```
Class DynamicalSystem( object ):
    """
    Base class for the numerical integration of a dynamical system.
    """
    def __init__( self, sv=None, derivs=None, tstart=0, hstart=0.01,
                  stepper='RK4', output=None, stepmin=STEPMIN,
stepmax=STEPMAX,
                  abstol=ABSTOL, reltol=RELTOL ):
        self.steppers = { 'RK4':RK4Stepper, 'RK45':RK45Stepper, 'RK5':RK5Stepper,
                          'RK8':RK8Stepper }
        self.init( sv, derivs, tstart, hstart, stepper, output, stepmin,   stepmax,
                   abstol, reltol )

    def init( self, sv, derivs, tstart=0, hstart=0.01, stepper='RK4', output=None,
              stepmin=STEPMIN, stepmax=STEPMAX, abstol=ABSTOL,
reltol=RELTOL ):
        """
        Initialize the dynamical system.

        The input state vector has its coordinate origin at the position of m1.
        """
        self.logger = mlogging.Log( name='dynamics.DynamicalSystem',
                                    millisec=False, console=False )
        if not isinstance( sv, StateVector ):
            self.logger.error( "<sv> must be a StateVector." )
            return
        if not stepper in self.steppers.keys():
            self.logger.error( "<stepper> must be one of %s." % self.steppers.keys() )
            return
        if not isinstance( derivs, SVDerivs ):
            self.logger.error( "<derivs> must be a SVDerivs." )
            return
        self.s = sv.copy()
        self.t = tstart
        self.h = hstart
        self.hnext = hstart
        self.stepper = self.steppers[stepper.upper()]( derivs, stepmin, stepmax,
                                                       abstol, reltol )
        self.steps     = 0      # Number of successful steps.
        self.failed    = 0      # Number of failed steps.
        self.output    = self.prints if output is None else output
        self.outfile = None # Output file descriptor. Optionally set with run().
        self.outfilename = None
        self.hmin      = stepmin
        self.hmax      = stepmax
        self.abstol    = abstol
```

```
        self.reltol   = reltol

    def prints( self, t, sv   ):
        """
        Basic printing of t, rmag, vmag, and state vector components.
        """
        svlist = svunpack( sv )
        print "%.5f" % t
        for s in svlist:
            s = " %12.8f "*8   % ( (s.rmag(), s.vmag()) + tuple(s) )
            print s
            if self.outfile:
                self.outfile.write( ("%7.2f " % t) + s + '\n' )

    def pos( self ):
        """
        Return the current position vector.
        """
        return self.s[:self.n]

    def vel( self ):
        """
        Return the current velocity vector.
        """
        return self.s[self.n:]

    def step( self, htry=None ):
        """
        Advance the dynamical system by one step in the independent variable.
        """
        if self.s is None:
            self.logger.error( "You forgot to initialize!" )
            return
        if htry is not None:
            self.hnext = htry
        self.t, self.s, self.h, self.hnext = self.stepper.step( self.t, self.s,
                                                                 self.hnext )
        self.steps   = self.stepper.steps
        self.failed = self.stepper.failed

    def run( self, tout, tstop, outfile=None ):
        """
        Evolve the dynamical system, with output at intervals <tout>, until the
        independent variable reaches <tstop>. If <outfile> is specified, a file
        named <outfile> for output. It is assumed output() will handle the output
        formatting at each integration output point.
```

```
"""
if (self.h > 0 and tstop <= self.t) or (self.h < 0 and tstop >= self.t):
    self.logger.error( "<tstop> = %.3f does not advance the system." % tstop )
    self.logger.debug( "Current t = %.3f" % self.t )
    return
if outfile:
    self.outfilename = outfile
    try:
        self.outfile = open( outfile, 'w' )
        self.logger.info( "%s opened for output." % outfile )
    except Exception:
        errmsg = "\n%s" % traceback.format_exc()
        self.logger.error( errmsg )
        self.logger.error( "Unable to open output file %s" % outfile )
        return
    prm = ( stat.S_IRUSR | stat.S_IWUSR | stat.S_IRGRP | stat.S_IWGRP |
            stat.S_IROTH | stat.S_IWOTH )
    os.chmod( outfile, prm )
t0 = time.clock()
tlast = self.t   # previous time of output
havg = 0
self.stepper.steps   = 0
self.stepper.failed = 0
self.output( self.t, self.s )
while True:
    if self.hnext > 0:
        c1 = 'self.t + self.hnext >= tstop'
        c2 = 'self.t + self.hnext >= tlast + tout'
        e   = 'tlast += tout'
    else:
        c1 = 'self.t + self.hnext <= tstop'
        c2 = 'self.t + self.hnext <= tlast - tout'
        e   = 'tlast -= tout'
    if eval(c1):
        hnext = tstop - self.t
        self.step(hnext)
        break
    if eval(c2):
        exec(e)
        hsave = self.hnext
        hnext = tlast - self.t
        self.step(hnext)
        self.output( self.t, self.s )
        self.hnext = hsave
    self.step()
    havg += self.h
```

```
            self.output( self.t, self.s )
            self.logger.info( "cpu = %.1f, havg = %.2e, steps = %i, failed = %i"
                              % ( time.clock() - t0, havg/self.steps, self.steps,
                                  self.failed ) )
            if outfile:
                self.outfile.close()
                self.logger.info( "Data is in %s." % outfile )




class SolarSystem( DynamicalSystem ):
    """
import mlogging; log = mlogging.Log(console=True)
import dynamics, astorb, mplot
import math as mt
db = astorb.AsteroidDB( '/Users/nikitabogdanov/Programming/astorb.dat', 100 )
asteroids = [ db[k].statevector() for k in range(10) ]
s = dynamics.SolarSystem( [0,0,1,1,1,0,0,0], [asteroids[0]], interactions='all',
abstol=1e-6, jd=db[0].epoch )
dt = 1/mt.exp(1)/mt.sqrt(2)
s.run( dt, 5000, 'EMJ-Ceres-5000.dat' )
s = dynamics.SolarSystem( [0,0,1,1,1,0,0,0], interactions='all', abstol=1e-6,
jd=db[0].epoch )
s.run( dt, 5000, 'EMJ-5000.dat' )
s = dynamics.SolarSystem( [0,1,1,1,1,0,0,0], interactions='all', abstol=1e-6,
jd=db[0].epoch )
s.run( dt, 5000, 'VEMJ-5000.dat' )
s = dynamics.SolarSystem( [0,0,1,1,1,0,0,0], [asteroids[0]], interactions='all',
abstol=1e-6, jd=db[0].epoch )
s.run( dt, 10000, 'EMJ-Ceres-10000.dat' )
s = dynamics.SolarSystem( [0,0,1,1,1,0,0,0], [], interactions='all', abstol=1e-6,
jd=db[0].epoch )
s.run( dt, 10000, 'EMJ-10000.dat' )
s = dynamics.SolarSystem( [0,0,1,1,1,1,0,0], [], interactions='all', abstol=1e-6,
jd=db[0].epoch )
s.run( dt, 10000, 'EMJS-10000.dat' )
s = dynamics.SolarSystem( [0,0,1,1,1,1,0,0], [asteroids[0]], interactions='all',
abstol=1e-6, jd=db[0].epoch )
s.run( dt, 10000, 'EMJS-Ceres-10000.dat' )
    """
    def __init__( self, planets=[0,1,1,1,1,1,0,0], others=[], interactions='partial',
                  tstart=0, hstart=0.05, stepper='RK8', output=None,
stepmin=STEPMIN,
```

```
                    stepmax=STEPMAX, abstol=ABSTOL, reltol=RELTOL, jd=2451544.5
):
        """
        A heliocentric solar system DynamicalSystem.

        Turn on/off the various planets via <planets>. OrbitalStateVectors of
        other objects are in the list <others>. Initial values of the planet
        state vectors are derived from their orbutal elements at Julian Date <jd>.

        <interactions> may take these string values:
          'all'         interactions are calculated between all bodies (full N-body
problem)
          'partial'     no interactions between minor bodies
          'restricted'  minor bodies have no effect on planets (restricted N-body
problem)

        Reference orbit is Earth+Moon.
        """
        self.logger = mlogging.Log( name='dynamics.SolarSystem', millisec=False,
                                    console=False )
        if not isinstance( others, list ):
            self.logger.error( "<others> must be a list of OrbitalStateVectors." )
            return
        if not isinstance( interactions, str ) or interactions not in ['all', 'partial', 'restricted']:
            self.logger.error( "<interactions> must be one of 'all', 'partial', or 'restricted'."
)
            return
        # Convert time in Earth years to integration scaled time (equivalent to
        # circular reference orbit true anomaly).
        tstart   = 2*math.pi*tstart
        hstart   = 2*math.pi*hstart
        stepmin = 2*math.pi*stepmin
        stepmax = 2*math.pi*stepmax
        # Here is where we set the reference orbit body.
        self.mscaling = 1.0 + mconst['MEM']
        # Hardwire the solar system planet masses and mass scalings. Use a
        # circular 1 AU orbit as the scaling reference. Thus, time is in Earth
        # years, distances are in AU, and velocities are in 2*pi*AU/yr.
        self.pmasses = [ mconst['MMercury'], mconst['MVenus'], mconst['MEM'],
                         mconst['MMars'], mconst['MJupiter' ], mconst['MSaturn'],
                         mconst['MUranus'], mconst['MNeptune'] ]
        self.pmscales = [ (1+mconst['MMercury'])/self.mscaling,
                          (1+mconst['MVenus'])/self.mscaling,
                          (1+mconst['MEM'])/self.mscaling,
                          (1+mconst['MMars'])/self.mscaling,
                          (1+mconst['MJupiter'])/self.mscaling,
```

```
                                (1+mconst['MSaturn'])/self.mscaling,
                                (1+mconst['MUranus'])/self.mscaling,
                                (1+mconst['MNeptune'])/self.mscaling ]
        self.pnames = [ 'Mercury', 'Venus', 'Earth+Moon', 'Mars', 'Jupiter',
                        'Saturn', 'Uranus', 'Neptune']
        self.masses   = []
        self.mscales = []
        self.names    = []
        self.planets = planets
        # Create the requested planets.
        S = []
        plnts = [ mastro.Mercury, mastro.Venus, mastro.EarthMoon, mastro.Mars,
                  mastro.Jupiter, mastro.Saturn, mastro.Uranus, mastro.Neptune ]
        for k, p in enumerate(planets):
            if p:
                P = plnts[k]()
                elmnts = P.elements( jd=jd )      # Angles in degrees.
                # Convert mean motion to true anomaly.
                elmnts[-1] = keplert( elmnts[-1], elmnts[1] )
                s = elements_to_statevector( elmnts, self.pnames[k], self.pmasses[k],
                                             self.pmscales[k] )
                self.names.append( self.pnames[k] )
                self.masses.append( self.pmasses[k] )
                self.mscales.append( self.pmscales[k] )
                S.append( s )
        # Add the non-planetary bodies.
        if others:
            self.names    += [ s.name for s in others ]
            self.masses   += [ s.mass for s in others ]
            self.mscales += [ s.mscale for s in others ]
            for k, s in enumerate(others):
                S.append( s )
        # Create the solar system as a DynamicalSystem.
        s = svpack(S)
        nplanets = sum(planets)
        derivs = SShelioDerivs( self.masses, self.mscaling, interactions, nplanets )
        DynamicalSystem.__init__( self, s, derivs, tstart, hstart, stepper,
                                  output, stepmin, stepmax, abstol, reltol )

    def prints( self, t, sv ):
        """
        Basic printing of t and body state vector components.
        """
        eps = sys.float_info.epsilon
        Nnames = max( [ len(s) for s in self.names ] )
        fs = "%%%is " % Nnames
```

```
            if t == 0:
                print "\nt = %7.3f" % self.time()
                print " "*Nnames + "     r              v              a              e              i
(deg)     argperi      node         trueanom"
            else:
                print "\n%7.3f" % self.time()
            S = svunpack( sv, classtype=OrbitalStateVector )
            sout = ""
            for n, s in enumerate(S):
                mscale = (1 + self.masses[n])/self.mscaling
                a, e, i, w, W, f = statevector_to_elements( s, mscale )
                r = a*(1-e*e)/( 1 + e*math.cos( mastro.torad(f) ) )
                if r < eps or a < eps:
                    v = 0
                else:
                    v = math.sqrt( mscale*(2/r - 1/a) )
                i, w, W, f = map( lambda x: mastro.unwrap(x,360), (i, w, W, f) )
                print ( (fs + "%12.8f "*3 + " %10.8f" + " %10.6f"*4)
                        % (self.names[n], r, v, a, e, i, w, W, f) )
                if self.outfile:
                    sout += ("%.16e "*6 + " ") % (a, e, i, w, W, f)
            if self.outfile:
                self.outfile.write( ("%.16e   " % self.time()) + sout + '\n' )

        def run( self, tout, tstop, outfile=None ):
            """
            Evolve the dynamical system, with output at intervals <tout>, until the
            independent variable reaches <tstop>.
            """
            tout   = 2*math.pi*tout
            tstop = 2*math.pi*tstop
            DynamicalSystem.run( self, tout, tstop, outfile )

        def time( self ):
            """
            Current time, in units of reference orbit period.
            """
            return 0.5*self.t/math.pi

        def pos( self, n ):
            """
            Return the current position vector of body n.
            """
            return self.s[3*n:3*(n+1)].copy()

        def vel( self, n ):
```

```
        """
        Return the current velocity vector of body n.
        """
        N2 = int( len(self.s)/2 )
        v   = self.s[N2+3*n:N2+3*(n+1)].copy()
        return v




class ODEStepper( object ):
    """
    Base class for advancing a system of ODEs by one step.

    derivs   A SVDerivs object.
    """
    def __init__( self, derivs, stepmin=STEPMIN, stepmax=STEPMAX,
abstol=ABSTOL,
                  reltol=RELTOL ):
        self.logger = mlogging.Log( name='dynamics.ODEStepper' )
        self.D = None
        if not isinstance( derivs, SVDerivs ):
            self.logger.error( "<derivs> must be a SVDerivs." )
            return
        self.D = derivs
        self.hmin    = stepmin
        self.hmax    = stepmax
        self.abstol = abstol
        self.reltol = reltol
        self.errvec = None
        self.steps   = 0       # Number of successful steps.
        self.failed = 0       # Number of failed steps.

    def step( self, t, s, htry ):
        """
        Take a step.

        Returns the new time, the new state vector, the step taken, and the
        predicted next step size.
        """
        success = True
        while True:
            new_t, new_s = self._step( t, s, htry )
            err = self._errcalc( s, new_s )
```

```
            success, hnext = self._check_step( err, htry, success )
            if success:
                self.steps += 1
                return new_t, new_s, htry, hnext
            else:
                htry = hnext
                self.failed += 1
                if abs(htry) < self.hmin:
                    raise FloatingPointError( "Step size underflow!" )

    def _step( self, t, s, h ):
        """
        Take one ODE system step in the independent variable. Returns a 2-tuple
        consisting of the new value of the independent variable and the new
        state vector.

        Must also calculate an error vector.

        arguments:
          t             Independent variable (a scalar).
          s             A StateVector corresponding to <t>.
          h             Size of step to take.
        """
        raise NotImplementedError( "You must subclass ODEStepper." )

    def _errcalc( self, old_s, new_s ):
        """
        Calculate error measure for the current step.
        """
        N = len( old_s )
        err = 0.0
        for k in range(N):
            sk = self.abstol + self.reltol*max( abs(old_s[k]), abs(new_s[k]) )
            err += ( self.errvec[k]/sk )**2
        return math.sqrt(err/N)

    def _check_step( self, err, htry, good ):
        """
        Check whether or not the step was successful.

        Returns success (or not) and the predicted next step size.
        """
        alpha = -0.2
        safe   =   0.9
        minscale =   0.2
        maxscale = 10.0
```

```
            if err <= 1.0:
                if err == 0:
                    scale = maxscale
                else:
                    scale = safe*err**alpha
                    if scale < minscale:
                        scale = minscale
                    if scale > maxscale:
                        scale = maxscale
                if good:
                    # Previous attempt was good.
                    hnext = htry*scale
                else:
                    # Previous attempt was bad.
                    hnext = htry*min( scale, self.hmax )
                isgood = True
            else:
                hnext = htry*max( minscale, safe*err**alpha )
                isgood = False
            return isgood, max( self.hmin, min( self.hmax, hnext ) )




class RK8Stepper( ODEStepper ):
    """
    Eighth-order adjustable step size Dormand-Prince embedded method Runga-Kutta
    ODE stepper.

    Returns new time, new state vector, and suggested next step size.

    derivs    A SVDerivs object.
    """
    def __init__( self, derivs, stepmin=STEPMIN, stepmax=STEPMAX,
abstol=ABSTOL,
                  reltol=RELTOL ):
        ODEStepper.__init__( self, derivs, stepmin, stepmax, abstol, reltol )
        self.logger = mlogging.Log( name='dynamics.RK8Stepper' )

    def _step( self, t, s, h ):
        """
        Eighth-order adjustable step size Dormand-Prince embedded method
        Runga-Kutta ODE stepper.
```

```
        Adapted from Numerical Recipes stepperdopr853.h
        """
        k1   = h * self.D.derivs( t, s )
        k2   = h * self.D.derivs( t + RK8['c2']*h, s + RK8['a21']*k1 )
        k3   = h * self.D.derivs( t + RK8['c3']*h, s + RK8['a31']*k1 + RK8['a32']*k2 )
        k4   = h * self.D.derivs( t + RK8['c4']*h, s + RK8['a41']*k1 + RK8['a43']*k3 )
        k5   = h * self.D.derivs( t + RK8['c5']*h, s + RK8['a51']*k1 + RK8['a53']*k3 +
                                  RK8['a54']*k4 )
        k6   = h * self.D.derivs( t + RK8['c6']*h, s + RK8['a61']*k1 + RK8['a64']*k4 +
                                  RK8['a65']*k5 )
        k7   = h * self.D.derivs( t + RK8['c7']*h, s + RK8['a71']*k1 + RK8['a74']*k4 +
                                  RK8['a75']*k5 + RK8['a76']*k6 )
        k8   = h * self.D.derivs( t + RK8['c8']*h, s + RK8['a81']*k1 + RK8['a84']*k4 +
                                  RK8['a85']*k5 + RK8['a86']*k6 + RK8['a87']*k7 )
        k9   = h * self.D.derivs( t + RK8['c9']*h, s + RK8['a91']*k1 + RK8['a94']*k4 +
                                  RK8['a95']*k5 + RK8['a96']*k6 + RK8['a97']*k7 +
                                  RK8['a98']*k8 )
        k10 = h * self.D.derivs( t + RK8['c10']*h, s + RK8['a101']*k1 + RK8['a104']*k4 +
                                  RK8['a105']*k5 + RK8['a106']*k6 + RK8['a107']*k7 +
                                  RK8['a108']*k8 + RK8['a109']*k9 )
        k11 = h * self.D.derivs( t + RK8['c11']*h, s + RK8['a111']*k1 + RK8['a114']*k4 +
                                  RK8['a115']*k5 + RK8['a116']*k6 + RK8['a117']*k7 +
                                  RK8['a118']*k8 + RK8['a119']*k9 + RK8['a1110']*k10 )
        k12 = h * self.D.derivs( t + h, s + RK8['a121']*k1 + RK8['a124']*k4 +
                                  RK8['a125']*k5 + RK8['a126']*k6 + RK8['a127']*k7 +
                                  RK8['a128']*k8 + RK8['a129']*k9 + RK8['a1210']*k10 +
                                  RK8['a1211']*k11 )
        ksum = ( RK8['b1']*k1 + RK8['b6']*k6 + RK8['b7']*k7 + RK8['b8']*k8 +
                 RK8['b9']*k9 + RK8['b10']*k10 + RK8['b11']*k11 + RK8['b12']*k12 )
        err5 = ksum - RK8['bhh1']*k1 - RK8['bhh2']*k9 - RK8['bhh3']*k12                # 5th
order
        err3 = ( RK8['er1']*k1 + RK8['er6']*k6 + RK8['er7']*k7 + RK8['er8']*k8 +   # 3rd
order
                 RK8['er9']*k9 + RK8['er10']*k10 + RK8['er11']*k11 +
                 RK8['er12']*k12 )
        self.errvec = np.concatenate( [err5, err3] )
        sv = s + ksum
        t = t + h
        return t, sv

    def _errcalc( self, old_s, new_s ):
        """
        Calculate error measure for the current 8th-order step.
        """
        return dynamics_calcs.errcalc_RK8( old_s, new_s, self.errvec,
                                           self.abstol, self.reltol )
```

*****CYTHON CODE:

```
def errcalc_RK8( np.ndarray[double, ndim=1] old_s, np.ndarray[double, ndim=1] new_s,
                 np.ndarray[double, ndim=1] errvec, double abstol, double reltol ):
    """
    Calculate error measure for the current 8th-order Runga-Kutta ODE stepper step.
    """
    cdef int N = len( old_s )
    cdef np.ndarray[double, ndim=1] err5 = errvec[:N]   # Fifth-order error vector.
    cdef np.ndarray[double, ndim=1] err3 = errvec[N:]   # Third-order error vector.
    cdef double sk
    cdef double errsum5 = 0.0
    cdef double errsum3 = 0.0
    cdef int k
    for k in range(N):
        sk = abstol + reltol*max( abs(old_s[k]), abs(new_s[k]) )
        errsum5 += ( err5[k]/sk )**2
        errsum3 += ( err3[k]/sk )**2
    cdef double denom = errsum3 + 0.01*errsum5
    if denom <= sys.float_info.epsilon:
        denom = 1.0
    return errsum5/sqrt(N*denom)
```

```
def SShelio_derivs( double t, np.ndarray[double, ndim=1] s,
                    np.ndarray[double, ndim=1] m, double mscaling,
                    str interactions, int nplanets ):
    """
    Calculate the derivatives of the heliocentric solar system state vector <s>
    at time <t>. <m>[k] contains body k mass in units of solar masses. <mscaling>
    is the mass scaling: 1.0 + m0/M, where m0 is the mass of the reference body
    and M is the mass of the sun.

    <interactions>:
      'all'          interactions are calculated between all bodies (full N-body problem)
      'partial'      no interactions between minor bodies
      'restricted'   minor bodies have no effect on planets (restricted N-body problem)

    Returns a numpy ndarray which contains the derivatives of <s>.
```

```
    Note: the array optimizations here reduce cpu time by 98.54 percent.
    """
    # Velocities.
    cdef int N  = len(s)
    cdef int N2 = int(N/2)
    cdef np.ndarray[double, ndim=1] ds = np.zeros(N)
    ds[:N2] = np.copy( s[N2:] )

    # Scalar distance quantities.
    cdef int Nb = int(N/6)   # number of bodies
    cdef np.ndarray[double, ndim=1] rc3  = np.zeros(Nb)          # 1/|r_i|**3 where r_i
is distance from central body
    cdef np.ndarray[double, ndim=2] rrel3 = np.zeros([Nb,Nb])  # 1/|r_i - r_k|**3
(diagonally symmetric)
    cdef int k, i, k3, i3
    cdef double xk, yk, zk, rk2, xi, yi, zi, rik2
    for k in range(Nb):
        # Keplerian interaction with central body.
        k3  = 3*k
        xk  = s[k3]
        yk  = s[k3+1]
        zk  = s[k3+2]
        rk2 = xk*xk + yk*yk + zk*zk
        rc3[k] = rk2*rk2**0.5   # Amazingly, this is slightly faster than sqrt

        # Body-body interactions.
        for i in range(k):
            if i >= nplanets:
                if interactions == 'restricted':
                    break
                elif interactions == 'partial' and k >= nplanets:
                    break
            # Lower triangle.
            i3 = 3*i
            xi = s[i3]
            yi = s[i3+1]
            zi = s[i3+2]
            rik2 = (xi-xk)**2 + (yi-yk)**2 + (zi-zk)**2
            rrel3[k,i] = rik2*rik2**0.5
            # Upper triangle, minus the diagonal.
            if i != k:
                rrel3[i,k] = rrel3[k,i]

    # Acceleration calculation.
    cdef np.ndarray[double, ndim=1] a = np.zeros(Nb*3)
```

```
    cdef double mi, mk, rr3, r3, akx, aky, akz, aix, aiy, aiz
## Huh. Oddly enough, by-hand pointer indexing actually slows things down slightly.
##     cdef double *sdata = <double *>s.data
##     cdef double *xka, *yka, *zka, *xia, *yia, *zia
##     cdef double *rr3data = <double *>rrel3.data
##     cdef double *rr3p
    for k in range(Nb):
        # Keplerian interaction with central body.
        k3  = 3*k
        xk  = s[k3]
        yk  = s[k3+1]
        zk  = s[k3+2]
        mk  = -(1.0 + m[k])/mscaling
        r3  = rc3[k]/mk
        akx = xk/r3
        aky = yk/r3
        akz = zk/r3
##          xka = sdata + k3
##          yka = xka + 1
##          zka = yka + 1
##          mk  = -(1.0 + m[k])/mscaling
##          r3  = rc3[k]/mk
##          akx = xka[0]/r3
##          aky = yka[0]/r3
##          akz = zka[0]/r3

        # Body-body interactions.
        aix = aiy = aiz = 0.0
        for i in range(Nb):
            if i == k:
                continue
            if i >= nplanets:
                if interactions == 'restricted':
                    break
                elif interactions == 'partial' and k >= nplanets:
                    break
            i3 = 3*i
            xi = s[i3]
            yi = s[i3+1]
            zi = s[i3+2]
            mi = m[i]/mscaling
            rr3 = rrel3[k,i]/mi
##              rr3p = rr3data + Nb*k + i
##              rr3 = rr3p[0]/mi
            r3 = rc3[i]/mi
            aix += (xi-xk)/rr3 - xi/r3
```

```
                aiy += (yi-yk)/rr3 - yi/r3
                aiz += (zi-zk)/rr3 - zi/r3
##                  xia = sdata + i3
##                  yia = xia + 1
##                  zia = yia + 1
##                  mi = m[i]/mscaling
##                  rr3 = rrel3[k,i]/mi
##                  r3 = rc3[i]/mi
##                  aix += (xia[0]-xka[0])/rr3 - xia[0]/r3
##                  aiy += (yia[0]-yka[0])/rr3 - yia[0]/r3
##                  aiz += (zia[0]-zka[0])/rr3 - zia[0]/r3
            a[k3] = akx + aix
            a[k3+1] = aky + aiy
            a[k3+2] = akz + aiz

        ds[N2:] = a

        return ds
```

*****FFT AND TIME SERIES CODE:

```
def load_baseline( self ):
        """
        Load the selected baseline data file.
        """
        self.show_header()
        self.getparams()
        files = mGUIutils.get_selected_filenames( self.dirTreeView )
        if not files:
            return
        self.setCursor( QtCore.Qt.BusyCursor )
        self.fname = files[0]
        self.dataFileLabel.setText( "%s" % self.fname )
        self.dataFileLabel.setVisible(True)
        M = np.loadtxt( self.fname )
        self.t = M[:,0]
        if self.n and M.shape[0] != self.n:
            self.log( "New baseline data is different length!" )
            self.log( "Dumping previous comparison data." )
            self.comparedata = None
            self.ncols_comp   = None
        if self.subtractlinear:
            # Remove linear trend.
```

```
                ncols = M.shape[1]
                for k in range(1, ncols):
                    y = M[:,k]
                    T = timeseries.TimeSeries( self.t, y, self.queue )
                    T.removelinear()
                    M[:,k] = T.f
            self.basedata = M[:,1:]
            self.n, self.ncols_base = self.basedata.shape
            T = timeseries.TimeSeries( self.t, self.basedata[:,0], self.queue )
            self.freqs = T.freqs
            self.xmin_ts = self.t[0]
            self.xmax_ts = self.t[-1]
            self.ymin_ts = 0
            self.ymax_ts = 1
            self.xmin_spec = self.freqs[0]
            self.xmax_spec = self.freqs[-1]
            self.ymin_spec = 0
            self.ymax_spec = 1
            self.setparams()

            self.log( "%i data rows and %i dependent variable columns read from %s"
                      % (self.n, self.ncols_base, self.fname) )
            self.addColumnComboBox.clear()
            self.addColumnComboBox.addItems( ["%i" % (k+1) for k in
range(self.ncols_base)] )
            self.addColumnComboBox.setEnabled(True)
            self.plots_clear()
            self.subtractLinearCheckBox.setEnabled(False)
            if not self.subtractlinear:
                self.compareDataButton.setEnabled(True)

            # If just one data column, process it now.
            if self.basedata.shape[1] == 1:
                self.choose_column(0)
            self.setCursor( QtCore.Qt.ArrowCursor )

        def load_comparison( self ):
            """
            Load the selected comparison data file.
            """
            self.show_header()
            self.getparams()
            files = mGUIutils.get_selected_filenames( self.dirTreeView )
            if not files:
                return
            self.setCursor( QtCore.Qt.BusyCursor )
```

```
            self.fname = files[0]
            M = np.loadtxt( self.fname )
            if M.shape[0] != self.n:
                self.log( "Baseline and comparison data must be same length!" )
                return
            if M[1,0]-M[0,0] != self.t[1]-self.t[0]:
                self.log( "Baseline and comparison output time steps must be equal!" )
                return
            oldfname = os.path.basename( str( self.dataFileLabel.text() ) )
            self.dataFileLabel.setText( "(%s) - (%s)" % (self.fname, oldfname) )
            self.comparedata = M[:,1:]
            self.n, self.ncols_comp = self.comparedata.shape
            self.log( "%i data rows and %i dependent variable columns read from %s"
                        % (self.n, self.ncols_comp, self.fname) )
            n = min( self.ncols_base, self.ncols_comp )
            self.addColumnComboBox.clear()
            self.addColumnComboBox.addItems( ["%i" % (k+1) for k in range(n)] )
            ndiff = abs( self.ncols_base - self.ncols_comp )
            self.comparisonOffsetComboBox.clear()
            self.comparisonOffsetComboBox.addItems( ["%i" % k for k in
range(-ndiff,ndiff+1)] )
            self.comparisonOffsetComboBox.setCurrentIndex( ndiff )
            self.comparisonOffsetLabel.setEnabled(True)
            self.comparisonOffsetComboBox.setEnabled(True)
            #self.subtractLinearCheckBox.setEnabled(True)
            # If just one data column, process it now.
            if self.comparedata.shape[1] == 1:
                self.choose_column(0)
            self.setCursor( QtCore.Qt.ArrowCursor )

        def plot_time_series( self, T ):
            """
            Plot the supplied time series.
            """
            self.setCursor( QtCore.Qt.BusyCursor )
            self.getparams()
            self.tsmin = min( self.tsmin, T.min )
            self.tsmax = max( self.tsmax, T.max )
            c = self.colors[ self.ncurves % len(self.colors) ]
            self.axes_ts.plot( T.t, T.f, c, label=self.labels[-1] )
            self.set_plots_page()
            self.setCursor( QtCore.Qt.ArrowCursor )

        def plot_spectrum( self, T ):
            """
            Plot the frequency spectrum from the supplied time series.
```

```
"""
self.setCursor( QtCore.Qt.BusyCursor )
self.getparams()
# Keep track of min.
specsorted = np.sort( T.spec )
specmin = 0
for k, val in enumerate(specsorted):
      if val > 0:
            specmin = val
            break
self.specmin = min( self.specmin, specmin )
self.specmax = max( self.specmax, T.spec.max() )
# Plot.
c = self.colors[ self.ncurves % len(self.colors) ]
self.axes_spec.plot( T.freqs, T.spec, c, label=self.labels[-1] )
self.set_plots_page()
self.setCursor( QtCore.Qt.ArrowCursor )
```