations on the theme.

FUNDAMENTAL MATRIX

The EKF requires linearized observation-model packaged in a *fundamental matrix* $\boldsymbol{F}[\boldsymbol{\Psi}, t]$ (we sometimes call the state vector $\boldsymbol{x}$ by the Greek capital letter $\boldsymbol{\Psi}$, and derivatives with overdots, as in $\dot{\boldsymbol{\Psi}}$).

If the exact dynamics are the first-order differential equation $\dot{\mathbf{\Psi}} = Dx[\mathbf{\Psi}, t]$, then the fundamental matrix factors out the state:

$$\dot{\mathbf{\Psi}} = \boldsymbol{F}[\mathbf{\Psi}, t] \cdot \mathbf{\Psi}$$

We could get sympy to find the fundamental matrix starting with Jacobians and *contraction* of *Indexed Objects* (http://docs.sympy.org/0.6.7/modules/matrices.html (http://docs.sympy.org/0.6.7/modules/matrices.html), http://docs.sympy.org/latest/modules/tensor/indexed.html (http://docs.sympy.org/latest/modules/tensor/indexed.html)). However this is easier to do in Mathematica. Just check it, here.

Start with a symbolic form for $\mathbf{\Psi}$:

In [23]:
```
psi = [q(t), qdot, theta(t), omega, m, k, nu, l]
psi
```

Out[23]:
$$\left[q(t), \quad \frac{d}{dt}q(t), \quad \theta(t), \quad \frac{d}{dt}\theta(t), \quad m, \quad k, \quad \nu, \quad l\right]$$

Evaluate `Dx` symbolically, as before, just using $\mathbf{\Psi}$ as shorthand:

In [24]:
```
from sympy import simplify, expand
psidot = Dx(psi, t)
map (expand, psidot)
```

Out[24]:
$$\left[\frac{d}{dt}q(t), \quad \frac{4k}{m}l - \frac{4k}{m}q(t) + q(t)\left(\frac{d}{dt}\theta(t)\right)^2 - \frac{4\nu}{m}\frac{d}{dt}q(t), \quad \frac{d}{dt}\theta(t), \quad -\frac{2\frac{d}{dt}q(t)}{q(t)}\frac{d}{dt}\theta(t), \quad 0, \quad 0, \quad 0, \quad 0\right]$$

$\boldsymbol{F}[\mathbf{\Psi}, t]$ must satisfy $Dx[\mathbf{\Psi}, t] = \boldsymbol{F}[\mathbf{\Psi}, t] \cdot \mathbf{\Psi}$. Using `numpy.dot` and the following definition:

In [25]:
```
def F(x, t):
    q, qdot, theta, thetadot, m, k, nu, l = x # unpack the state
    return np.array([
        [0, 1, 0, 0, 0, 0, 0, 0],
        [-4*k/m + thetadot * thetadot,
         -4*nu/m, 0, 0,
          4*(k*(q-l) + nu*qdot)/(m * m),
         -4*(q-l)/m, -4*qdot/m, 4*k/m],
        [0, 0, 0, 1, 0, 0, 0, 0],
        [ 2*qdot*thetadot/(q * q),
         -2*thetadot/q, 0,
         -2*qdot/q, 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],
    ])
```

find a match:

In [26]:
```
map (expand, ((F(psi, t).dot(psi))).tolist())
```

Out[26]:
$$\left[\frac{d}{dt}q(t), \quad \frac{4k}{m}l - \frac{4k}{m}q(t) + q(t)\left(\frac{d}{dt}\theta(t)\right)^2 - \frac{4\nu}{m}\frac{d}{dt}q(t), \quad \frac{d}{dt}\theta(t), \quad -\frac{2\frac{d}{dt}q(t)}{q(t)}\frac{d}{dt}\theta(t), \quad 0, \quad 0, \quad 0, \quad 0\right]$$

PROPAGATOR MATRIX

Further linearize in a Taylor expansion as $\mathbf{\Psi}(t + \delta t) \approx \mathbf{\Psi}(t) + \delta t \boldsymbol{F}[\mathbf{\Psi}, t] \cdot \mathbf{\Psi} = (\boldsymbol{I} + \delta t \boldsymbol{F}[\mathbf{\Psi}, t]) \cdot \mathbf{\Psi}(t)$. This is Euler integration with its hazards of bias accumulation.

$\boldsymbol{I} + \delta t \boldsymbol{F}[\boldsymbol{\Psi}, t] \stackrel{\text{def}}{=} \boldsymbol{\Phi}[\boldsymbol{F}, \boldsymbol{\Psi}, t, \delta t]$ is the *propagator matrix*

The *propagator* Euler-integrates linearized equations.

In [27]:
```
def Phi(F, x, t, dt):
    return np.identity(8) + F(x,t) * dt
```

PROCESS NOISE

The EKF almost always requires nominal noise added to the dynamics to compensate for inaccuracies of linearized dynamics. Without this *process noise*, the Kalman equations will "believe" the linearized dynamics exactly and use only observation covariance to calculate state covariance, usually leading to incorrect estimates or negative (impossible) covariances.

Deriving the process-noise matrix requires integrating assumed process-noise standard deviations across small times through the propagator matrix. It is the most tricky mathematics in the EKF. Note that assuming process-noise standard deviations is similar to assuming cost matrices in optimal control. In fact, Kalman filtering is mathematically *dual* to optimal control. These heuristic inputs specify the meaning of *optimality* in both cases.

This symbolic integration is much easier to do in Mathematica; we just check it here. The units of measure of each term is different and it is time-consuming to check them. Here, we trust Mathematica's symbolic integrator. Below is our manual transcription of Mathematica's result. For another time, we will go through the exercise of symbolically integrating in Python.

In [28]:
```
def Xi(sigmas, F, x, t, dt):
    sigmaqddot, sigmathetaddot = sigmas
    q, qdot, theta, thetadot, m, k, nu, l = x
    dt3 = dt ** 3
    dt2 = dt * dt
    td2 = thetadot * thetadot
    sq2 = sigmaqddot * sigmaqddot
    st2 = sigmathetaddot * sigmathetaddot
    tm1 = 1 - 4*dt*nu/m
    d = np.diag([
        dt3 * sq2,
        dt  * tm1*tm1 * sq2 / m,
        dt3 * st2,
        dt  * (4*dt2*td2*sq2 + st2*(q-2*dt*qdot)**2) / (q*q),
        0, 0, 0, 0
    ])
    ut = np.array([
        [0, dt2*sq2*tm1,
         0, -2*dt3*sq2*thetadot/q, 0,0,0,0],
        [0,0,0, -2*dt2*thetadot*tm1*sq2/q, 0,0,0,0],
        [0,0,0, dt2*st2*(1-2*qdot*dt/q), 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],
        [0,0,0,0, 0,0,0,0]
    ])
    return d + ut + np.transpose(ut)
```

In [29]:
```
sigmaqddot     = Symbol('\sigma_{\ddot{q}}')
sigmathetaddot = Symbol('\sigma_{\ddot{t}}') # doesn't work with \theta or theta
dt             = Symbol('\delta{t}')
```

Here is Mathematica's version:

```
Ξ[{σqddot_, σθddot_}][F_, x_, t_, δt_] :=
  With[{Eξ =
      {{0, 0, 0, 0, 0, 0, 0, 0},
       {0, σqddot^2, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, 0, 0, 0, 0, 0},
       {0, 0, 0, σθddot^2, 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}}},
    Integrate[Φ[F, x, τ, δt].Eξ.Φ[F, x, τ, δt]^T, {τ, 0, δt}]];
Ξ[{σq, σθ}][F, Ψ, τ, δt] // FullSimplify // MatrixForm
```

trixForm=

$$\begin{pmatrix}
\delta t^3\,\sigma_q^2 & \delta t^2\left(1-\frac{4\,\delta t\,\nu}{m}\right)\sigma_q^2 & 0 & -\frac{2\,\delta t^3\,\theta dot\,\sigma_q^2}{q} & 0&0&0&0\\
\delta t^2\left(1-\frac{4\,\delta t\,\nu}{m}\right)\sigma_q^2 & \frac{\delta t\,(m-4\,\delta t\,\nu)^2\,\sigma_q^2}{m^2} & 0 & -\frac{2\,\delta t^2\,\theta dot\left(1-\frac{4\,\delta t\,\nu}{m}\right)\sigma_q^2}{q} & 0&0&0&0\\
0 & 0 & \delta t^3\,\sigma_\theta^2 & \delta t^2\left(1-\frac{2\,qdot\,\delta t}{q}\right)\sigma_\theta^2 & 0&0&0&0\\
-\frac{2\,\delta t^3\,\theta dot\,\sigma_q^2}{q} & -\frac{2\,\delta t^2\,\theta dot\left(1-\frac{4\,\delta t\,\nu}{m}\right)\sigma_q^2}{q} & \delta t^2\left(1-\frac{2\,qdot\,\delta t}{q}\right)\sigma_\theta^2 & \frac{\delta t\left(4\,\delta t^2\,\theta dot^2\,\sigma_q^2+(q-2\,qdot\,\delta t)^2\,\sigma_\theta^2\right)}{q^2} & 0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0
\end{pmatrix}$$

Compare to `sympy`'s version:

In [30]:
```
map (lambda x: map (lambda y: y, x), Xi([sigmaqddot, sigmathetaddot], F, psi, t, dt))
```

Out[30]:

$$\left[\left[\delta t^3\sigma_{\ddot{q}}^2,\quad \delta t^2\sigma_{\ddot{q}}^2\left(-\frac{4\delta t}{m}\nu+1\right),\quad 0,\quad -\frac{2\delta t^3\sigma_{\ddot{q}}^2}{q(t)}\frac{d}{dt}\theta(t),\quad 0,\quad 0,\quad 0,\quad 0\right],\quad \left[\delta t^2\sigma_{\ddot{q}}^2\left(-\frac{4\delta t}{m}\nu+1\right),\quad \frac{\delta t\sigma_{\ddot{q}}^2}{m}\left(-\frac{4\delta}{m}\right.\right.$$

$$-\frac{2\delta t^2\sigma_{\ddot{q}}^2}{q(t)}\left(-\frac{4\delta t}{m}\nu+1\right)\frac{d}{dt}\theta(t),\quad 0,\quad 0,\quad 0,\quad 0\Big],\quad \left[0,\quad 0,\quad \delta t^3\sigma_{\dot{t}}^2,\quad \delta t^2\sigma_{\dot{t}}^2\left(-\frac{2\delta t\frac{d}{dt}q(t)}{q(t)}+1\right),\quad 0,\quad 0,\right.$$

$$\left[-\frac{2\delta t^3\sigma_{\ddot{q}}^2}{q(t)}\frac{d}{dt}\theta(t),\quad -\frac{2\delta t^2\sigma_{\ddot{q}}^2}{q(t)}\left(-\frac{4\delta t}{m}\nu+1\right)\frac{d}{dt}\theta(t),\quad \delta t^2\sigma_{\dot{t}}^2\left(-\frac{2\delta t\frac{d}{dt}q(t)}{q(t)}+1\right),\quad \frac{\delta t}{q^2(t)}\left(4\delta t^2\sigma_{\ddot{q}}^2\left(\frac{d}{dt}\theta(t)\right)^2+\sigma_{\dot{t}}^2\left(-2\delta\right.\right.\right.$$

$$0,\quad 0,\quad 0,\quad 0\Big],\quad [0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0],\quad [0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0]$$

$$[0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0],\quad [0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0,\quad 0]\Big]$$

DATA FUSION AND OBSERVATION MODEL

The ***state propagation model*** has been presented above as

$$\dot{\mathbf{\Psi}} = \boldsymbol{F}[\mathbf{\Psi}, t] \cdot \mathbf{\Psi}$$

The ***observation model*** is

$$z = \boldsymbol{A}[\mathbf{\Psi}, t] \cdot \mathbf{\Psi}$$

The matrix $\boldsymbol{A}$ contains the ***observation partials***. The original Kalman filter only works properly when the matrix does not depend directly on $\mathbf{\Psi}$. The EKF does not cover the more general case, but other variations of Kalman filtering do.

The observation model predicts observations from states.

When observing only the angle θ, the observation model is the following (nearly trivial) row vector in numpy.

In [31]:
```
At = np.array([[0, 0, 1, 0, 0, 0, 0, 0]])
```

We can also easily observe both extension q and angle θ via the following 2×8 matrix.

In [32]:
```
Aqt = np.array([[1,0,0,0,0,0,0,0],
                [0,0,1,0,0,0,0,0]])
```

This is a simple instance of *data fusion*. We could incorporate much more complicated observation models at this point. We also note that the observations need not be combined or *batched* in a vector. The filter works equally well in *scalar* mode, with different partials matrices on each call.

Data fusion is combining multiple observation models in one filter.

Mathematically, the EKF computes the least-squares regression over all the data. It just does so one observation at a time.

The Kalman filter incrementally computes $(\boldsymbol{A}^\intercal \boldsymbol{A})^{-1} \cdot \boldsymbol{A}^\intercal \cdot z$, the least-squares regression over all the data.

A-PRIORI COVARIANCE

As usual in Kalman filtering, the ***a-priori state covariance*** specifies elements of the state that we do not want to estimate from observations, the elements that we trust to dynamics. The dynamics couple to the so-called *unobserved states*, so observations will affect their estimates indirectly. In practice, the actual values in the a-priori do not matter much when there is a lot of data because the a-priori stands in for just one observation. It gets overwhelmed by real data very quickly, quadratically with the number of steps.

Set zero (or low) a-priori state covariances for elements of the state vector that should not be affected much by observations.

In passing, this trick allowed us at JPL to reverse standard spacecraft tracking to directly estimate the drift of the tectonic plates under the telescopes to a precision of millimeters per year!

In [33]:
```
def P_a_priori(sq, sqd, st, std, sm, sk, snu, sl):
    return np.diag([sq**2, sqd**2, st **2, std**2,
                    sm**2, sk **2, snu**2, sl**2])
```

The following a-priori covariance specifies that we should estimate q and θ and their velocities from the data, but nothing else.

In [34]:
```
P = P_a_priori(1,1,1,1,0,0,0,0)
```

SIMILARITY TRANSFORM

The similarity tranform of matrix $\boldsymbol{P}$ by matrix $\boldsymbol{A}$ is $\boldsymbol{A} \cdot \boldsymbol{P} \cdot \boldsymbol{A}^\intercal$. This allows us to scale the state covariance back to the units of observations, essential to Kalman's ***statistical inversion*** of the observation model.

In [35]:
```
def simt (ma, mp):
    return np.dot (ma, np.dot (mp, np.transpose (ma)))
```

OBSERVATION NOISE

In our data-fusion example, the observation noise is a 2×2 matrix. In the final, *scalar-mode* filter, it is a 1×1 matrix.

In [36]:
```
def Zeta(sz, x, t):
    sq, st = sz
    return np.array ([[sq**2, 0    ],
                      [0    , st**2]])
```

The following values are plausible, heuristic starts:

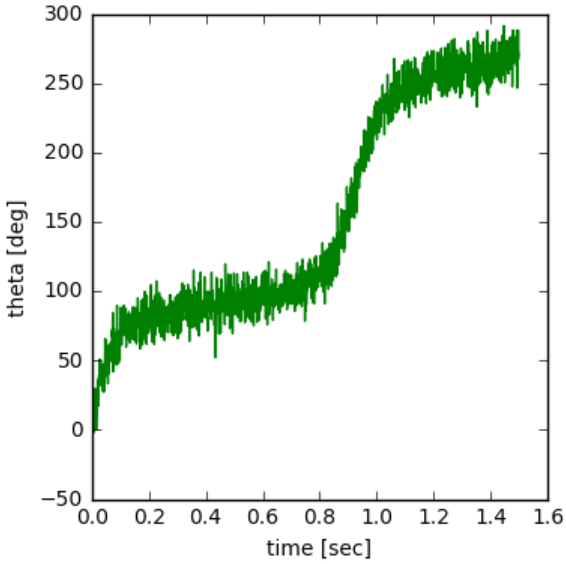
In [37]:
```
sigmaq     = (0.125 * units.inch).to_base_units().magnitude
sigmatheta = (10 * units.arcdegree).to_base_units().magnitude
```

SYNTHETIC DATA

To run the filter, we add artificial noise to the ground truth that we derived above in the integration section.

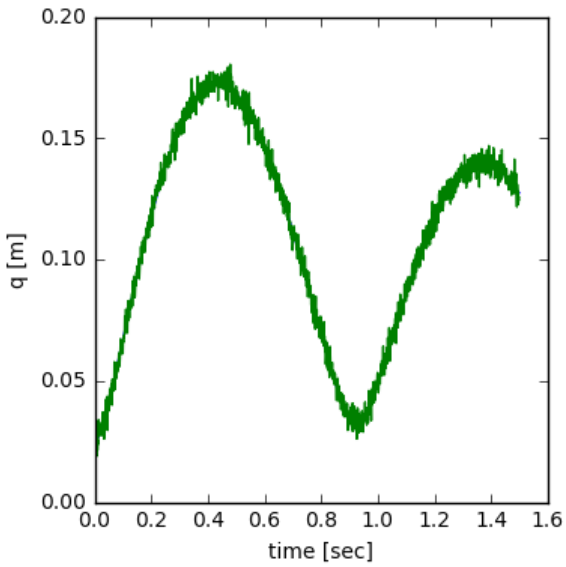
In [38]:
```
synthetas = thetas + np.random.normal(
                     0, sigmatheta, len(thetas))
```

In [39]:
```
fig = plt.figure(figsize=(4,4))

ax1 = fig.add_subplot(111)
ax1.plot(times, thetas   *180/np.pi)
ax1.plot(times, synthetas*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

plt.tight_layout()
plt.show()
```



In [40]:
```
synqs = qs + np.random.normal (0, sigmaq, len(qs))
```

In [41]:
```
fig = plt.figure(figsize=(4, 4))

ax1 = fig.add_subplot(111)
ax1.plot(times, qs)
ax1.plot(times, synqs)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('q [m]')

plt.tight_layout()
plt.show()
```



The EKF internally integrates the exact dynamics at a higher frequency $1/\texttt{idt}$ than the fundamental frequency of the filter data $1/\texttt{fdt}$ The integrator period `idt` must be less than `fdt` by a factor of 2 or more (probably a Nyquist phenomenon). In manual tuning, we found that an integration rate 32 times the filter rate worked well, producing unbiased estimates and tight covariances. Such manual tuning is usually necessary for EKFs and it is not difficult to understand why. Linearized Euler integration of the observation model accumulates biases proportional to `idt`. Smaller `idt` means smaller accumulating biases.

Set the integration rate higher than the filter rate in EKFs.

PARTIALLY EVALUATED EKF

Partially evaluate the EKF over all its parameters except the last two. The partially evaluated form is foldable over the observations, in a list, in a stream, asynchronous as a callback. Package the state, renamed x, and its covariance, P, in a tuple named ψ. Observations arrive to the partially evaluated filter in a tuple of time, t, observation partials, A, and actual observations, z. Remarkably, this very general EKF has only six "real" lines of code.

In [42]:
```
def EKF(Dx, F, Phi, Xi, Zeta, intg, fdt, idt, psi, taz):
    assert (fdt >= idt)
    x, P    = psi
    t, A, z = taz
    # "Predict" step: integrate exact dynamics.
    x2 = reduce (fn.partial(intg, Dx),
                 [(idt, t + idt*i) for i in np.arange(fdt/idt)],
                 (t, x))[1]
    # Propagate covariance by linearized dynamics (Euler integration).
    phi = Phi(F, x, t, fdt)
    P2  = Xi(F, x, t, fdt) + simt(phi, P)
    # "Gain" is roughly covar(state)/(covar(obsn) + covar(state)) ...
    #  with covar(state) scaled to units of obsn through similarity transform
    D   = Zeta(x, t) + simt(A, P2)
    # print {"D": D, "D^T": np.transpose(D)}
    # print {"A": A, "P2": P2, "A.P2": np.dot(A, P2)}
    #     The following is equivalent to K = P2.A^T.Inverse(D), which is
    #     the "schoolbook" definition, only it's almost always better to
    #     solve linear systems than to compute inverses:
    #     K == P2.A^T.Inverse(D)
    #     <=> K.D == P2.A^T
    #     <=> D^T.K^T == A.P2 (because P2 is symmetric)
    #     <=> K^T == Inverse[D^T].A.P2   --- whenever you see an inverse ...
    #     <=> K^T == linsolve(D^T, A.P2) --- on the left, think this instead
    # "Update" step: modify "predicted" covariance
    K   = np.transpose(np.linalg.solve(np.transpose(D), np.dot(A, P2)))
    # new covar(state) is roughly (1 - gain) * covar(predicted state)
    L   = np.identity(len(x)) - np.dot(K, A)
    # print {"K": K}
    # print {"x2": x2}
    # print {"residual": z - np.dot(A, x2)}
    # print {"correction": np.dot(K, z - np.dot(A, x2))}
    # print {"new x": x2 + np.dot(K, z - np.dot(A, x2))}
    # print {"new P": np.dot(L, P2)}
    # print {"new tuple": (x2 + np.dot(K, z - np.dot(A, x2)), np.dot(L, P2))}
    return (x2 + np.dot(K, z - np.dot(A, x2)), np.dot(L, P2))
```

Here it is with the comments stripped, to show the elegance and small size:

```
def EKF(Dx, F, Phi, Xi, Zeta, intg, fdt, idt, psi, taz): assert (fdt >= idt) x, P = psi t, A, z = taz x2 = reduce (fn.partial(intg, Dx), [(idt, t + idt*i) for i in np.arange(fdt/idt)], (t, x))[1] phi = Phi(F, x, t, fdt) P2 = Xi(F, x, t, fdt) + simt(phi, P) D = Zeta(x, t) + simt(A, P2) K = np.transpose(np.linalg.solve(np.transpose(D), np.dot(A, P2))) L = np.identity(len(x)) - np.dot(K, A) return (x2 + np.dot(K, z - np.dot(A, x2)), np.dot(L, P2))
```

The ***updated state*** is the ***predicted state*** x_2 plus a correction:

$$x \leftarrow x_2 + \langle correction \rangle$$

The ***correction*** is the ***gain*** K times a residual:

$$x \leftarrow x_2 + K \cdot \langle residual \rangle$$

The ***residual*** is the difference between the actual observation z and the predicted observation $A \cdot x_2$:

$$x \leftarrow x_2 + K \cdot (z - A \cdot x_2)$$

This beautiful form shows up over and over again in our work. We should strive to write all statistical recurrences in this form.

The ***gain*** K, schematically, is the predicted scaled state covariance $P_2 \cdot A^\intercal$ divided by the sum of scaled state covariance $A \cdot P_2 \cdot A^\intercal$ and observation covariance Z:

$$K \approx \frac{P_2 \cdot A^\intercal}{Z + A \cdot P_2 \cdot A^\intercal}$$

The gain K is inversely proportional to the observation covariance Z if the state covariance P_2 is small, yielding low credence to the observation residual. K is nearly scaled unity if the state covariance P_2 is large, giving high credence to the observation residual.

The updated state covariance is the predicted state covariance P_2 minus a gain-dependent correction:

$$P \leftarrow P_2 - K \cdot A \cdot P_2$$

The covariance should go down with each observation, but cannot, mathematically, be negative. Numerically negative covariances are a sign that the filter is diverging and must be tuned. There are many variations on this computation to avoid numerical issues. See http://vixra.org/abs/1607.0059 (http://vixra.org/abs/1607.0059), http://vixra.org/abs/1607.0084 (http://vixra.org/abs/1607.0084), http://vixra.org/abs/1606.0348 (http://vixra.org/abs/1606.0348)

RESULTS

STATE ESTIMATION (TRACKING)

Run the filter over the data-fusion scenario.

In [43]:
```
sqddot = 0.25
stddot = 0.25
sq     = sigmaq
st     = sigmatheta
zeta   = fn.partial(Zeta, (sq, st))
xi     = fn.partial(Xi, (sqddot, stddot))
```

Remind ourselves about the observation partials:

In [44]:
```
Aqt
```

Out[44]:
```
array([[1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0]])
```

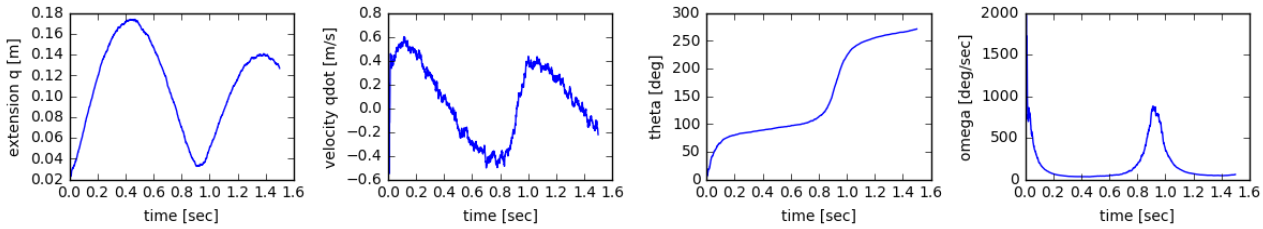
In [45]:
```
zs = [(i*h, Aqt, np.array([synqs[i], synthetas[i]])) for i in np.arange(int(duration//h))]
```

In [46]:
```
from functools import reduce
scenario = \
    accumulate(fn.partial(EKF, Dx, F, Phi, xi, zeta, rk4step, h, h/32),
               zs,
               (x0, P))
```

In [47]:
```
scenario_list = list(scenario)
```

In [48]:
```
def get_estimate(n):
    return np.array ([s[0][n] for s in scenario_list])
```

In [49]:
```
fig = plt.figure(figsize = (12,4))

ax0 = fig.add_subplot(241)
ax0.plot(times, get_estimate(0))
ax0.set_xlabel('time [sec]')
ax0.set_ylabel('extension q [m]')

ax01 = fig.add_subplot(242)
ax01.plot(times, get_estimate(1))
ax01.set_xlabel('time [sec]')
ax01.set_ylabel('velocity qdot [m/s]')

ax1 = fig.add_subplot(243)
ax1.plot(times, get_estimate(2)*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

ax2 = fig.add_subplot(244)
ax2.plot(times, get_estimate(3)*180/np.pi)
ax2.set_xlabel('time [sec]')
ax2.set_ylabel('omega [deg/sec]')

plt.tight_layout()
plt.show()
```



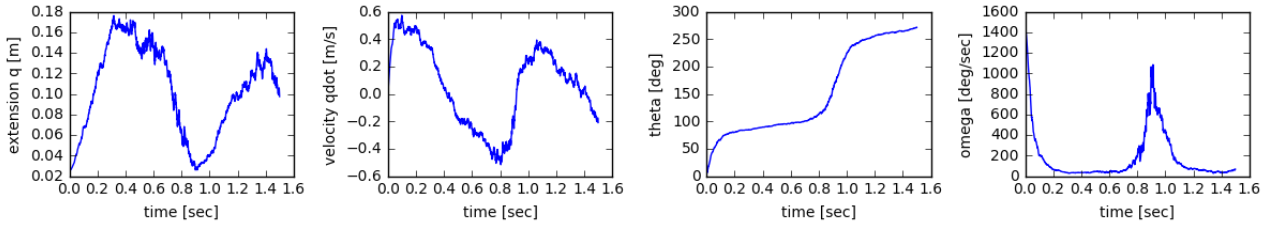
The filter does an excellent job of tracking all four states. This is not surprising as we directly observe both the extension q and the angle θ.

Now observe only the angle θ:

In [50]:
```
scenario = \
    accumulate(fn.partial(EKF, Dx, F, Phi, xi,
                          lambda x, t: [[st**2]],
                          rk4step, h, h/32),
               [(i*h, At, np.array([synthetas[i]])) for i in np.arange(int(duration//h))],
               (x0, np.diag([0, 0, 1, 1, 0, 0, 0, 0])))
```

In [51]:
```
scenario_list = list(scenario)
```

In [52]:
```
fig = plt.figure(figsize = (12,4))

ax0 = fig.add_subplot(241)
ax0.plot(times, get_estimate(0))
ax0.set_xlabel('time [sec]')
ax0.set_ylabel('extension q [m]')

ax01 = fig.add_subplot(242)
ax01.plot(times, get_estimate(1))
ax01.set_xlabel('time [sec]')
ax01.set_ylabel('velocity qdot [m/s]')

ax1 = fig.add_subplot(243)
ax1.plot(times, get_estimate(2)*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

ax2 = fig.add_subplot(244)
ax2.plot(times, get_estimate(3)*180/np.pi)
ax2.set_xlabel('time [sec]')
ax2.set_ylabel('omega [deg/sec]')

plt.tight_layout()
plt.show()
```

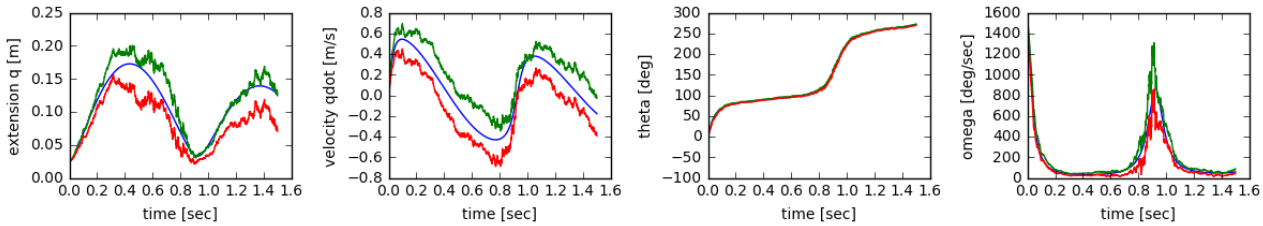


Surprisingly, the filter does a great job of tracking the extension q.

We now plot the 1-sigma variance envelopes of each state around the ground truth.

In [53]:
```
def get_variance(n):
    return np.array ([np.sqrt(s[1][n,n]) for s in scenario_list])
```

In [54]:
```
fig = plt.figure(figsize = (12,4))

ax0 = fig.add_subplot(241)
ax0.plot(times, qs)
ax0.plot(times, get_estimate(0) + get_variance(0))
ax0.plot(times, get_estimate(0) - get_variance(0))
ax0.set_xlabel('time [sec]')
ax0.set_ylabel('extension q [m]')

ax01 = fig.add_subplot(242)
ax01.plot(times, qdots)
ax01.plot(times, get_estimate(1) + get_variance(1))
ax01.plot(times, get_estimate(1) - get_variance(1))
ax01.set_xlabel('time [sec]')
ax01.set_ylabel('velocity qdot [m/s]')

ax1 = fig.add_subplot(243)
ax1.plot(times, thetas*180/np.pi)
ax1.plot(times, get_estimate(2)*180/np.pi + get_variance(2)*180/np.pi)
ax1.plot(times, get_estimate(2)*180/np.pi - get_variance(2)*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

ax2 = fig.add_subplot(244)
ax2.plot(times, omegas*180/np.pi)
ax2.plot(times, get_estimate(3)*180/np.pi + get_variance(3)*180/np.pi)
ax2.plot(times, get_estimate(3)*180/np.pi - get_variance(3)*180/np.pi)
ax2.set_xlabel('time [sec]')
ax2.set_ylabel('omega [deg/sec]')

plt.tight_layout()
plt.show()
```



Note the small bias in the estimate of q. It might be due to linearization or Euler integration, implicit in the linearized update step. In a more serious application, we would *be obligated* to diagnose it.

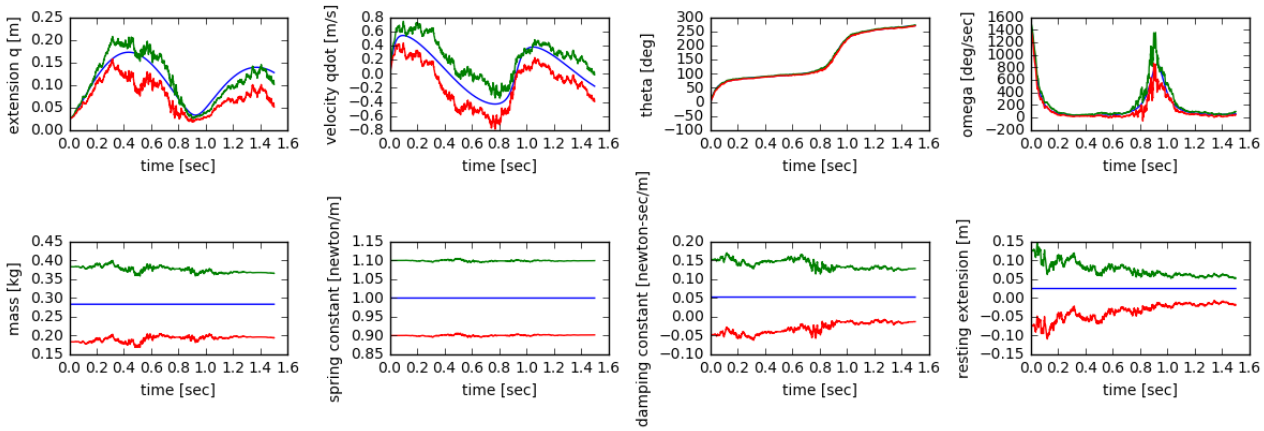
PARAMETER ESTIMATION (SYSTEM IDENTIFICATION)

Estimate the parameters m, k, ν, and l. Begin with small a-priori covariances, saying that we believe the dynamics more than the observations.

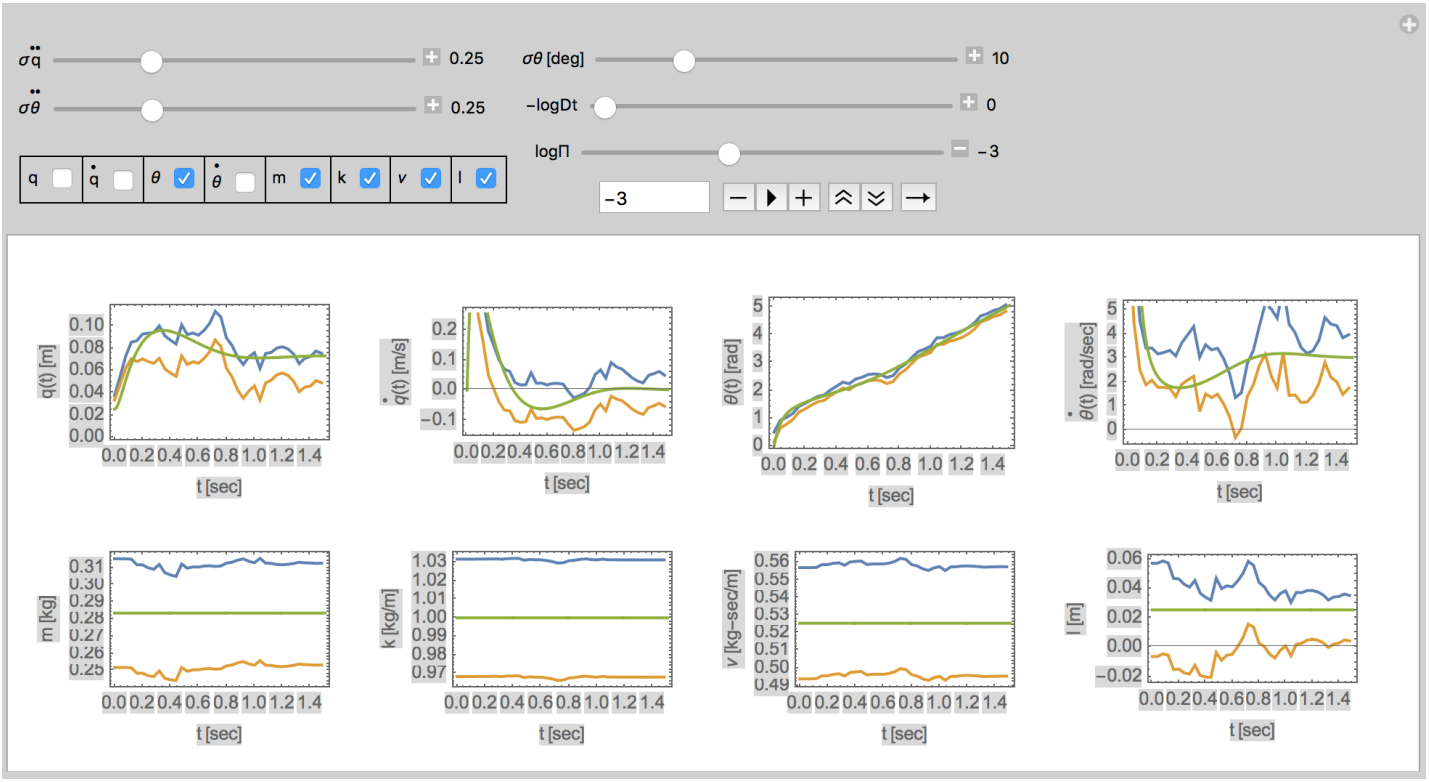
In [55]:
```
scenario = \
    accumulate(fn.partial(EKF, Dx, F, Phi, xi,
                          lambda x, t: [[st**2]],
                          rk4step, h, h/32),
               [(i*h, At, np.array([synthetas[i]])) for i in np.arange(int(duration//h))],
               (x0, np.diag([0, 0, 1, 1, 0.01, 0.01, 0.01, 0.01])))
```

In [56]:
```
scenario_list = list(scenario)
```

In [57]:
```
fig = plt.figure(figsize = (12,4))

ax0 = fig.add_subplot(241)
ax0.plot(times, qs)
ax0.plot(times, get_estimate(0) + get_variance(0))
ax0.plot(times, get_estimate(0) - get_variance(0))
ax0.set_xlabel('time [sec]')
ax0.set_ylabel('extension q [m]')

ax01 = fig.add_subplot(242)
ax01.plot(times, qdots)
ax01.plot(times, get_estimate(1) + get_variance(1))
ax01.plot(times, get_estimate(1) - get_variance(1))
ax01.set_xlabel('time [sec]')
ax01.set_ylabel('velocity qdot [m/s]')

ax1 = fig.add_subplot(243)
ax1.plot(times, thetas*180/np.pi)
ax1.plot(times, get_estimate(2)*180/np.pi + get_variance(2)*180/np.pi)
ax1.plot(times, get_estimate(2)*180/np.pi - get_variance(2)*180/np.pi)
ax1.set_xlabel('time [sec]')
ax1.set_ylabel('theta [deg]')

ax2 = fig.add_subplot(244)
ax2.plot(times, omegas*180/np.pi)
ax2.plot(times, get_estimate(3)*180/np.pi + get_variance(3)*180/np.pi)
ax2.plot(times, get_estimate(3)*180/np.pi - get_variance(3)*180/np.pi)
ax2.set_xlabel('time [sec]')
ax2.set_ylabel('omega [deg/sec]')

ax4 = fig.add_subplot(245)
ax4.plot(times, [x0[4] for t in times])
ax4.plot(times, get_estimate(4) + get_variance(4))
ax4.plot(times, get_estimate(4) - get_variance(4))
ax4.set_xlabel('time [sec]')
ax4.set_ylabel('mass [kg]')

ax5 = fig.add_subplot(246)
ax5.plot(times, [x0[5] for t in times])
ax5.plot(times, get_estimate(5) + get_variance(5))
ax5.plot(times, get_estimate(5) - get_variance(5))
ax5.set_xlabel('time [sec]')
ax5.set_ylabel('spring constant [newton/m]')

ax6 = fig.add_subplot(247)
ax6.plot(times, [x0[6] for t in times])
ax6.plot(times, get_estimate(6) + get_variance(6))
ax6.plot(times, get_estimate(6) - get_variance(6))
ax6.set_xlabel('time [sec]')
ax6.set_ylabel('damping constant [newton-sec/m]')

ax7 = fig.add_subplot(248)
ax7.plot(times, [x0[7] for t in times])
ax7.plot(times, get_estimate(7) + get_variance(7))
ax7.plot(times, get_estimate(7) - get_variance(7))
ax7.set_xlabel('time [sec]')
ax7.set_ylabel('resting extension [m]')

plt.tight_layout()
plt.show()
```



This compares qualitatively well against the results from Mathematica.



Again, notice that we have introduced bias into the estimate of extension q. This is not unreasonable given the extremely weak coupling of observations to the constants m, k, ν, and l. However, we can declare success: we have used Kalman filtering for system identification in this model.

CONCLUSION

Using a very general and testable, *i.e.,* foldable EKF and foldable integrators, we estimated four dynamical states and four parameters given observations only of the angle of a dashpot in spinning free fall. It is remarkable that weak observations can yield system identification. The model can be easily extended to three dimensions and to more complex models.