Maxima by Example:
Ch. 3, Ordinary Differential Equation Tools *

Edwin L. Woollett

September 16, 2010

Contents

*This version uses **Maxima 5.18.1** Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see later in this chapter
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the XMaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.18.1
 (2009). http://maxima.sourceforge.net/
```

The homemade function **fll(x)** (first, last, length) is used to return the first and last elements of lists (as well as the length), and is automatically loaded in with **mbe1util.mac** from Ch. 1. We will include a reference to this definition when working with lists.

This function has the definitions

```
fll(x) := [first(x),last(x),length(x)]$
declare(fll,evfun)$
```

Some of the examples used in these notes are from the Maxima html help manual or the Maxima mailing list: **http://maxima.sourceforge.net/maximalist.html**.

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

3.1 Solving Ordinary Differential Equations

3.2 Solution of One First Order Ordinary Differential Equation (ODE)

3.2.1 Summary Table

ode2 and ic1
gsoln : ode2 (de, u, t); where de involves 'diff(u,t). psoln : ic1 (gsoln, t = t0, u = u0);
desolve
gsoln : desolve(de, u(t)); where de includes the equal sign (=) and 'diff(u(t),t) and possibly u(t). psoln : ratsubst(u0val,u(o),gsoln)
plotdf
plotdf (dudt, [t,u], [trajectory_at, t0, u0], [direction,forward], [t, tmin, tmax], [u, umin, umax])$
rk
points : rk (dudt, u, u0, [t, t0, tlast, dt])$
where dudt is a function of t and u which determines diff(u,t).

Table 1: Methods for One First Order ODE

We will use these four different methods to solve the first order ordinary differential equation

$$\frac{\mathrm{d}\,\mathrm{u}}{\mathrm{d}\,\mathrm{t}} = \mathrm{e}^{-\mathrm{t}} + \mathrm{u} \tag{3.1}$$

subject to the condition that when $\mathrm{t} = 2, \quad \mathrm{u} = -0.1$.

3.2.2 Exact Solution with ode2 and ic1

Most ordinary differential equations have no known exact solution (or the exact solution is a complicated expression involving many terms with special functions) and one normally uses approximate methods. However, some ordinary differential equations have simple exact solutions, and many of these can be found using **ode2**, **desolve**, or **contrib_ode**. The manual has the following information about **ode2**

> Function: **ode2 (eqn, dvar, ivar)**
> The function **ode2** solves an ordinary differential equation (ODE) of first or second order. It takes three arguments: an ODE given by **eqn**, the dependent variable **dvar**, and the independent variable **ivar**. When successful, it returns either an explicit or implicit solution for the dependent variable. `%c` is used to represent the integration constant in the case of first-order equations, and `%k1` and `%k2` the constants for second-order equations. The dependence of the dependent variable on the independent variable does not have to be written explicitly, as in the case of **desolve**, but the independent variable must always be given as the third argument.

If the differential equation has the structure **Left(dudt,u,t) = Right(dudt,u,t)** (here **u** is the dependent variable and **t** is the independent variable), we can always rewrite that differential equation as **de = Left(dudt,u,t) - Right(dudt,u,t) = 0**, or **de = 0**.

We can use the syntax **ode2(de,u,t)**, with the first argument an expression which includes derivatives, instead of the complete equation including the **" = 0"** on the end, and **ode2** will assume we mean **de = 0** for the differential equation. (Of course you can also use **ode2 (de=0, u, t)**

We rewrite our example linear first order differential equation Eq. 3.1 in the way just described, using the **noun** form **'diff**, which uses a single quote. We then use **ode2**, and call the general solution **gsoln**.

```
(%i1) de : 'diff(u,t)- u - exp(-t);
                                 du          - t
(%o1)                            -- - u - %e
                                 dt
(%i2) gsoln : ode2(de,u,t);
                                          - 2 t
                                       %e          t
(%o2)                        u = (%c - -------) %e
                                         2
```

The general solution returned by **ode2** contains one constant of integration **%c**, and is an explicit solution for **u** as a function of **t**, although the above does not bind the symbol **u**.

We next find the particular solution which has $\mathbf{t = 2}$, $\mathbf{u = -0.1}$ using **ic1**, and call this particular solution **psoln**. We then check the returned solution in two ways: 1. does it satisfy the conditions given to **ic1**?, and 2. does it produce a zero value for our expression **de**?

```
(%i3) psoln : ic1(gsoln,t = 2, u = -0.1),ratprint:false;
                            - t - 4     2        2 t       4
                         %e        ((%e  - 5) %e    + 5 %e )
(%o3)             u = - ------------------------------------
                                       10
(%i4) rhs(psoln),t=2,ratsimp;
                                       1
(%o4)                                - --
                                       10
(%i5) de,psoln,diff,ratsimp;
(%o5)                                  0
```

Both tests are passed by this particular solution. We can now make a quick plot of this solution using **plot2d**.

```
(%i6) us : rhs(psoln);
                          - t - 4     2        2 t       4
                       %e        ((%e  - 5) %e    + 5 %e )
(%o6)                - ------------------------------------
                                     10
(%i7) plot2d(us,[t,0,7],
           [style,[lines,5]],[ylabel," "],
           [xlabel,"t0 = 2, u0 = -0.1, du/dt = exp(-t) + u"])$
```

which looks like

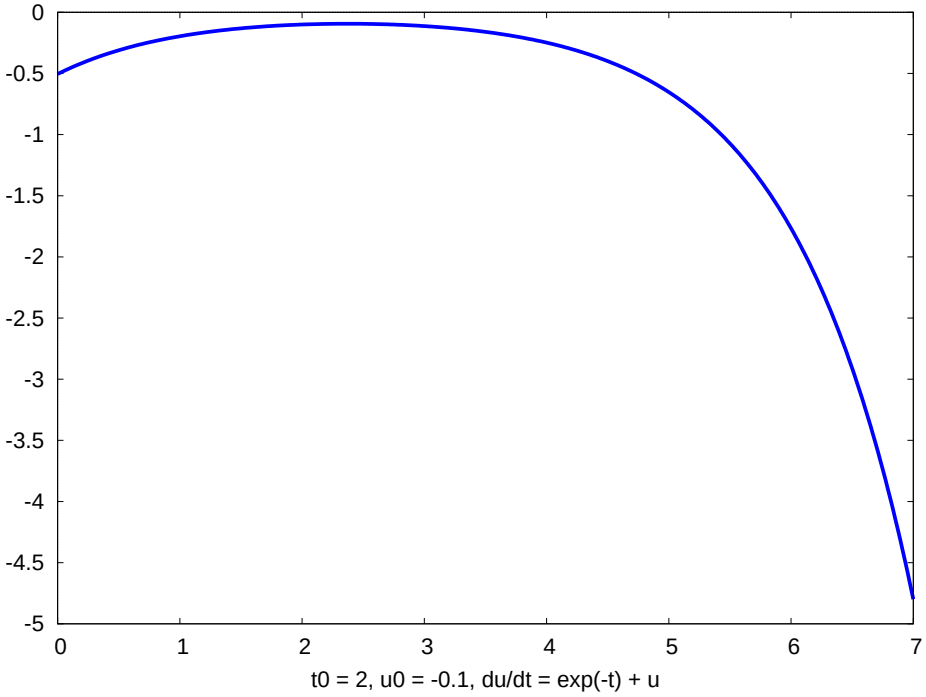

Figure 1: Solution for which $\mathbf{t = 2},\ \mathbf{u = -0.1}$

3.2.3 Exact Solution Using desolve

desolve uses Laplace transform methods to find an exact solution. To be able to use **desolve**, we need to write our example differential equation Eq.3.1 in a more explicit form, with every **`u -> u(t)`**, and include the **=** sign in the definition of the differential equation.

```
(%i1) eqn : 'diff(u(t),t) - exp(-t) - u(t) = 0;
                        d                    - t
(%o1)                   -- (u(t)) - u(t) - %e    = 0
                        dt
(%i2) gsoln : desolve(eqn,u(t));
                                             t     - t
                                (2 u(0) + 1) %e    %e
(%o2)                   u(t) = ---------------- - -----
                                       2             2
(%i3) eqn,gsoln,diff,ratsimp;
                                    0 = 0
(%i4) bc : subst ( t=2, rhs(gsoln)) = - 0.1;
                        2                   - 2
                       %e  (2 u(0) + 1)   %e
(%o4)                  ---------------- - ----- = - 0.1
                              2             2
(%i5) solve ( eliminate ( [gsoln, bc],[u(0)]), u(t) ),ratprint:false;
                                - t     t - 2       t - 4
                           - 5 %e    - %e      + 5 %e
(%o5)              [u(t) = -------------------------------]
                                       10
(%i6) us : rhs(%[1]);
                              - t     t - 2       t - 4
                         - 5 %e    - %e      + 5 %e
(%o6)                    -------------------------------
                                       10
```

```
(%i7) us, t=2, ratsimp;
                                        1
(%o7)                                 - --
                                        10
(%i8) plot2d(us,[t,0,7],
           [style,[lines,5]],[ylabel," "],
           [xlabel,"t0 = 2, u0 = -0.1, du/dt = exp(-t) + u"])$
```

and we get the same plot as before. The function **desolve** returns a solution in terms of the "initial value" **u(0)**, which here means **u(t = 0)**, and we must go through an extra step to eliminate **u(0)** in a way that assures our chosen boundary condition **t = 2, u = -0.1** is satisfied.

We have checked that the general solution satisfies the given differential equation in **%i3**, and have checked that our particular solution satisfies the desired condition at **t = 2** in **%i7**.

If your problem requires that the value of the solution **us** be specified at **t = 0**, the route to the particular solution is much simpler than what we used above. You simply use **subst (u(0) = -1, rhs (gsoln))** if, for example, you wanted a particular solution to have the property that when **t = 0, u = -1**.

```
(%i9) us : subst( u(0) = -1,rhs(gsoln) ),ratsimp;
                                  - t    2 t
                                 %e    (%e    + 1)
(%o9)                          - -----------------
                                          2
(%i10) us,t=0,ratsimp;
(%o10)                                  - 1
```

3.2.4 Numerical Solution and Plot with plotdf

We next use **plotdf** to numerically integrate the given first order ordinary differential equation, draw a direction field plot which governs any particular solution, and draw the particular solution we have chosen.

The default color choice of **plotdf** is to use small blue arrows give the local direction of the trajectory of the particular solution passing though that point. This direction can be defined by an angle α such that if $\mathbf{u}' = \mathbf{f}(\mathbf{t}, \mathbf{u})$, then $\mathbf{tan}(\alpha) = \mathbf{f}(\mathbf{t}, \mathbf{u})$, and at the point $(\mathbf{t_0},\ \mathbf{u_0})$,

$$\mathbf{d\,u} = \mathbf{f}(\mathbf{t_0}, \mathbf{u_0}) \times \mathbf{d\,t} = \mathbf{d\,t} \times \left(\frac{\mathbf{d\,u}}{\mathbf{d\,t}}\right)_{\mathbf{t=t_0,\ u=u_0}} \tag{3.2}$$

This equation determines the increase **d u** in the value of the dependent variable **u** induced by a small increase **d t** in the independent variable **t** at the point $(\mathbf{t_0},\ \mathbf{u_0})$. We then define a local vector with **t** component **d t** and **u** component **d u**, and draw a small arrow in that direction at a grid of chosen points to construct a direction field associated with the given first order differential equation. The length of the small arrow can be increased some to reflect large values of the magnitude of $\mathbf{d\,u}/\mathbf{d\,t}$.

For one first order ordinary differential equation, **plotdf**, has the syntax

```
plotdf( dudt,[t,u], [trajectory_at, t0, u0], options ... )
```

in which **dudt** is the function of **(t,u)** which determines the rate of change $\mathbf{d\,u}/\mathbf{d\,t}$.

```
(%i1) plotdf(exp(-t) + u, [t, u], [trajectory_at,2,-0.1],
             [direction,forward], [t,0,7], [u, -6, 1] )$
```

produces the plot

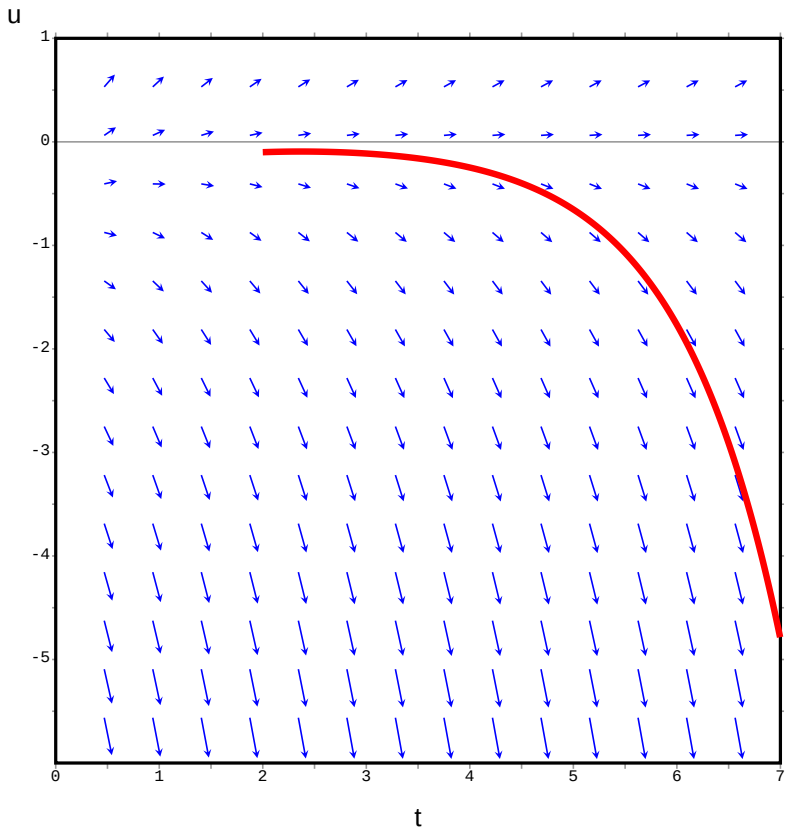

Figure 2: Direction Field for the Solution $\mathbf{t = 2},\ \mathbf{u = -0.1}$

(We have thickened the red curve using the **Config, Linewidth** menu option of **plotdf**, followed by **Replot**).

The help manual has an extensive discussion and examples of the use of the direction field plot utility **plotdf**.

3.2.5 Numerical Solution with 4th Order Runge-Kutta: rk

Although the **plotdf** function is useful for orientation about the shapes and types of solutions possible, if you need a list with coordinate points to use for other purposes, you can use the fourth order Runge-Kutta function **rk**.

For one first order ordinary differential equation, the syntax has some faint resemblance to that of **plotdf**:

```
   rk ( dudt, u, u0, [ t, t0, tlast, dt ] )
```

in which we are assuming that **u** is the dependent variable and **t** is the independent variable, and **dudt** is that function of **(t, u)** which locally determines the value of $\mathbf{d\,u/d\,t}$. This will numerically integrate the corresponding first order ordinary differential equation and return a list of pairs of **(t, u)** on the solution curve which has been requested:

```
  [ [t0, u0], [t0 + dt, y(t0 + dt)], .... ,[tlast, y(tlast)] ]
```

For our example first order ordinary differential equation, choosing the same initial conditions as above, and choosing **dt = 0.01**,

```
(%i1) fpprintprec:8$
(%i2) points : rk (exp(-t) + u, u, -0.1, [ t, 2, 7, 0.01 ] )$
(%i3) %, fll;
(%o3)                 [[2, - 0.1], [7.0, - 4.7990034], 501]
(%i4) plot2d( [ discrete, points ], [ t, 0, 7],
           [style,[lines,5]],[ylabel," "],
           [xlabel,"t0 = 2, u0 = -0.1, du/dt = exp(-t) + u"])$
```

(We have used our homemade function **fll(x)**, loaded in at startup with the other functions defined in **mbe1util.mac**, available with the Ch. 1 material. We have provided the definition of **fll** in the preface of this chapter. Instead of **%, fll ;**, you could use **[%[1],last(%),length(%)];** to get the same information.)

The plot looks like

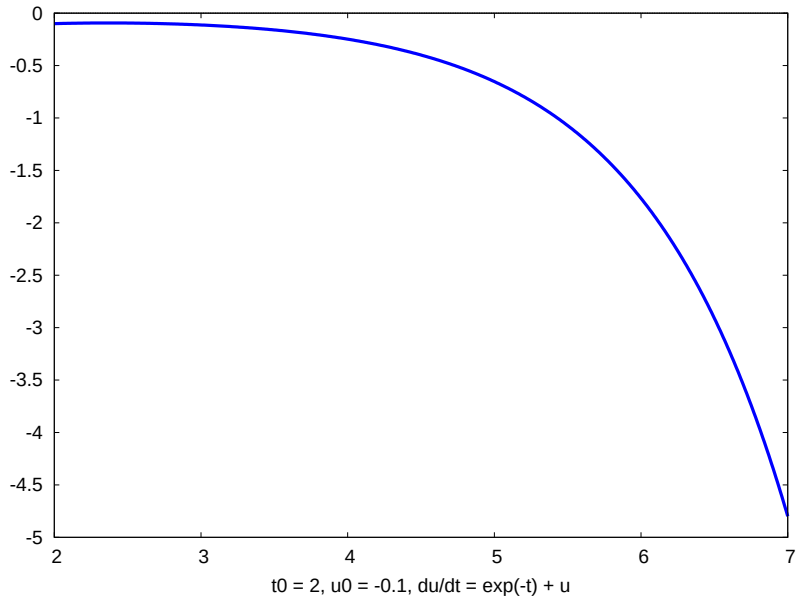

Figure 3: Runge-Kutta Solution with $\mathbf{t = 2},\ \mathbf{u = -0.1}$

3.3 Solution of One Second Order ODE or Two First Order ODE's

3.3.1 Summary Table

ode2 and ic1
gsoln : ode2 (de, u, t); where de involves 'diff(u,t,2) and possibly 'diff(u,t). psoln : ic2 (gsoln, t = t0, u = u0, 'diff(u,t) = up0);
desolve
atvalue ('diff(u,t), t = 0, v(0)); gsoln : desolve(de, u(t)); where de includes the equal sign (=), 'diff(u(t),t,2), and possibly 'diff(u(t),t) and u(t). One type of particular solution is returned by using psoln : subst([u(o) = u0, v(0) = v0] , gsoln)
plotdf
plotdf ([dudt, dvdt], [u, v], [trajectory_at, u0, v0], [u, umin, umax],[v, vmin, vmax], [tinitial, t0], [direction,forward], [versus_t, 1],[tstep, timestepval], [nsteps, nstepsvalue])$
rk
points : rk ([dudt, dvdt],[u, v],[u0, v0],[t, t0, tlast, dt])$
where dudt and dvdt are functions of t,u, and v which determine diff(u,t) and diff(v,t).

Table 2: Methods for One Second Order or Two First Order ODE's

We apply the above four methods to the simple second order ordinary differential equation:

$$\frac{\mathbf{d}^2\,\mathbf{u}}{\mathbf{d}\,\mathbf{t}^2} = 4\,\mathbf{u} \tag{3.3}$$

subject to the conditions that when $\mathbf{t} = \mathbf{2}$, $\mathbf{u} = \mathbf{1}$ and $\mathbf{d}\,\mathbf{u}/\mathbf{d}\,\mathbf{t} = \mathbf{0}$.

3.3.2 Exact Solution with ode2, ic2, and eliminate

The main difference here is the use of **ic2** rather than **ic1**.

```
(%i1) de : 'diff(u,t,2) - 4*u;
                                    2
                                   d u
(%o1)                              --- - 4 u
                                     2
                                   dt
(%i2) gsoln : ode2(de,u,t);
                                    2 t          - 2 t
(%o2)                  u = %k1 %e     + %k2 %e
(%i3) de,gsoln,diff,ratsimp;
(%o3)                                 0
```

```
(%i4) psoln : ic2(gsoln,t=2,u=1,'diff(u,t) = 0);
                                  2 t - 4     4 - 2 t
                                %e          %e
(%o4)                      u = --------- + ---------
                                     2           2
(%i5) us : rhs(psoln);
                                2 t - 4     4 - 2 t
                              %e          %e
(%o5)                         --------- + ---------
                                  2           2
(%i6) us, t=2, ratsimp;
(%o6)                                 1
(%i7) plot2d(us,[t,0,4],[y,0,10],
          [style,[lines,5]],[ylabel," "],
          [xlabel," U versus t, U''(t) = 4 U(t), U(2) = 1, U'(2) = 0 "])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
```

which produces the plot

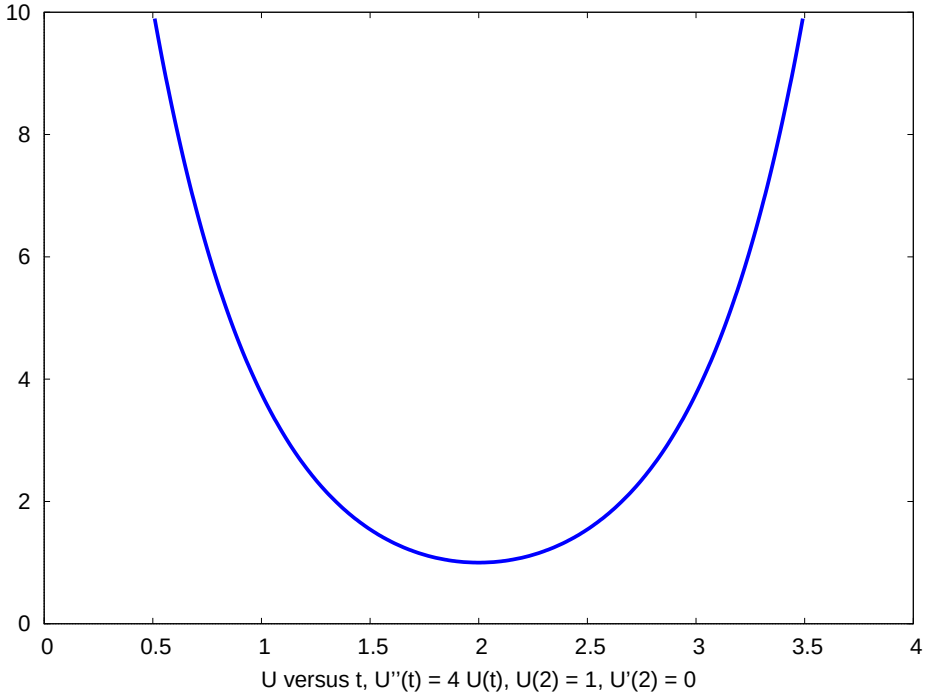

Figure 4: Solution for which $\mathbf{t = 2},\ \mathbf{u = 1},\ \mathbf{u' = 0}$

Next we make a "phase space plot" which is a plot of $\mathbf{v = d\,u/d\,t}$ versus $\mathbf{u}$ over the range $\mathbf{1 \le t \le 3}$.

```
(%i8) vs : diff(us,t),ratsimp;
                              - 2 t - 4    4 t     8
(%o8)                        %e          (%e    - %e )
(%i9) for i thru 3 do
        d[i]:[discrete,[float(subst(t=i,[us,vs]))]]$
(%i10) plot2d( [[parametric,us,vs,[t,1,3]],d[1],d[2],d[3] ],
            [x,0,8],[y,-12,12],
          [style, [lines,5,1],[points,4,2,1],
            [points,4,3,1],[points,4,6,1]],
          [ylabel," "],[xlabel," "],
           [legend," du/dt vs u "," t = 1 ","t = 2","t = 3"] )$
```

which produces the plot

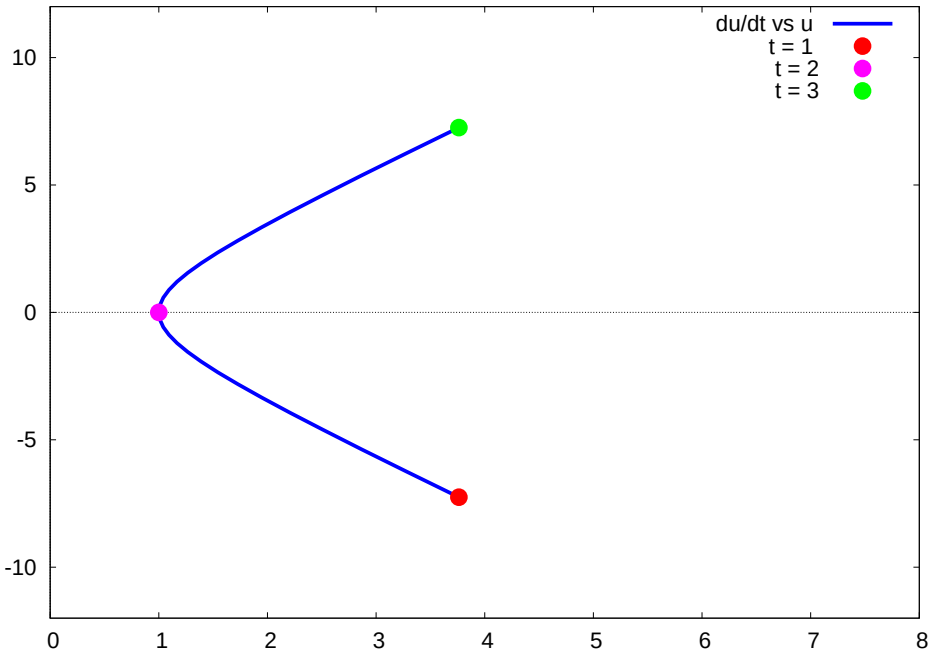

Figure 5: $\mathbf{t = 2,\ y = 1,\ y' = 0}$ Solution

If your boundary conditions are, instead, for **`t=0, u = 1`**, and for **`t = 2, u = 4`**, then one can eliminate the two constants "by hand" instead of using **ic2** (see also next section).

```
(%i4) bc1 : subst(t=0,rhs(gsoln)) = 1$
(%i5) bc2 : subst(t = 2, rhs(gsoln)) = 4$
(%i6) solve(
        eliminate([gsoln,bc1,bc2],[%k1,%k2]), u ),
           ratsimp, ratprint:false;
                        - 2 t       4         4 t     8       4
                       %e      ((4 %e  - 1) %e    + %e  - 4 %e )
(%o6)            [u = ------------------------------------------]
                                       8
                                     %e  - 1
(%i7) us : rhs(%[1]);
                       - 2 t       4         4 t     8       4
                     %e      ((4 %e  - 1) %e    + %e  - 4 %e )
(%o7)                ------------------------------------------
                                      8
                                    %e  - 1
(%i8) us,t=0,ratsimp;
(%o8)                                 1
(%i9) us,t=2,ratsimp;
(%o9)                                 4
```

3.3.3 Exact Solution with desolve, atvalue, and eliminate

The function **desolve** uses Laplace transform methods which are set up to expect the use of initial values for dependent variables and their derivatives. (However, we will show how you can impose more general boundary conditions.) If the dependent variable is `u(t)`, for example, the solution is returned in terms of a constant `u(0)`, which refers to the value of `u(t = 0)` (here we are assuming that the independent variable is `t`). To get a simple result from **desolve** which we can work with (for the case of a second order ordinary differential equation), we can use the **atvalue** function with the syntax (for example):

```
atvalue ( 'diff(u,t), t = 0, v(0) )
```

which will allow **desolve** to return the solution to a second order ODE in terms of the pair of constants `( u(0), v(0) )`. Of course, there is nothing sacred about using the symbol `v(0)` here. The function **atvalue** should be invoked before the use of **desolve**.

If the desired boundary conditions for a particular solution refer to `t = 0`, then you can immediately find that particular solution using the syntax (if `ug` is the general solution, say)

```
us : subst( [ u(0) = u0val, v(0) = v0val], ug ),
```

or else by using **ratsubst** twice.

In our present example, the desired boundary conditions refer to `t = 2` , and impose conditions on the value of **u** and its first derivative at that value of `t`. This requires a little more work, and we use **eliminate** to get rid of the constants `(u(0), v(0))` in a way that allows our desired conditions to be satisfied.

```
(%i1) eqn : 'diff(u(t),t,2) - 4*u(t) = 0;
                              2
                             d
(%o1)                        --- (u(t)) - 4 u(t) = 0
                               2
                             dt
(%i2) atvalue ( 'diff(u(t),t), t=0, v(0) )$
(%i3) gsoln : desolve(eqn,u(t));
                                     2 t                       - 2 t
                   (v(0) + 2 u(0)) %e       (v(0) - 2 u(0)) %e
(%o3)       u(t) = ------------------- - ----------------------
                             4                        4
(%i4) eqn,gsoln,diff,ratsimp;
(%o4)                                0 = 0
(%i5) ug : rhs(gsoln);
                                  2 t                       - 2 t
                (v(0) + 2 u(0)) %e       (v(0) - 2 u(0)) %e
(%o5)           ------------------- - ----------------------
                          4                        4
(%i6) vg : diff(ug,t),ratsimp$
(%i7) ubc : subst(t = 2,ug) = 1$
(%i8) vbc : subst(t = 2,vg) = 0$
(%i9) solve (
          eliminate([gsoln, ubc, vbc],[u(0), v(0)]), u(t) ),
           ratsimp,ratprint:false;
                              - 2 t - 4    4 t     8
                             %e          (%e    + %e )
(%o9)                [u(t) = -------------------------]
                                       2
```

```
(%i10) us : rhs(%[1]);
                             - 2 t - 4    4 t     8
                           %e          (%e    + %e )
(%o10)                     ------------------------
                                      2
(%i11) subst(t=2, us),ratsimp;
(%o11)                                1
(%i12) vs : diff(us,t),ratsimp;
                             - 2 t - 4    4 t     8
(%o12)                     %e          (%e    - %e )
(%i13) subst(t = 2,vs),ratsimp;
(%o13)                                0
(%i14) plot2d(us,[t,0,4],[y,0,10],
           [style,[lines,5]],[ylabel," "],
           [xlabel," U versus t, U''(t) = 4 U(t), U(2) = 1, U'(2) = 0 "])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
(%i15) for i thru 3 do
        d[i]:[discrete,[float(subst(t=i,[us,vs]))]]$
(%i16) plot2d( [[parametric,us,vs,[t,1,3]],d[1],d[2],d[3] ],
             [x,0,8],[y,-12,12],
          [style, [lines,5,1],[points,4,2,1],
            [points,4,3,1],[points,4,6,1]],
          [ylabel," "],[xlabel," "],
           [legend," du/dt vs u "," t = 1 ","t = 2","t = 3"] )$
```

which generates the same plots found with the **ode2** method above.

If the desired boundary conditions are that **u** have given values at **t = 0** and **t = 3**, then we can proceed from the same general solution above as follows with **up** being a partially defined particular solution (assume **u(0) = 1** and **u(3) = 2**):

```
(%i17) up : subst(u(0) = 1, ug);
                                  2 t                 - 2 t
                     (v(0) + 2) %e       (v(0) - 2) %e
(%o17)               ---------------- - ----------------
                            4                   4
(%i18) ubc : subst ( t=3, up) = 2;
                       6                - 6
                     %e  (v(0) + 2)   %e    (v(0) - 2)
(%o18)               -------------- - ---------------- = 2
                           4                 4
(%i19) solve(
          eliminate ( [ u(t) = up, ubc ],[v(0)] ), u(t) ),
             ratsimp, ratprint:false;
                        - 2 t       6        4 t     12       6
                       %e      ((2 %e  - 1) %e    + %e   - 2 %e )
(%o19)          [u(t) = ---------------------------------------]
                                      12
                                     %e   - 1
(%i20) us : rhs (%[1]);
                     - 2 t       6        4 t     12       6
                    %e      ((2 %e  - 1) %e    + %e   - 2 %e )
(%o20)              ---------------------------------------
                                    12
                                   %e   - 1
(%i21) subst(t = 0, us),ratsimp;
(%o21)                                1
(%i22) subst (t = 3, us),ratsimp;
(%o22)                                2
```

```
(%i23) plot2d(us,[t,0,4],[y,0,10],
           [style,[lines,5]],[ylabel," "],
           [xlabel," U versus t, U''(t) = 4 U(t), U(0) = 1, U(3) = 2 "])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
```

which produces the plot

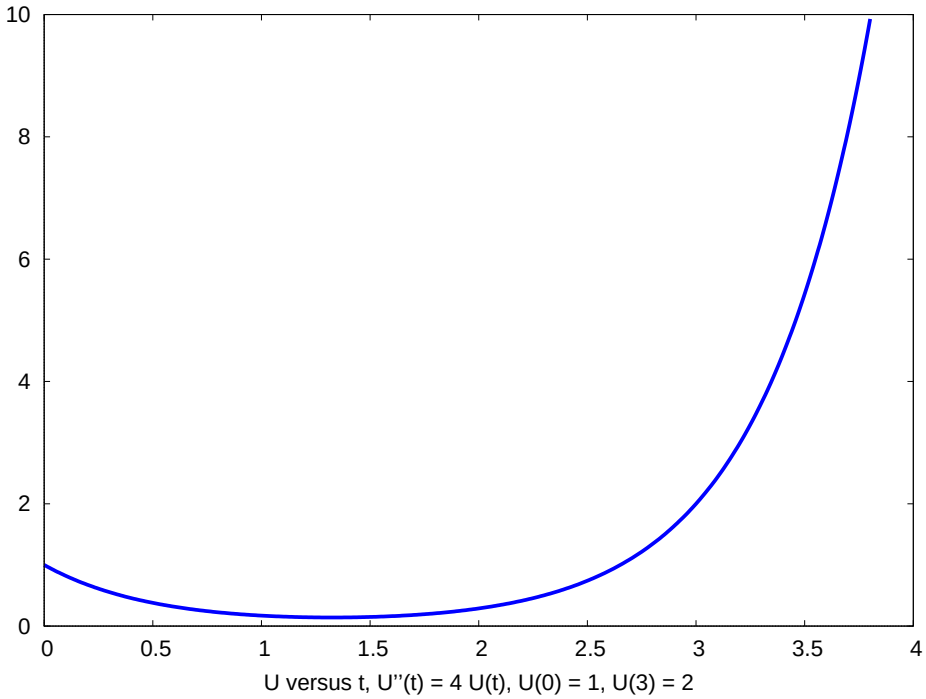

Figure 6: Solution for $\mathbf{u(0) = 1,\ u(3) = 2}$

If instead, you need to satisfy **`u(1) = -1`** and **`u(3) = 2`**, you could proceed from **`gsoln`** and **`ug`** as follows:

```
(%i24) ubc1 : subst ( t=1, ug) = -1$
(%i25) ubc2 : subst ( t=3, ug) = 2$
(%i26) solve(
         eliminate ( [gsoln, ubc1, ubc2],[u(0),v(0)]), u(t) ),
          ratsimp, ratprint:false;
                        - 2 t       4         4 t      12        8
                      %e      ((2 %e  + 1) %e    - %e   - 2 %e )
(%o26)        [u(t) = ------------------------------------------]
                                     10     2
                                   %e   - %e
(%i27) us : rhs(%[1]);
                      - 2 t       4         4 t      12        8
                    %e      ((2 %e  + 1) %e    - %e   - 2 %e )
(%o27)              ------------------------------------------
                                    10     2
                                  %e   - %e
(%i28) subst ( t=1, us), ratsimp;
                                     - 1
(%o28)
(%i29) subst ( t=3, us), ratsimp;
                                      2
(%o29)
(%i30) plot2d ( us, [t,0,4], [y,-2,8],
           [style,[lines,5]],[ylabel," "],
           [xlabel," U versus t, U''(t) = 4 U(t), U(1) = -1, U(3) = 2 "])$
```

which produces the plot

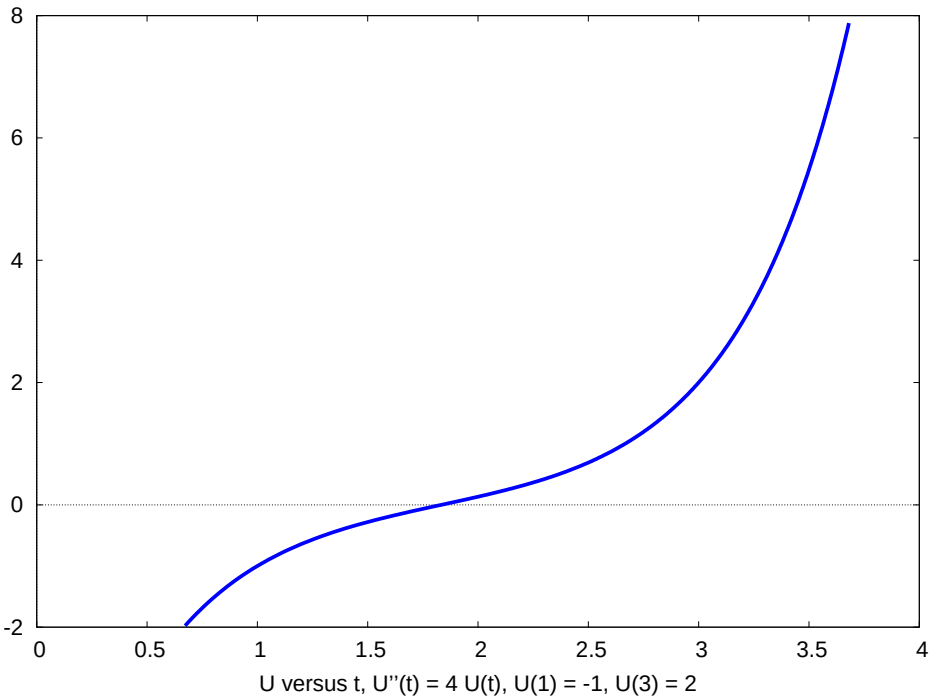

Figure 7: Solution for $\mathbf{u(1) = -1,\ u(3) = 2}$

The simplest case of using **desolve** is the case in which you impose conditions on the solution and its first derivative at `t = 0`, in which case you simply use:

```
(%i4) psoln : subst([u(0) = 1,v(0)=0],gsoln);
                                      2 t     - 2 t
                                    %e      %e
(%o4)                        u(t) = ----- + -------
                                      2        2
(%i5) us : rhs(psoln);
                                    2 t     - 2 t
                                  %e      %e
(%o5)                             ----- + -------
                                    2        2
```

in which we have chosen the initial conditions `u(0) = 1`, and `v(0) = 0`.

3.3.4 Numerical Solution and Plot with plotdf

Given a second order autonomous ODE, one needs to introduce a second dependent variable **v(t)**, say, which is defined as the first derivative of the original single dependent variable **u(t)**. Then for our example, the starting ODE

$$\frac{d^2 u}{d t^2} = 4\,u \tag{3.4}$$

is converted into two first order ODE's

$$\frac{d u}{d t} = v, \quad \frac{d v}{d t} = 4\,u \tag{3.5}$$

and the **plotdf** syntax for two first order ODE's is

```
plotdf ( [dudt, dvdt], [u, v], [trajectory_at, u0, v0], [u, umin, umax],
        [v, vmin, vmax], [tinitial, t0], [versus_t, 1],
        [tstep, timestepval], [nsteps, nstepsvalue] )$
```

in which at **t = t0**, **u = u0** and **v = v0**. If **t0 = 0** you can omit the option **[tinitial, t0]**. The options **[u, umin, umax]** and **[v, vmin, vmax]** allow you to control the horizontal and vertical extent of the phase space plot (here **v** versus **u**) which will be produced. The option **[versus_t,1]** tells **plotdf** to create a separate plot of both **u** and **v** versus the dependent variable. The last two options are only needed if you are not satisfied with the plots and want to experiment with other than the default values of **tstep** and **nsteps**.

Another option you can add is **[direction,forward]**, which will display the trajectory for **t** greater than or equal to **t0**, rather than for a default interval around the value **t0** which corresponds to **[direction,both]**.
Here we invoke **plotdf** for our example.

```
(%i1) plotdf ( [v, 4*u], [u, v], [trajectory_at, 1, 0],
             [u, 0, 8], [v, -10, 10], [versus_t, 1],
               [tinitial, 2])$
```

The plot versus **t** is

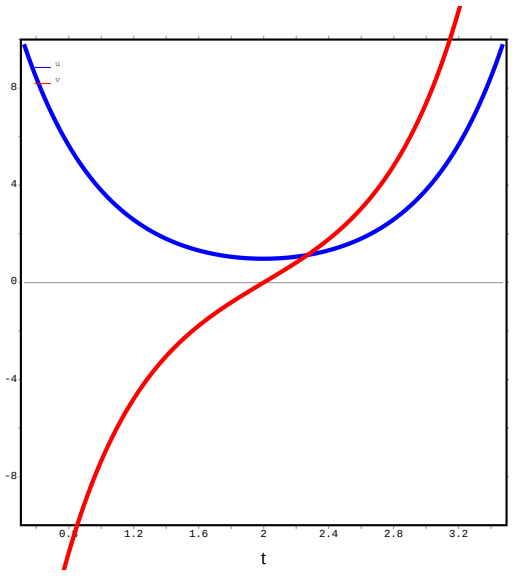

Figure 8: u(t) and u'(t) vs. t for $u(2) = 1,\ u'(2) = 0$

and the phase space plot is

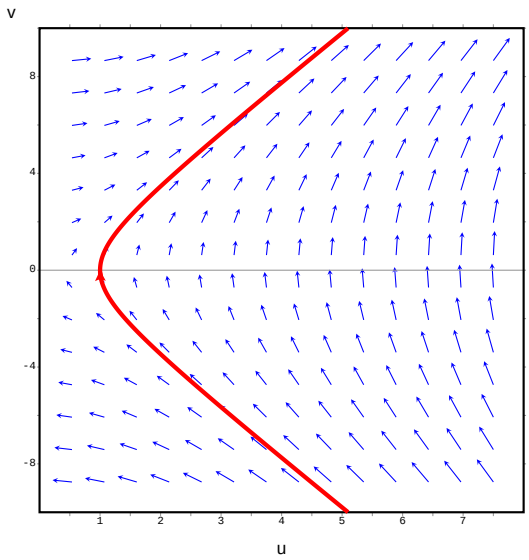

Figure 9: u'(t) vs. u(t) for $\mathbf{u(2) = 1,\ u'(2) = 0}$

In both of these plots we used the **Config** menu to increase the linewidth, and then clicked on **Replot**. We also cut and pasted the color **red** to be the second choice on the color cycle (instead of green) used in the plot versus the independent variable **t**. Note that no matter what you call your independent variable, it will always be called **t** on the plot of the dependent variables versus the independent variable.

3.3.5 Numerical Solution with 4th Order Runge-Kutta: rk

To use the fourth order Runge-Kutta numerical integrator **rk** for this example, we need to follow the procedure used in the previous section using **plotdf**, converting the second order ODE to a pair of first order ODE's.

The syntax for two first order ODE's with dependent variables **[u,v]** and independent variable **t** is

```
rk ( [ dudt, dvdt ], [u,v], [u0,v0], [t, t0, tmax, dt] )
```

which will produce the list of lists:

```
[ [t0, u0,v0],[t0+dt, u(t0+dt),v(t0+dt)], ..., [tmax, u(tmax),v(tmax)] ]
```

For our example, following our discussion in the previous section with **plotdf**, we use

```
points : rk ( [v, 4*u], [u, v], [1, 0], [t, 2, 3.6, 0.01] )
```

We again use the homemade function **fll** (see the preface) to look at the first element, the last element, and the length of various lists.

```
(%i1) fpprintprec:8$
(%i2) points : rk([v,4*u],[u,v],[1,0],[t,2,3.6,0.01])$
(%i3) %, fll;
(%o3)             [[2, 1, 0], [3.6, 12.286646, 24.491768], 161]
(%i4) uL : makelist([points[i][1],points[i][2]],i,1,length(points))$
(%i5) %, fll;
(%o5)                    [[2, 1], [3.6, 12.286646], 161]
```

```
(%i6) vL : makelist([points[i][1],points[i][3]],i,1,length(points))$
(%i7) %, fll;
(%o7)                    [[2, 0], [3.6, 24.491768], 161]
(%i8) plot2d([ [discrete,uL],[discrete,vL]],[x,1,5],
           [style,[lines,5]],[y,-1,24],[ylabel," "],
           [xlabel,"t"],[legend,"u(t)","v(t)"])$
```

which produces the plot

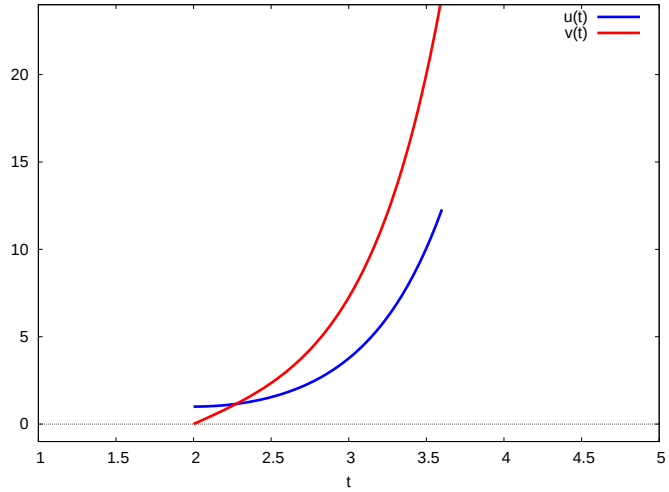

Figure 10: Runge-Kutta for $\mathbf{u(2) = 1,\ u'(2) = 0}$

Next we make a phase space plot of $\mathbf{v}$ versus $\mathbf{u}$ from the result of the Runge-Kutta integration.

```
(%i9) uvL : makelist([points[i][2],points[i][3]],i,1,length(points))$
(%i10) %, fll;
(%o10)                  [[1, 0], [12.286646, 24.491768], 161]
(%i11) plot2d( [ [discrete,uvL]],[x,0,13],[y,-1,25],
                [style,[lines,5]],[ylabel," "],
                [xlabel," v vs. u "])$
```

which produces the phase space plot

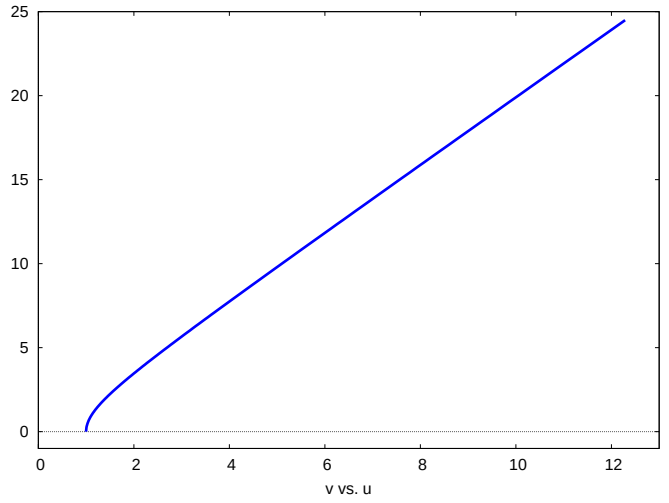

Figure 11: R-K Phase Space Plot for $\mathbf{u(2) = 1,\ u'(2) = 0}$

3.4 Examples of ODE Solutions

3.4.1 Ex. 1: Fall in Gravity with Air Friction: Terminal Velocity

Let's explore a problem posed by Patrick T. Tam (A Physicist's Guide to Mathematica, Academic Press, 1997, page 349).

> A small body falls downward with an initial velocity $\mathbf{v_0}$ from a height $\mathbf{h}$ near the surface of the earth. For low velocities (less than about $\mathbf{24\,m/s}$), the effect of air resistance may be approximated by a frictional force proportional to the velocity. Find the displacement and velocity of the body, and determine the terminal velocity. Plot the speed as a function of time for several initial velocities.

The net vector force $\mathbf{F}$ acting on the object is thus assumed to be the (constant) force of gravity and the (variable) force due to air friction, which is in a direction opposite to the direction of the velocity vector $\mathbf{v}$. We can then write Newton's Law of motion in the form

$$\mathbf{F} = m\,\mathbf{g} - b\,\mathbf{v} = m\,\frac{d\,\mathbf{v}}{d\,t} \tag{3.6}$$

In this vector equation, m is the mass in kg., $\mathbf{g}$ is a vector pointing downward with magnitude g, and b is a positive constant which depends on the size and shape of the object and on the viscosity of the air. The velocity vector $\mathbf{v}$ points down during the fall.

If we choose the y axis positive downward, with the point $y = 0$ the launch point, then the net y components of the force and Newton's Law of motion are:

$$F_y = m\,g - b\,v_y = m\,\frac{d\,v_y}{d\,t} \tag{3.7}$$

where g is the positive number $9.8\,m/s^2$ and since the velocity component $v_y > 0$ during the fall, the effects of gravity and air resistance are in competition.

We see that the rate of change of velocity will become zero at the instant that $m\,g - b\,v_y = 0$, or $v_y = m\,g/b$, and the downward velocity stops increasing at that moment, the "**terminal velocity**" having been attained.

While working with Maxima, we can simplify our notation and let $v_y \to v$ and $(b/m) \to a$ so both v and a represent positive numbers. We then use Maxima to solve the equation $d\,v/d\,t = g - a\,v$. The dimension of each term of this equation must evidently be the dimension of v/t, so a has dimension $1/t$.

```
(%i1) de : 'diff(v,t) - g + a*v;
                                dv
(%o1)                           -- + a v - g
                                dt
(%i2) gsoln : ode2(de,v,t);
                                          a t
                                - a t  g %e
(%o2)                    v = %e       (------- + %c)
                                           a
(%i3) de, gsoln, diff,ratsimp;
(%o3)                                    0
```

We then use **ic1** to get a solution such that $v = v0$ when $t = 0$.

```
(%i4) psoln : expand ( ic1 (gsoln,t = 0, v = v0 ) );
                                             - a t
                                - a t      g %e        g
(%o4)                    v = %e      v0 - --------- + -
                                             a         a
(%i5) vs : rhs(psoln);
                                           - a t
                              - a t      g %e        g
(%o5)                          %e      v0 - --------- + -
                                           a         a
```

For consistency, we must get the correct **terminal speed** for large t:

```
(%i6) assume(a>0)$
(%i7) limit( vs, t, inf );
                                        g
(%o7)                                   -
                                        a
```

which agrees with our analysis.

To make some plots, we can introduce a dimensionless time u with the replacement $t \rightarrow u = a\,t$, and a dimensionless speed w with the replacement $v \rightarrow w = a\,v/g$.

```
(%i8) expand(vs*a/g);
                              - a t
                         a %e      v0     - a t
(%o8)                    ------------ - %e      + 1
                              g
(%i9) %,[t=u/a,v0=w0*g/a];
                              - u         - u
(%o9)                       %e    w0 - %e    + 1
(%i10) ws : collectterms (%, exp (-u));
                                - u
(%o10)                        %e    (w0 - 1) + 1
```

As our dimensionless time u gets large, $\mathbf{ws} \rightarrow \mathbf{1}$, which is the value of the terminal speed in dimensionless units.

Let's now plot three cases, two cases with initial speed less than terminal speed and one case with initial speed greater than the terminal speed. (The use of dimensionless units for plots generates what are called "universal curves", since they are generally valid, no matter what the actual numbers are).

```
(%i11) plot2d([[discrete,[[0,1],[5,1]]],subst(w0=0,ws),subst(w0=0.6,ws),
                  subst(w0=1.5,ws)],[u,0,5],[y,0,2],
          [style,[lines,2,7],[lines,4,1],[lines,4,2],[lines,4,3]],
          [legend,"terminal speed", "w0 = 0", "w0 = 0.6", "w0 = 1.5"],
        [ylabel, " "],
        [xlabel, " dimensionless speed w vs dimensionless time u"])$
```

which produces:

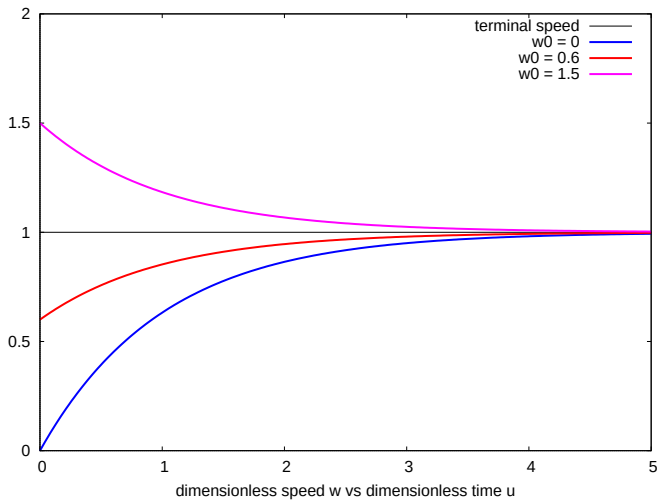

Figure 12: Dimensionless Speed Versus Dimensionless Time

An object thrown down with an initial speed greater than the terminal speed (as in the top curve) slows down until its speed is the terminal speed.

Thus far we have been only concerned with the relation between velocity and time. We can now focus on the implications for distance versus time. A dimensionless length $\mathbf{z}$ is $\mathbf{a^2\,y/g}$ and the relation $\mathbf{d\,y/d\,t = v}$ becomes $\mathbf{d\,z/d\,u = w}$, or $\mathbf{d\,z = w\,d\,u}$, which can be integrated over corresponding intervals: $\mathbf{z}$ over the interval $[\mathbf{0}, \mathbf{z_f}]$, and $\mathbf{u}$ over the interval $[\mathbf{0}, \mathbf{u_f}]$.

```
(%i12) integrate(1,z,0,zf) = integrate(ws,u,0,uf);
                              - uf             uf
(%o12)              zf = - %e      (w0 - uf %e    - 1) + w0 - 1
(%i13) zs : expand(rhs(%)),uf = u;
                              - u              - u
(%o13)                    - %e    w0 + w0 + %e    + u - 1
(%i14) zs, u=0;
(%o14)                                    0
```

(Remember the object is launched at $\mathbf{y = 0}$ which means at $\mathbf{z = 0}$). Let's make a plot of distance travelled vs time (dimensionless units) for the three cases considered above.

```
(%i15) plot2d([subst(w0=0,zs),subst(w0=0.6,zs),
          subst(w0=1.5,zs)],[u,0,1],[style,[lines,4,1],[lines,4,2],
                  [lines,4,3]], [legend,"w0 = 0", "w0 = 0.6", "w0 = 1.5"],
          [ylabel," "],
          [xlabel,"dimensionless distance z vs dimensionless time u"],
          [gnuplot_preamble,"set key top left;"])$
```

which produces:

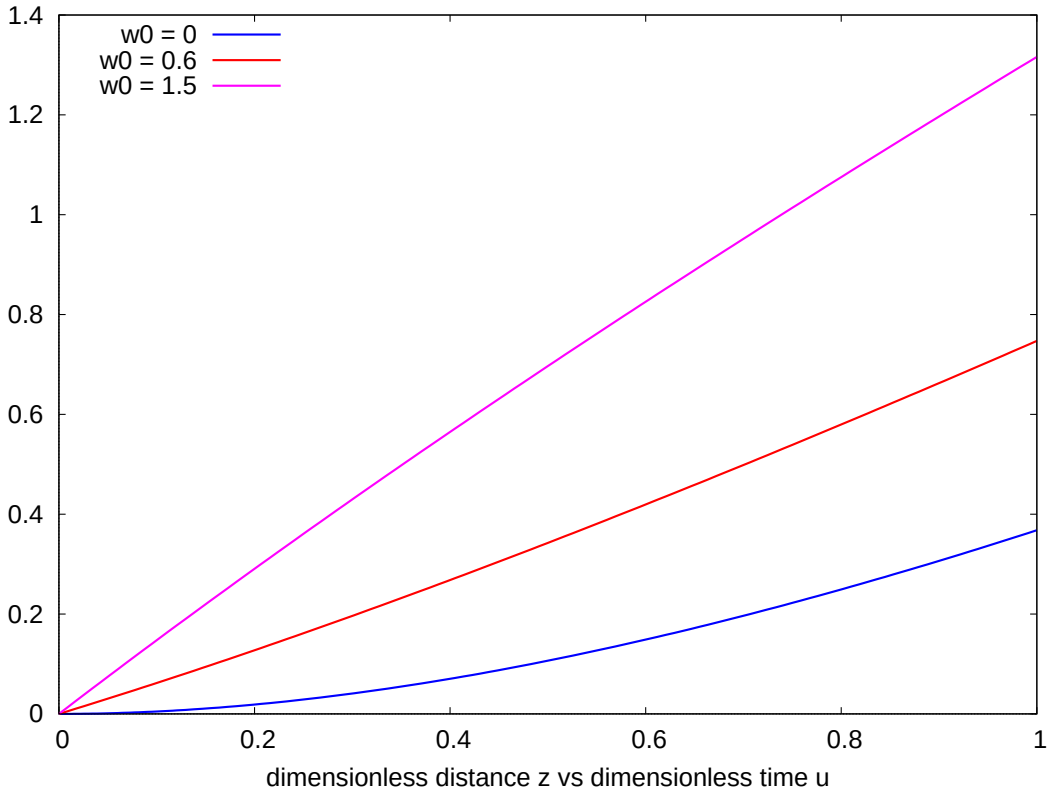

Figure 13: Dimensionless Distance Versus Dimensionless Time

3.4.2 Ex. 2: One Nonlinear First Order ODE

Let's solve

$$x^2\, y\, \frac{d\, y}{d\, x} = x\, y^2 + x^3 - 1 \tag{3.8}$$

for a solution such that when $x = 1,\ y = 1$.

```
(%i1) de : x^2*y*'diff(y,x) - x*y^2 - x^3 + 1;
                              2   dy      2    3
(%o1)                        x  y -- - x y  - x  + 1
                                  dx
(%i2) gsoln : ode2(de,y,x);
                               2      3
                         3 x y  - 6 x  log(x) - 2
(%o2)                    ------------------------ = %c
                                    3
                                 6 x
(%i3) psoln : ic1(gsoln,x=1,y=1);
                               2      3
                         3 x y  - 6 x  log(x) - 2   1
(%o3)                    ------------------------ = -
                                    3              6
                                 6 x
```

This implicitly determines y as a function of the independent variable x By inspection, we see that $x = 0$ is a singular point we should stay away from, so we assume from now on that $x \neq 0$.

To look at explicit solutions **y(x)** we use **solve**, which returns a list of two expressions depending on **x**. Since the implicit solution is a quadratic in **y**, we will get two solutions from **solve**, which we call **y1** and **y2**.

```
(%i4) [y1,y2] : map('rhs, solve(psoln,y) );
                     2           2   2           2           2   2
             sqrt(6 x  log(x) + x  + -)  sqrt(6 x  log(x) + x  + -)
                                    x                           x
(%o4)     [- -------------------------, -------------------------]
                       sqrt(3)                     sqrt(3)
(%i5) [y1,y2], x = 1, ratsimp;
(%o5)                              [- 1, 1]
(%i6) de, diff, y= y2, ratsimp;
(%o6)                                0
```

We see from the values at $x = 1$ that $y2$ is the particular solution we are looking for, and we have checked that $y2$ satisfies the original differential equation. From this example, we learn the lesson that **ic1** sometimes needs some help in finding the particular solution we are looking for.

Let's make a plot of the two solutions found.

```
(%i7) plot2d([y1,y2],[x,0.01,5],
        [style,[lines,5]],[ylabel, " Y "],
          [xlabel," X "] , [legend,"Y1", "Y2"],
         [gnuplot_preamble,"set key bottom center;"])$
```

which produces:

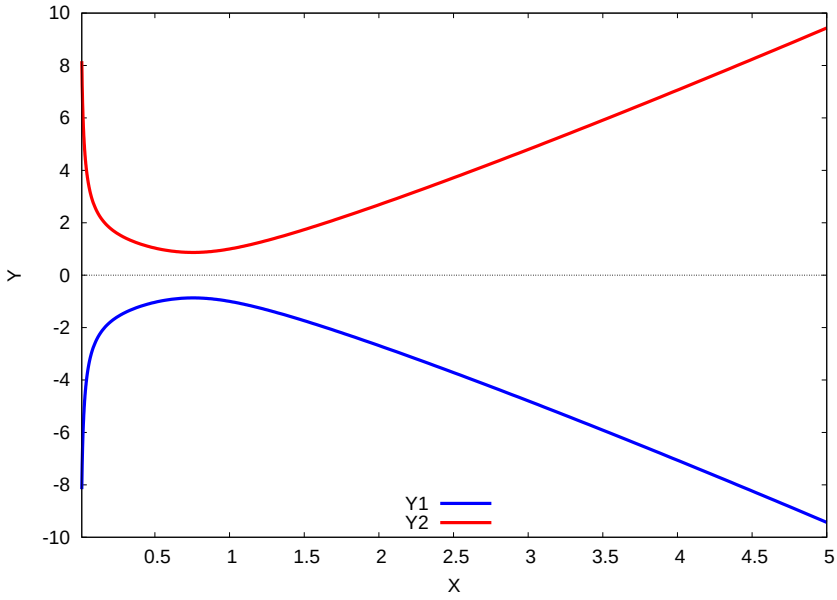

Figure 14: Positive X Solutions

3.4.3 Ex. 3: One First Order ODE Which is Not Linear in Y'

The differential equation to solve is

$$\left(\frac{\mathrm{d}\,x}{\mathrm{d}\,t}\right)^2 + 5\,x^2 = 8 \tag{3.9}$$

with the initial conditions $t = 0, \quad x = 0$.

```
(%i1) de: 'diff(x,t)^2 + 5*x^2 - 8;
                                  dx 2      2
(%o1)                            (--)  + 5 x  - 8
                                  dt
(%i2) ode2(de,x,t);
                                  dx 2      2
(%t2)                            (--)  + 5 x  - 8
                                  dt

                     first order equation not linear in y'

(%o2)                                  false
```

We see that direct use of **ode2** does not succeed. We can use **solve** to get equations which are linear in the first derivative, and then using **ode2** on each of the resulting linear ODE's.

```
(%i3) solve(de,'diff(x,t));
                   dx                  2   dx                  2
(%o3)             [-- = - sqrt(8 - 5 x ), -- = sqrt(8 - 5 x )]
                   dt                      dt
(%i4) ode2 ( %[2], x, t );
                                  5 x
                            asin(----------)
                                 2 sqrt(10)
(%o4)                       ---------------- = t + %c
                                sqrt(5)
```

```
(%i5) solve(%,x);
                      2 sqrt(10) sin(sqrt(5) t + sqrt(5) %c)
(%o5)           [x = --------------------------------------]
                                    5
(%i6) gsoln2 : %[1];
                      2 sqrt(10) sin(sqrt(5) t + sqrt(5) %c)
(%o6)            x = --------------------------------------
                                    5
(%i7) trigsimp ( ev (de,gsoln2,diff ) );
(%o7)                               0
(%i8) psoln : ic1 (gsoln2, t=0, x=0);
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
                           2 sqrt(10) sin(sqrt(5) t)
(%o8)                  x = -------------------------
                                     5
(%i9) xs : rhs(psoln);
                         2 sqrt(10) sin(sqrt(5) t)
(%o9)                    -------------------------
                                    5
(%i10) xs,t=0;
(%o10)                               0
```

We have selected only one of the linear ODE's to concentrate on here. We have shown that the solution satisfies the original differential equation and the given boundary condition.

3.4.4 Ex. 4: Linear Oscillator with Damping

The equation of motion for a particle of mass **m** executing one dimensional motion which is subject to a linear restoring force proportional to **|x|** and subject to a frictional force proportional to its speed is

$$\mathbf{m}\,\frac{\mathbf{d^2\,x}}{\mathbf{d\,t^2}} + \mathbf{b}\,\frac{\mathbf{d\,x}}{\mathbf{d\,t}} + \mathbf{k\,x} = \mathbf{0} \tag{3.10}$$

Dividing by the mass **m**, we note that if there were no damping, this motion would reduce to a linear oscillator with the angular frequency

$$\omega_0 = \left(\frac{\mathbf{k}}{\mathbf{m}}\right)^{1/2}. \tag{3.11}$$

In the presence of damping, we can define

$$\gamma = \frac{\mathbf{b}}{\mathbf{2\,m}} \tag{3.12}$$

and the equation of motion becomes

$$\frac{\mathbf{d^2\,x}}{\mathbf{d\,t^2}} + \mathbf{2}\,\gamma\,\frac{\mathbf{d\,x}}{\mathbf{d\,t}} + \omega_0^2\,\mathbf{x} = \mathbf{0} \tag{3.13}$$

In the presence of damping, there are now two natural time scales

$$\mathbf{t1} = \frac{\mathbf{1}}{\omega_0}, \qquad \mathbf{t2} = \frac{\mathbf{1}}{\gamma} \tag{3.14}$$

and we can introduce a dimensionless time $\theta = \omega_0\,\mathbf{t}$ and the dimensionless positive constant $\mathbf{a} = \gamma/\omega_0$, to get

$$\frac{\mathbf{d^2\,x}}{\mathbf{d}\,\theta^2} + \mathbf{2\,a}\,\frac{\mathbf{d\,x}}{\mathbf{d}\,\theta} + \mathbf{x} = \mathbf{0} \tag{3.15}$$

The "underdamped" case corresponds to $\gamma < \omega_0$, or $a < 1$ and results in damped oscillations around the final $x = 0$. The "critically damped" case corresponds to $a = 1$, and the "overdamped" case corresponds to $a > 1$. We specialize to solutions which have the initial conditions $\theta = 0, \quad x = 1, \quad dx/dt = 0 \Rightarrow dx/d\theta = 0$.

```
(%i1) de : 'diff(x,th,2) + 2*a*'diff(x,th) + x ;
                                 2
                               d x         dx
(%o1)                          ---- + 2 a --- + x
                                 2        dth
                               dth
(%i2) for i thru 3 do
          x[i] : rhs ( ic2 (ode2 (subst(a=i/2,de),x,th), th=0,x=1,diff(x,th)=0))$
(%i3) plot2d([x[1],x[2],x[3]],[th,0,10],
          [style,[lines,4]],[ylabel," "],
           [xlabel," Damped Linear Oscillator " ],
           [gnuplot_preamble,"set zeroaxis lw 2"],
           [legend,"a = 0.5","a = 1","a = 1.5"])$
```

which produces

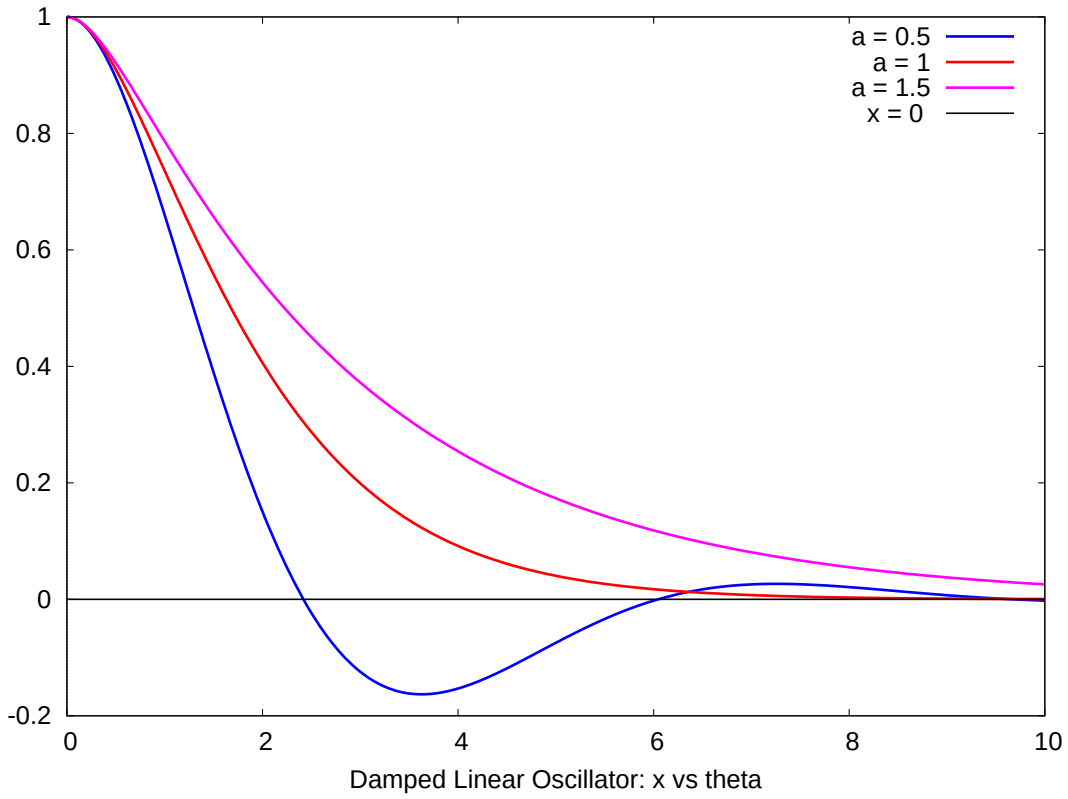

Figure 15: Damped Linear Oscillator

and illustrates why engineers seek the critical damping case, which brings the system to $x = 0$ most rapidly.

Now for a phase space plot with **dx/dth** versus **x**, drawn for the underdamped case:

```
(%i4) v1 : diff(x[1],th)$
(%i5) fpprintprec:8$
(%i6) [x5,v5] : [x[1],v1],th=5,numer;
(%o6)                        [- 0.0745906, 0.0879424]
(%i7) plot2d ( [ [parametric, x[1], v1, [th,0,10],[nticks,80]],
             [discrete,[[1,0]]], [discrete,[ [x5,v5] ] ] ],
         [x, -0.4, 1.2],[y,-0.8,0.2], [style,[lines,3,7],
                   [points,3,2,1],[points,3,6,1] ],
          [ylabel," "],[xlabel,"th = 0, x = 1, v = 0"],
          [legend," v vs x "," th = 0 "," th = 5 "])$
```

which shows

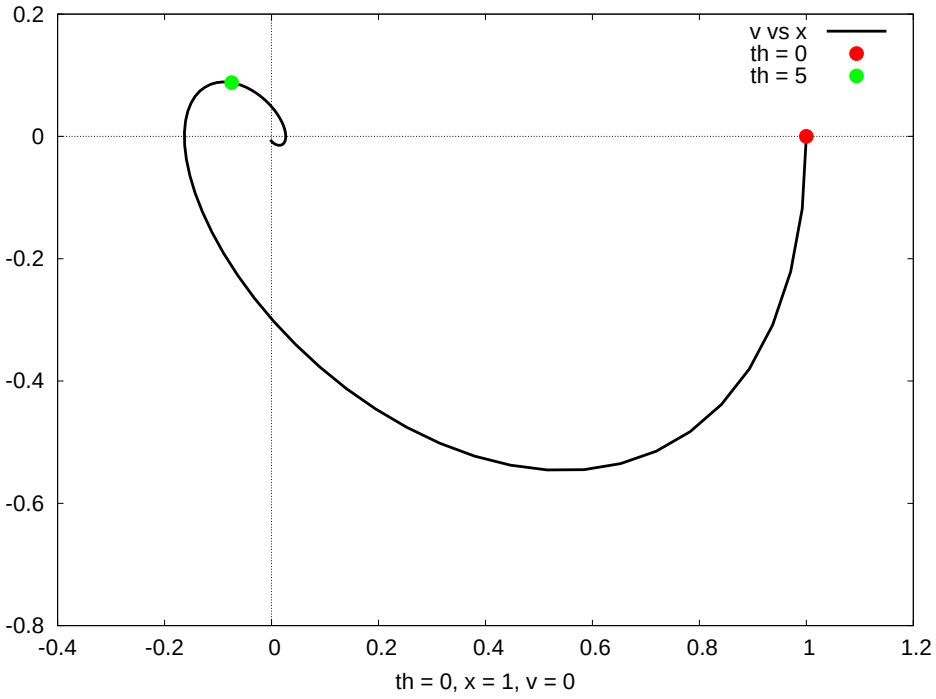

Figure 16: Underdamped Phase Space Plot

Using plotdf for the Damped Linear Oscillator

Let's use **plotdf** to show the phase space plot of our underdamped linear oscillator, using the syntax

```
plotdf ( [dudt, dvdt],[u,v], options...)
```

which requires that we convert our single second order ODE to an equivalent pair of first order ODE's. If we let $\mathbf{d\,x/d\,\theta = v}$, assume the dimensionless damping parameter $\mathbf{a = 1/2}$, we then have $\mathbf{d\,v/d\,\theta = -v - x}$, and we use the **plotdf** syntax

```
plotdf ( [dxdth, dvdth], [x, v], options... ).
```

One has to experiment with the number of steps, the step size, and the horizontal and vertical views. The $\mathbf{v}(\boldsymbol{\theta})$ values determine the vertical position and the $\mathbf{x}(\mathbf{boldsymbol}\theta)$ values determine the horizontal position of a point on the phase space plot curve. The symbols used for the horizontal and vertical ranges should correspond to the symbols used in the second argument (here **[x,v]**). Since we want to get a phase space plot which agrees with our work above, we require the trajectory begin at $\boldsymbol{\theta = 0}$, $\mathbf{x = 1}$, $\mathbf{v = 0}$, and we integrate forward in dimensionless time $\boldsymbol{\theta}$.

```
(%i8) plotdf([v,-v-x],[x,v],[trajectory_at,1,0],
           [direction,forward],[x,-0.4,1.2],[v,-0.6,0.2],
            [nsteps,400],[tstep,0.01])$
```

This will bring up the phase space plot **v** vs. **x**, and you can thicken the red curve by clicking the **Config** button (which brings up the **Plot Setup** panel), increasing the **linewidth** to 3, and then clicking **ok**. To actually see the thicker line, you must then click on the **Replot** button. This plot is

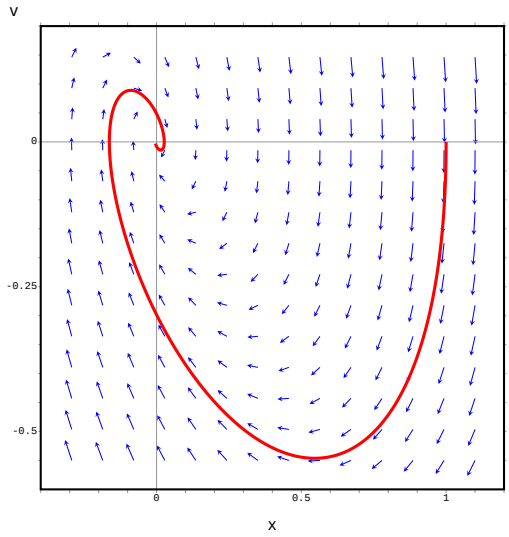

Figure 17: Underdamped Phase Space Plot Using plotdf

To see the separate curves $\mathbf{v}(\theta)$ and $\mathbf{x}(\theta)$, you can click on the **Plot Versus t** button. (The symbol **t** is simply a placeholder for the independent variable, which in our case is θ.) Again, you can change the linewidth and colors (we changed green to red) via the **Config** and **Replot** button process, which yields

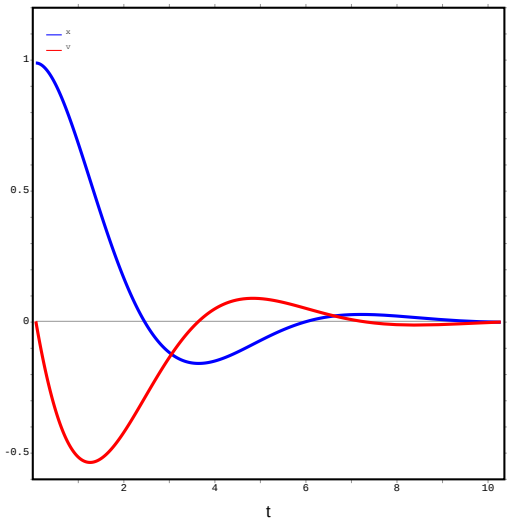

Figure 18: $\mathbf{x}(\theta)$ and $\mathbf{v}(\theta)$ Using plotdf

3.4.5 Ex. 5: Underdamped Linear Oscillator with Sinusoidal Driving Force

We extend our previous oscillator example by adding a sinusoidal driving force. The equation of motion is now

$$\mathbf{m}\,\frac{\mathbf{d^2\,x}}{\mathbf{d\,t^2}} + \mathbf{b}\,\frac{\mathbf{d\,x}}{\mathbf{d\,t}} + \mathbf{k\,x} = \mathbf{A}\cos(\boldsymbol{\omega}\,\mathbf{t}) \tag{3.16}$$

We again divide by the mass $\mathbf{m}$ and let

$$\boldsymbol{\omega_0} = \left(\frac{\mathbf{k}}{\mathbf{m}}\right)^{\mathbf{1/2}}. \tag{3.17}$$

As before, we define

$$\boldsymbol{\gamma} = \frac{\mathbf{b}}{\mathbf{2\,m}}. \tag{3.18}$$

Finally, let $\mathbf{B} = \mathbf{A}/\mathbf{m}$. The equation of motion becomes

$$\frac{\mathbf{d^2\,x}}{\mathbf{d\,t^2}} + \mathbf{2}\,\boldsymbol{\gamma}\,\frac{\mathbf{d\,x}}{\mathbf{d\,t}} + \boldsymbol{\omega_0^2}\,\mathbf{x} = \mathbf{B}\cos(\boldsymbol{\omega}\,\mathbf{t}) \tag{3.19}$$

There are now three natural time scales

$$\mathbf{t1} = \frac{\mathbf{1}}{\boldsymbol{\omega_0}}, \quad \mathbf{t2} = \frac{\mathbf{1}}{\boldsymbol{\gamma}}, \quad \mathbf{t3} = \frac{\mathbf{1}}{\boldsymbol{\omega}} \tag{3.20}$$

and we can introduce a dimensionless time $\boldsymbol{\theta} = \boldsymbol{\omega_0}\,\mathbf{t}$, the dimensionless positive damping constant $\mathbf{a} = \boldsymbol{\gamma}/\boldsymbol{\omega_0}$, the dimensionless oscillator displacement $\mathbf{y} = \mathbf{x}/\mathbf{B}$, and the dimensionless driving angular frequency $\mathbf{q} = \boldsymbol{\omega}/\boldsymbol{\omega_0}$ to get

$$\frac{\mathbf{d^2\,y}}{\mathbf{d\,\theta^2}} + \mathbf{2\,a}\,\frac{\mathbf{d\,y}}{\mathbf{d\,\theta}} + \mathbf{y} = \mathbf{cos}(\boldsymbol{q\,\theta}) \tag{3.21}$$

The "underdamped" case corresponds to $\boldsymbol{\gamma} < \boldsymbol{\omega_0}$, or $\mathbf{a} < \mathbf{1}$, and we specialize to the case $\mathbf{a} = \mathbf{1/2}$.

```
(%i1) de : 'diff(y,th,2) + 'diff(y,th) + y - cos(q*th);
                            2
                           d y    dy
(%o1)                      ---- + --- + y - cos(q th)
                             2    dth
                           dth
(%i2) gsoln : ode2(de,y,th);
                                2
          q sin(q th) + (1 - q ) cos(q th)
(%o2) y = --------------------------------
                     4    2
                    q  - q  + 1
                              - th/2          sqrt(3) th             sqrt(3) th
                         + %e       (%k1 sin(----------) + %k2 cos(----------))
                                                2                      2
(%i3) psoln : ic2(gsoln,th=0,y=1,diff(y,th)=0);
                                2
          q sin(q th) + (1 - q ) cos(q th)
(%o3) y = --------------------------------
                     4    2
                    q  - q  + 1
                              4      2      sqrt(3) th        4     sqrt(3) th
                           (q  - 2 q ) sin(----------)      q  cos(----------)
                 - th/2                        2                      2
           + %e       (------------------------------- + ------------------)
                                   4            2                 4    2
                        sqrt(3) q  - sqrt(3) q  + sqrt(3)      q  - q  + 1
```

We now specialize to a high (dimensionless) driving angular frequency case, $\mathbf{q = 4}$, which means that we are assuming that the actual driving angular frequency is four times as large as the natural angular frequency of this oscillator.

```
(%i4) ys : subst(q=4,rhs(psoln));
                        sqrt(3) th             sqrt(3) th
                 224 sin(----------)    256 cos(----------)
        - th/2             2                      2
(%o4) %e       (------------------- + --------------------)
                     241 sqrt(3)                241
                                                  4 sin(4 th) - 15 cos(4 th)
                                               + -------------------------
                                                             241
(%i5) vs : diff(ys,th)$
```

We now plot both the dimensionless oscillator amplitude and the dimensionless oscillator velocity on the same plot.

```
(%i6) plot2d([ys,vs],[th,0,12],
        [nticks,100],
        [style,[lines,5]],
        [legend," Y "," V "],
        [xlabel," dimensionless Y and V vs. theta"])$
```

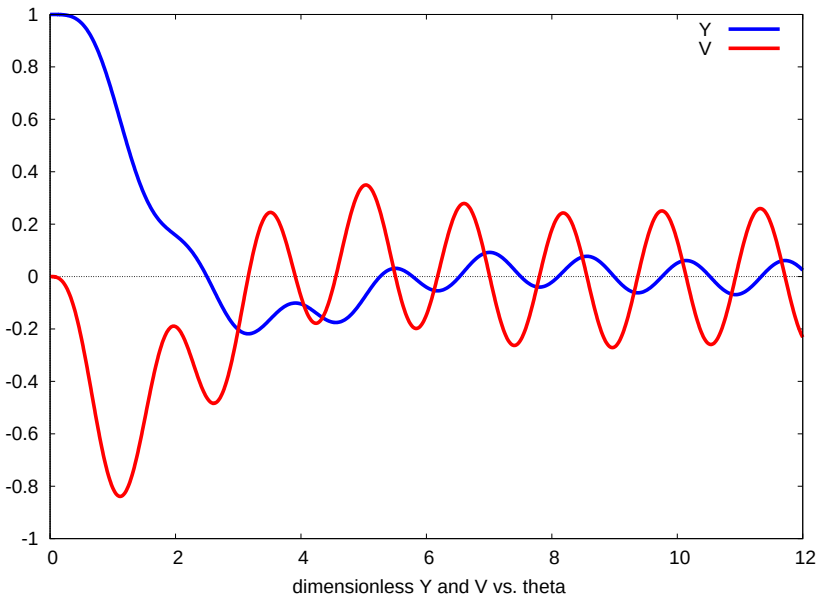

Figure 19: Dimensionless Y and V versus Dimensionless Time θ

We see that the driving force soon dominates the motion of the underdamped linear oscillator, which is forced to oscillate at the driving frequency. This dominance evidently has nothing to do with the actual strength **A newtons** of the peak driving force, since we are solving for a dimensionless oscillator amplitude, and we get the same qualitative curve no matter what the size of **A** is.

We next make a phase space plot for the early "capture" part of the motion of this system. (Note that **plotdf** cannot numerically integrate this differential equation because of the explicit appearance of the dependent variable.)

```
(%i7) plot2d([parametric,ys,vs,[th,0,8]],
           [style,[lines,5]],[nticks,100],
            [xlabel," V (vert) vs. Y (hor) "])$
```

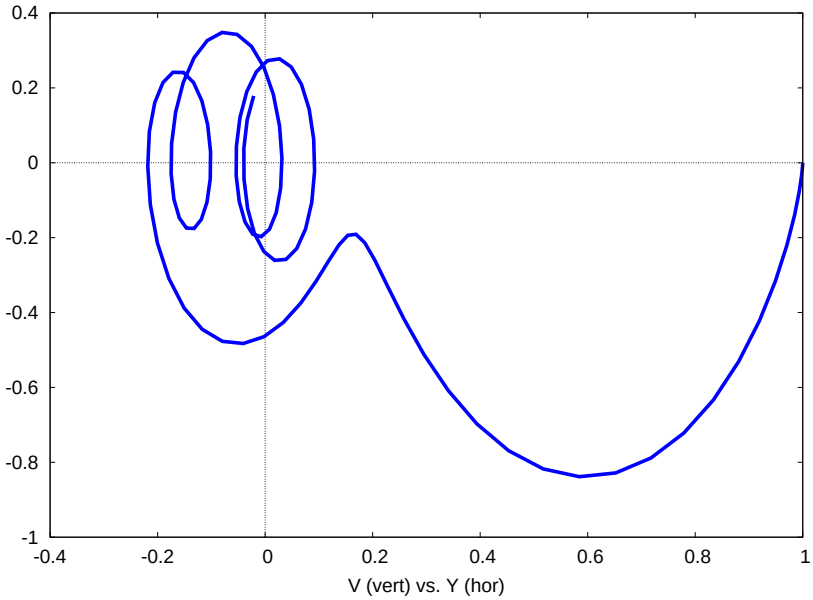

Figure 20: Dimensionless V Versus Dimensionless Y: Early History

We see the phase space plot being driven to regular oscillations about **y = 0** and **v = 0**.

3.4.6 Ex. 6: Regular and Chaotic Motion of a Driven Damped Planar Pendulum

The motion is pure rotation in a fixed plane (one degree of freedom), and if the pendulum is a simple pendulum with all the mass **m** concentrated at the end of a weightless support of length **L**, then the moment of inertia about the support point is $\mathbf{I = m\,L^2}$, and the angular acceleration is $\boldsymbol{\alpha}$, and rotational dynamics implies the equation of motion

$$\mathbf{I}\,\boldsymbol{\alpha} = \mathbf{m\,L^2}\,\frac{\mathbf{d^2}\boldsymbol{\theta}}{\mathbf{d\,t^2}} = \boldsymbol{\tau}_\mathbf{z} = -\mathbf{m\,g\,L}\sin\boldsymbol{\theta} - \mathbf{c}\,\frac{\mathbf{d}\,\boldsymbol{\theta}}{\mathbf{d\,t}} + \mathbf{A}\cos(\boldsymbol{\omega}_\mathbf{d}\,\mathbf{t}) \tag{3.22}$$

We introduce a dimensionless time $\boldsymbol{\tau} = \boldsymbol{\omega}_\mathbf{0}\,\mathbf{t}$ and a dimensionless driving angular frequency $\boldsymbol{\omega} = \boldsymbol{\omega}_\mathbf{d}/\boldsymbol{\omega}_\mathbf{0}$, where $\boldsymbol{\omega}_\mathbf{0}^\mathbf{2} = \mathbf{g}/\mathbf{L}$, to get the equation of motion

$$\frac{\mathbf{d^2}\boldsymbol{\theta}}{\mathbf{d}\,\boldsymbol{\tau}^\mathbf{2}} = -\sin\boldsymbol{\theta} - \mathbf{a}\,\frac{\mathbf{d}\,\boldsymbol{\theta}}{\mathbf{d}\,\boldsymbol{\tau}} + \mathbf{b}\cos(\boldsymbol{\omega}\,\boldsymbol{\tau}) \tag{3.23}$$

To simplify the notation for our exploration of this differential equation, we make the replacements $\boldsymbol{\theta} \to \mathbf{u}$, $\boldsymbol{\tau} \to \mathbf{t}$, and $\boldsymbol{\omega} \to \mathbf{w}$ (parameters **a**, **b**, and **w** are dimensionless) to work with the differential equation:

$$\frac{\mathbf{d^2\,u}}{\mathbf{d\,t^2}} = -\sin\mathbf{u} - \mathbf{a}\,\frac{\mathbf{d\,u}}{\mathbf{d\,t}} + \mathbf{b}\cos(\mathbf{w\,t}) \tag{3.24}$$

where now both **t** and **u** are dimensionless, with the measure of **u** being radians, and the physical values of the pendulum angle being limited to the range $-\boldsymbol{\pi} \le \mathbf{u} \le \boldsymbol{\pi}$, both extremes being the "flip-over-point" at the top of the motion.

We will use both **plotdf** and **rk** to explore this system, with

$$\frac{\mathbf{d\,u}}{\mathbf{d\,t}} = \mathbf{v}, \quad \frac{\mathbf{d\,v}}{\mathbf{d\,t}} = -\sin\mathbf{u} - \mathbf{a\,v} + \mathbf{b}\cos(\mathbf{w\,t}) \tag{3.25}$$

3.4.7 Free Oscillation Case

Using **plotdf**, the phase space plot for NO friction and NO driving torque is

```
(%i1) plotdf([v,-sin(u)],[u,v],[trajectory_at,float(2*%pi/3),0],
             [direction,forward],[u,-2.5,2.5],[v,-2.5,2.5],
              [tstep, 0.01],[nsteps,600])$
```

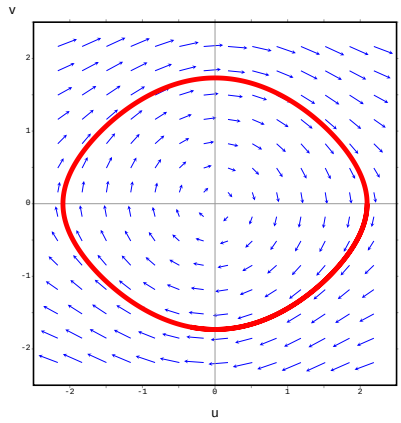

Figure 21: No Friction, No Driving Torque: V Versus Angle U

and now we use the **Plot Versus t** button of **plotdf** to show the angle **u radians** and the dimensionless rate of change of angle **v**

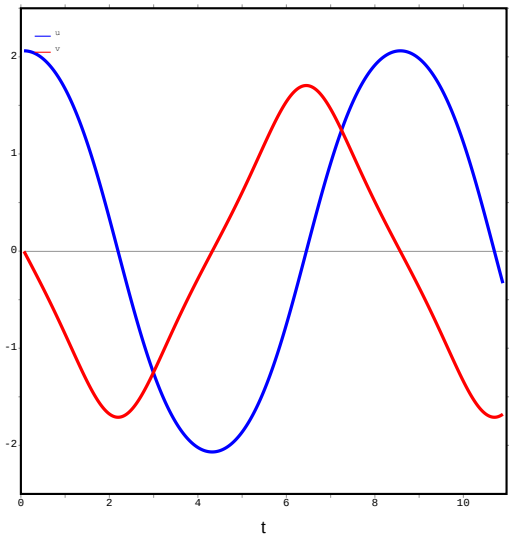

Figure 22: No Friction, No Driving Torque: Angle U [blue] and V [red]

3.4.8 Damped Oscillation Case

We now include some damping with $\mathbf{a} = \mathbf{1}/\mathbf{2}$.

```
(%i2)  plotdf([v,-sin(u)-0.5*v],[u,v],[trajectory_at,float(2*%pi/3),0],
            [direction,forward],[u,-1,2.5],[v,-1.5,1],
             [tstep, 0.01],[nsteps,450])$
```

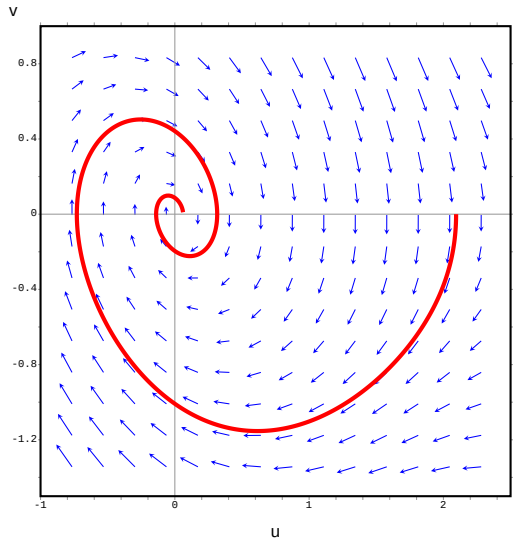

Figure 23: With Friction, but No Driving Force: V Versus Angle U

and now we use the **Plot Versus t** button of **plotdf** to show the angle **u radians** and the dimensionless rate of change of angle **y** for the friction present case.

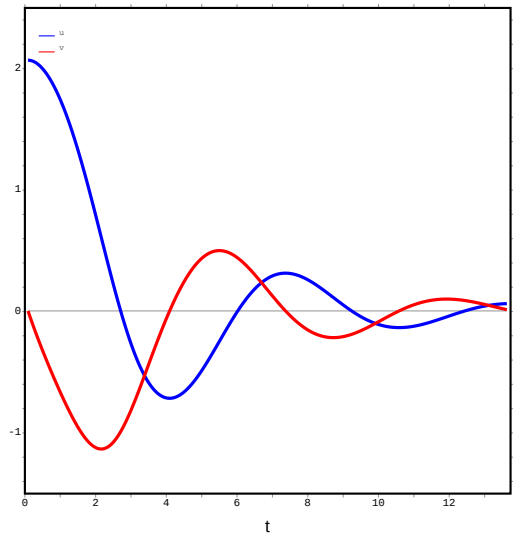

Figure 24: With Friction, but No Driving Force: Angle U [blue] and V [red]

3.4.9 Including a Sinusoidal Driving Torque

We now use the Runge-Kutta function **rk** to integrate the differential equation set forward in time for **ncycles**, which is the same as setting the final dimensionless **tmax** equal to **ncycles*2*%pi/w**, or **ncycles*T**, where we can call **T** the dimensionless period defined by the dimensionless angular frequency **w**. The physical meaning of **T** is the ratio of the period of the driving torque to the period of unforced and undamped small oscillations of the free simple pendulum.

For simplicity of exposition, we will call **t** the "time" and **T** the "period". We again use our homemade function **fll** described in the preface.

One cycle (period) of time is divided into **nsteps** subdivisions, so **dt = T/nsteps**.

For both the regular and chaotic parameter cases, we have used the same parameters as used in **Mathematica in Theoretical Physics**, by Gerd Baumann, Springer/Telos, 1996, pages 46 - 53.

3.4.10 Regular Motion Parameters Case

We find regular motion of this driven system with **a = 0.2, b = 0.52, and w = 0.694**, and with **u0 = 0.8 rad**, and **v0 = 0.8 rad/unit-time**.

```
(%i1) fpprintprec:8$
(%i2) (nsteps : 31, ncycles : 30, a : 0.2, b : 0.52, w : 0.694)$
(%i3) [dudt : v, dvdt : -sin(u) - a*v + b*cos(w*t),
          T : float(2*%pi/w ) ];
(%o3)        [v, - 0.2 v - sin(u) + 0.52 cos(0.694 t), 9.0535811]
(%i4) [dt : T/nsteps, tmax : ncycles*T ];
(%o4)                         [0.292051, 271.60743]
(%i5) tuvL : rk ([dudt,dvdt],[u,v],[0.8,0.8],[t,0,tmax, dt])$
(%i6) %, fll;
(%o6)      [[0, 0.8, 0.8], [271.60743, - 55.167003, 1.1281164], 931]
(%i7) 930*dt;
(%o7)                              271.60743
```

Plot of u(t) and v(t)

Plot of u(t) and v(t) against t

```
(%i8) tuL : makelist ([tuvL[i][1],tuvL[i][2]],i,1,length(tuvL))$
(%i9) %, fll;
(%o9)               [[0, 0.8], [271.60743, - 55.167003], 931]
(%i10) tvL : makelist ([tuvL[i][1],tuvL[i][3]],i,1,length(tuvL))$
(%i11) %, fll;
(%o11)               [[0, 0.8], [271.60743, 1.1281164], 931]
(%i12) plot2d([ [discrete,tuL], [discrete,tvL]],[x,0,280],
          [style,[lines,3]],[xlabel,"t"],
           [legend, "u", "v"],
             [gnuplot_preamble,"set key bottom left;"])$
```

which produces

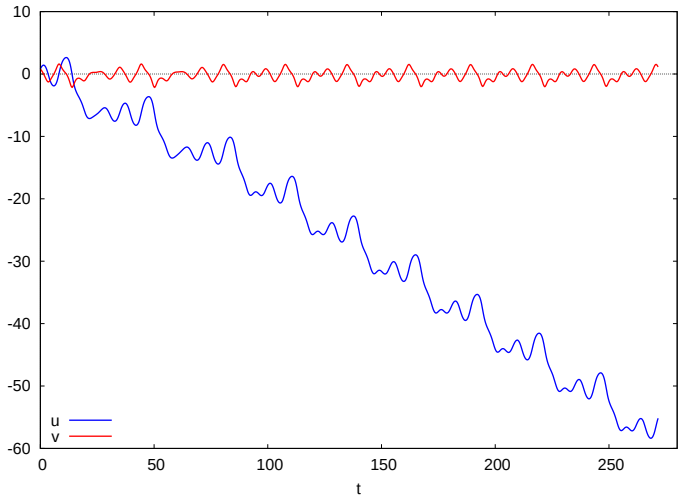

Figure 25: Angle u(t) and v(t)

The above plot shows nine flips of the pendulum at the top:
the first passage over the top at **`u = -3 pi/2 = -4.7 rad`**,
the second passage over the top at **`u = -7 pi/2 = -11 rad`**,
and so on.

Phase Space Plot

We next construct a phase space plot.

```
(%i13) uvL : makelist ([tuvL[i][2],tuvL[i][3]],i,1,length(tuvL))$
(%i14) %, fll;
(%o14)              [[0.8, 0.8], [- 55.167003, 1.1281164], 931]
(%i15) plot2d ( [discrete,uvL],[x,-60,5],[y,-5,5],
                  [style,[lines,3]],
             [ylabel," "],[xlabel," v vs u "] )$
```

which produces (note that we include the early points which are more heavily influenced by the initial conditions):

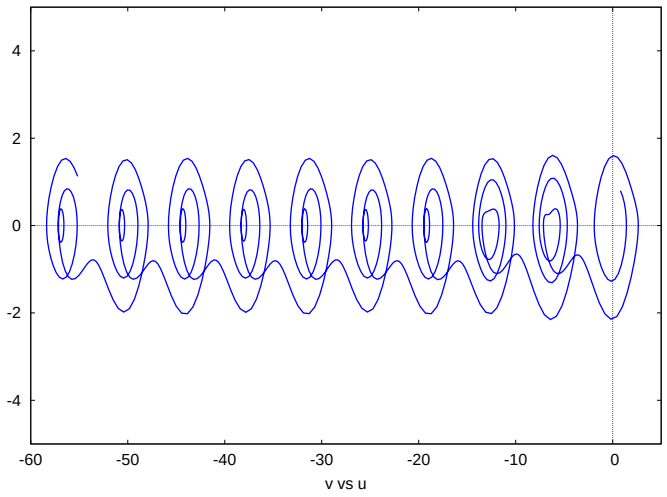

Figure 26: Non-Reduced Phase Space Plot

Reduced Phase Space Plot

Let's define a Maxima function **reduce** which brings **u** back to the interval **(-pi, pi)** and then make a reduced phase space plot. Since this is a strictly numerical task, we can simplify Maxima's efforts by defining a floating point number **pi** once and for all, and simply work with that definition. You can see the origin of our definition of **reduce** in the manual's entry on Maxima's modulus function **mod**.

```
(%i16) pi : float(%pi);
(%o16)                              3.1415927
(%i17) reduce(yy) := pi - mod (pi - yy,2*pi)$
(%i18) float( [-7*%pi/2,-3*%pi/2 ,3*%pi/2, 7*%pi/2] );
(%o18)          [- 10.995574, - 4.712389, 4.712389, 10.995574]
(%i19) map('reduce, % );
(%o19)         [1.5707963, 1.5707963, - 1.5707963, - 1.5707963]
(%i20) uvL_red : makelist ( [ reduce( tuvL[i][2]),
                        tuvL[i][3]],i,1,length(tuvL))$
(%i21) %, fll;
(%o21)              [[0.8, 0.8], [1.3816647, 1.1281164], 931]
```

To make a reduced phase space plot with our reduced regular motion points, we will only use the last two thirds of the pairs **(u,v)**. This will then show the part of the motion which has been "captured" by the driving torque and shows little influence of the initial conditions.

We use the Maxima function **rest (list, n)** which returns **list** with its first **n** elements removed if **n** is positive. Thus we use **rest (list, num/3)** to get the last two thirds.

```
(%i22) uvL_regular : rest (uvL_red, round(length (uvL_red)/3) )$
(%i23) %, fll;
(%o23)      [[0.787059, - 1.2368529], [1.3816647, 1.1281164], 621]
(%i24) plot2d ( [discrete,uvL_regular],[x,-3.2,3.2],[y,-3.2,3.2],
                  [style,[lines,2]],
             [ylabel," "],[xlabel,"reduced phase space v vs u "] )$
```

which produces

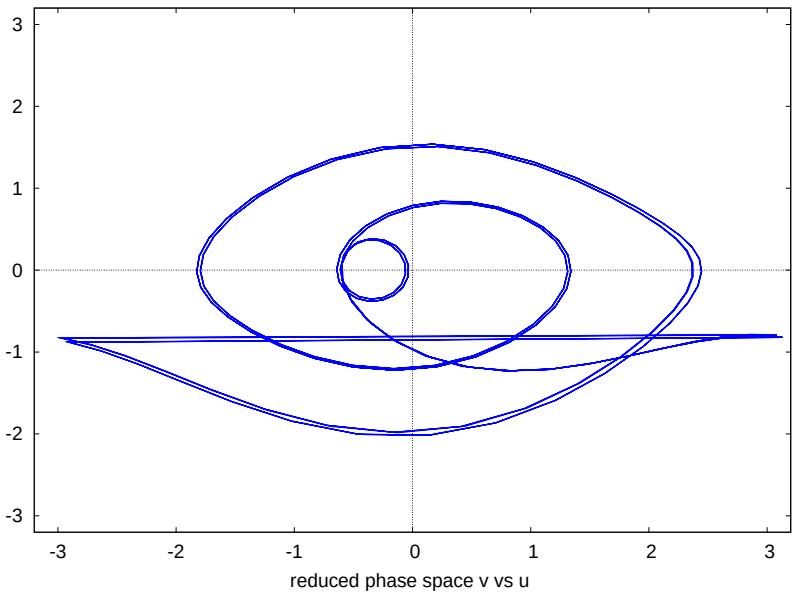

Figure 27: Reduced Phase Space Plot of Regular Motion Points

Poincare Plot

We next construct a Poincare plot of the regular (reduced) phase space points by using a "stroboscopic view" of this phase space, displaying only phase space points which correspond to times separated by the driving period **T**. We select **(u,v)** pairs which correspond to intervals of time **n*T**, where **n = 10, 11, ..., 30** which will give us **21** phase space points for our plot (this is roughly the same as taking the last two thirds of the points).

The time **t = 30*T** corresponds to **t = 30*31*dt** = **930*dt** which is the time associated with element **931**, the last element) **of uvL_red**. The value of **j** used to select the last Poincare point is the solution of the equation **1 + 10*nsteps + j*nsteps = 1 + ncycles*nsteps**, which for this case is equivalent to **311 + j*31 = 931**.

```
(%i25) solve(311 + j*31 = 931);
(%o25)                              [j = 20]
(%i26) pL : makelist (1+10*nsteps + j*nsteps, j, 0, 20);
(%o26) [311, 342, 373, 404, 435, 466, 497, 528, 559, 590, 621, 652, 683, 714,
                                                 745, 776, 807, 838, 869, 900, 931]
(%i27) length(pL);
(%o27)                               21
(%i28) poincareL : makelist (uvL_red[i], i, pL)$
(%i29) %,fll;
(%o29)       [[0.787059, - 1.2368529], [1.3816647, 1.1281164], 21]
(%i30) plot2d ( [discrete,poincareL],[x,-0.5,2],[y,-1.5,1.5],
                   [style,[points,1,1,1 ]],
             [ylabel," "],[xlabel," Poincare Section v vs u "] )$
```

which produces the plot

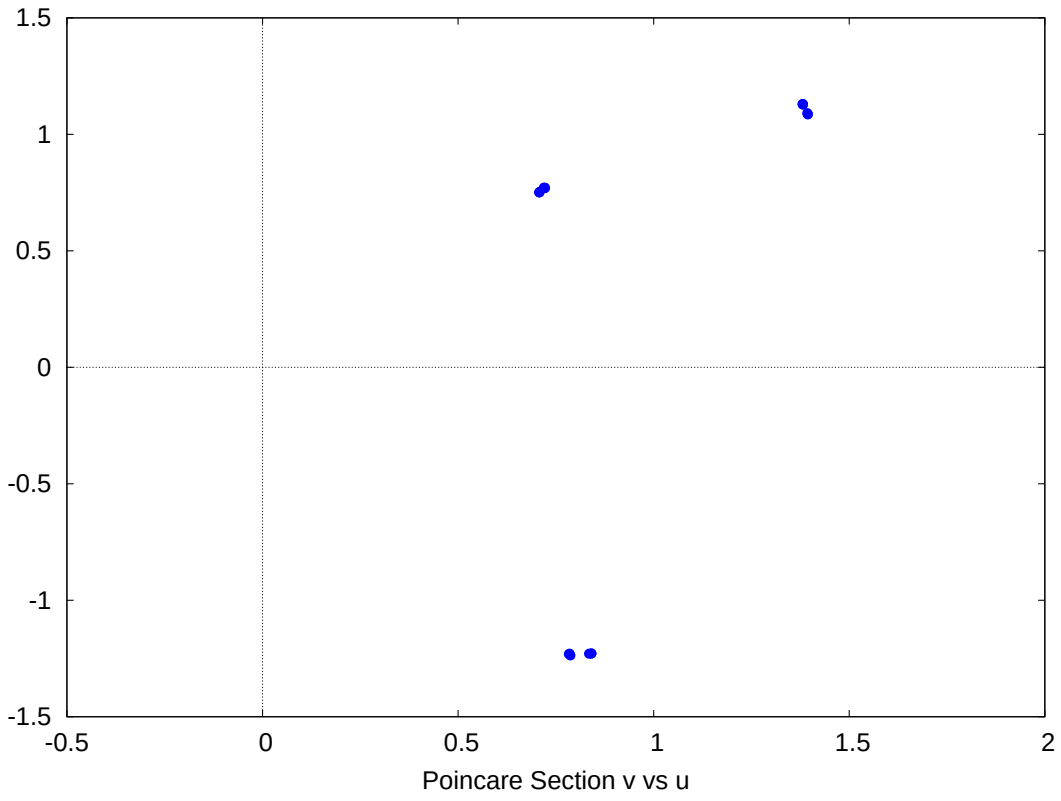

Figure 28: Reduced Phase Space Plot of Regular Motion Points

For this regular motion parameters case, the Poincare plot shows the phase space point coming back to one of three general locations in phase space at times separated by the period **T**.

3.4.11 Chaotic Motion Parameters Case.

To exhibit an example of chaotic motion for this system, we use the same initial conditions for **u** and **v**, but use the parameter set **a = 1/2, b = 1.15, w = 2/3**.

```
(%i1) fpprintprec:8$
(%i2) (nsteps : 31, ncycles : 240, a : 1/2, b : 1.15, w : 2/3)$
(%i3) [dudt : v, dvdt : -sin(u) - a*v + b*cos(w*t),
          T : float(2*%pi/w ) ];
                        v                      2 t
(%o3)             [v, - - - sin(u) + 1.15 cos(---), 9.424778]
                        2                      3
(%i4) [dt : T/nsteps, tmax : ncycles*T ];
(%o4)                          [0.304025, 2261.9467]
(%i5) tuvL : rk ([dudt,dvdt],[u,v],[0.8,0.8],[t,0,tmax, dt])$
(%i6) %, fll;
(%o6)       [[0, 0.8, 0.8], [2261.9467, 26.374502, 0.937008], 7441]
(%i7) dt*( last(%) - 1 );
(%o7)                                 2261.9467
(%i8) tuL : makelist ([tuvL[i][1],tuvL[i][2]],i,1,length(tuvL))$
(%i9) %, fll;
(%o9)                [[0, 0.8], [2261.9467, 26.374502], 7441]
(%i10) tvL : makelist ([tuvL[i][1],tuvL[i][3]],i,1,length(tuvL))$
(%i11) %, fll;
(%o11)               [[0, 0.8], [2261.9467, 0.937008], 7441]
(%i12) plot2d([ [discrete,tuL], [discrete,tvL]],[x,0,2000],
               [y,-15,30],
          [style,[lines,2]],[xlabel,"t"], [ylabel, " "],
           [legend, "u","v" ] ,[gnuplot_preamble,"set key bottom;"])$
```

which produces a plot of **u(t)** and **v(t)** over **0 <= t <= 2000**:

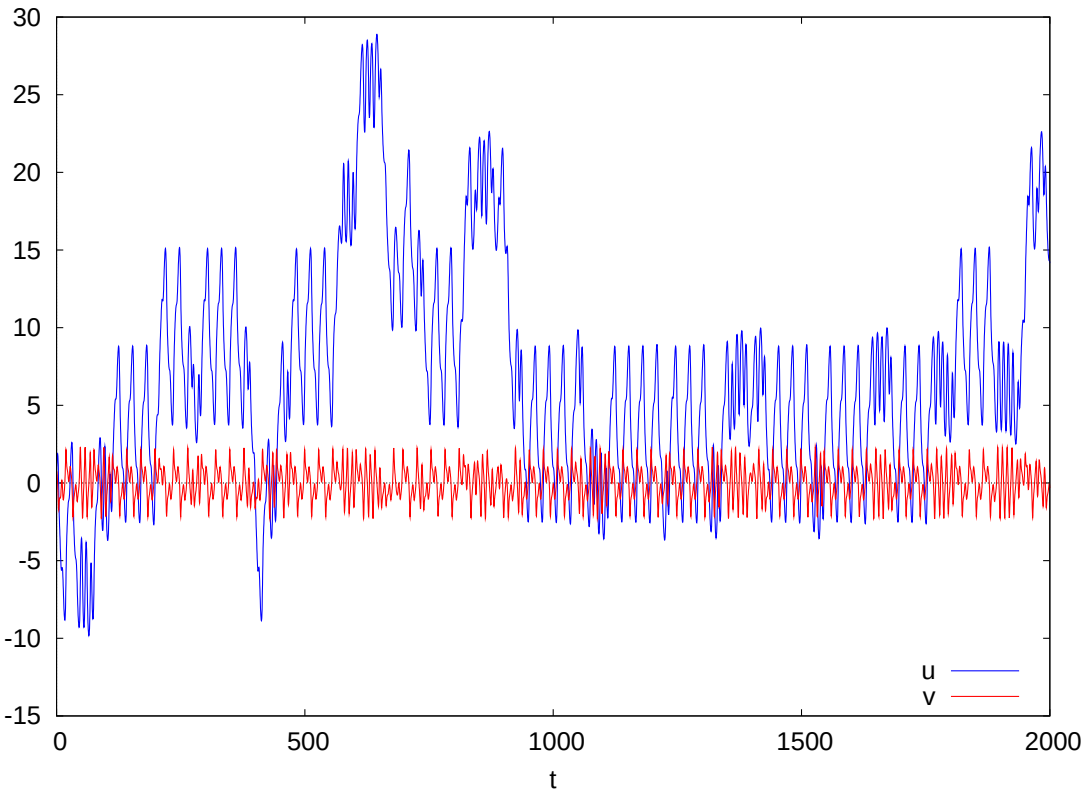

Figure 29: Angle u(t), and v(t) for Chaotic Parameters Choice

Phase Space Plot

We next construct a non-reduced phase space plot, but show only the first **2000** reduced phase space points.

```
(%i13) uvL : makelist ([tuvL[i][2],tuvL[i][3]],i,1,length(tuvL))$
(%i14) %, fll;
(%o14)              [[0.8, 0.8], [26.374502, 0.937008], 7441]
(%i15) uvL_first : rest(uvL, -5441)$
(%i16) %, fll;
(%o16)              [[0.8, 0.8], [23.492001, 0.299988], 2000]
(%i17) plot2d ( [discrete,uvL_first],[x,-12,30],[y,-3,3],
                    [style,[points,1,1,1]],
             [ylabel," "],[xlabel," v vs u "])$
```

which produces

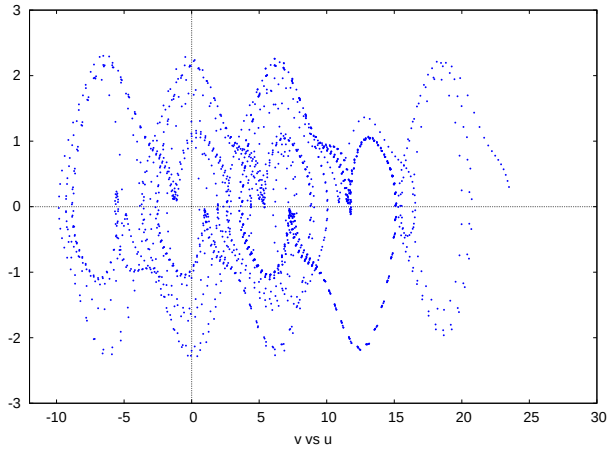

Figure 30: non-reduced phase space plot using first 2000 points

If we use the **discrete** default **style** option **lines** instead of **points**,

```
(%i18) plot2d ( [discrete,uvL_first],[x,-12,30],[y,-3,3],
             [ylabel," "],[xlabel," v vs u "])$
```

we get the non-reduced phase space plot drawn with lines between the points:

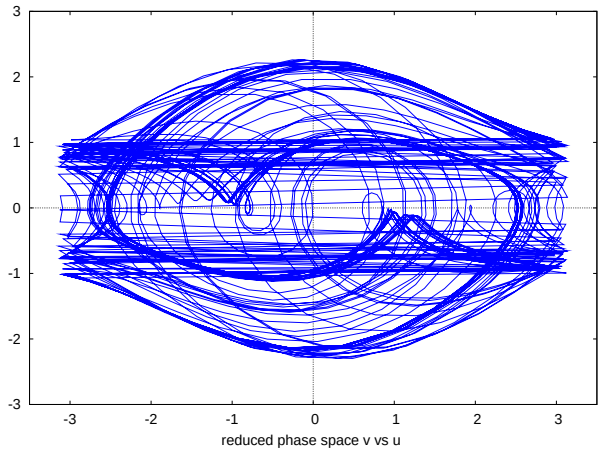

Figure 31: non-reduced phase space plot using first 2000 points

Reduced Phase Space Plot

We now construct the reduced phase space points as in the regular motion case and then omit the first **400**.

```
(%i19) pi : float(%pi);
(%o19)                              3.1415927
(%i20) reduce(yy) := pi - mod (pi - yy,2*pi)$
(%i21) uvL_red : makelist ( [ reduce( first( uvL[i] )),
                        second( uvL[i] ) ],i,1,length(tuvL))$
(%i22) %, fll;
(%o22)              [[0.8, 0.8], [1.2417605, 0.937008], 7441]
(%i23) uvL_cut : rest(uvL_red, 400)$
(%i24) %, fll;
(%o24)         [[0.25464, 1.0166641], [1.2417605, 0.937008], 7041]
```

We have discarded the first **400** reduced phase space points in defining **uvL_cut**. If we now only plot the first **1000** of the points retained in **uvL_cut**:

```
(%i25) uvL_first : rest (uvL_cut, -6041)$
(%i26) %, fll;
(%o26)         [[0.25464, 1.0166641], [2.2678603, 0.608686], 1000]
(%i27) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
                  [style,[points,1,1,1]],
              [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

we get the plot

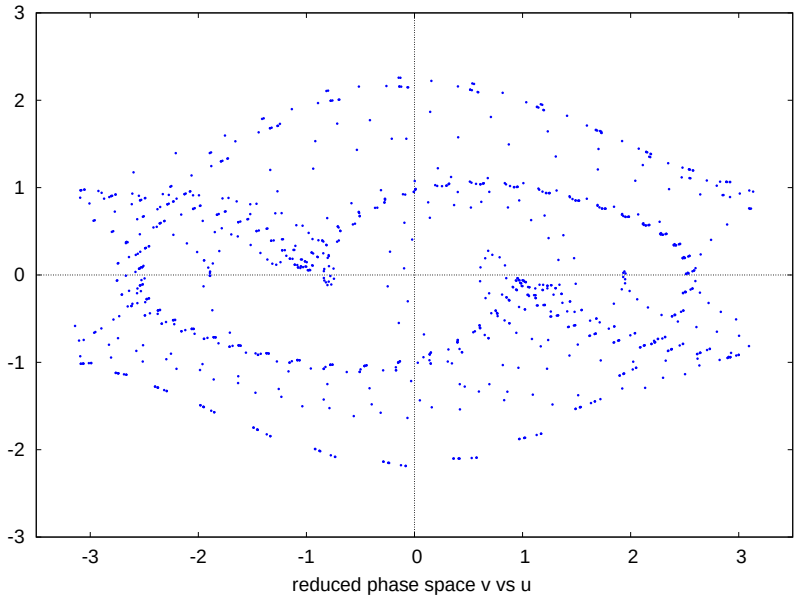

Figure 32: 1000 points reduced phase space plot

and the same set of points drawn with the default **lines** option:

```
(%i28) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
              [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

produces the plot

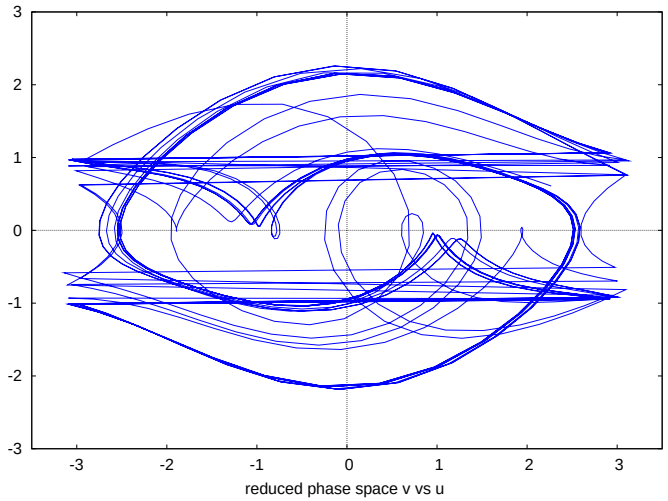

Figure 33: 1000 points reduced phase space plot

3000 point phase space plot

We next draw the same reduced phase space plot, but use the first **3000** points of **uvL_cut**.

```
(%i29) uvL_first : rest (uvL_cut, -4041)$
(%i30) %, fll;
(%o30)      [[0.25464, 1.0166641], [- 2.2822197, - 0.532184], 3000]
(%i31) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
                [style,[points,1,1,1]],
             [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

which produces

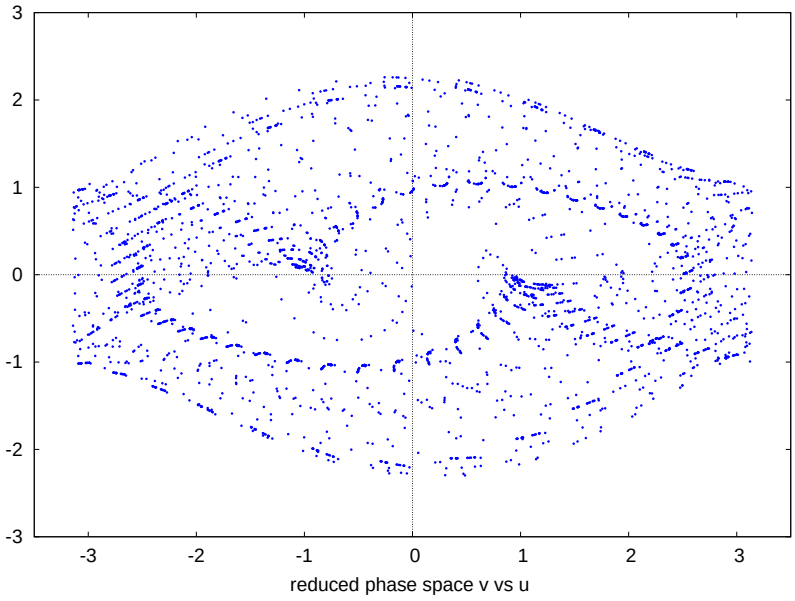

Figure 34: 3000 points reduced phase space plot

and again, the same set of points drawn with the default **lines** option

```
(%i32) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
              [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

which produces the plot

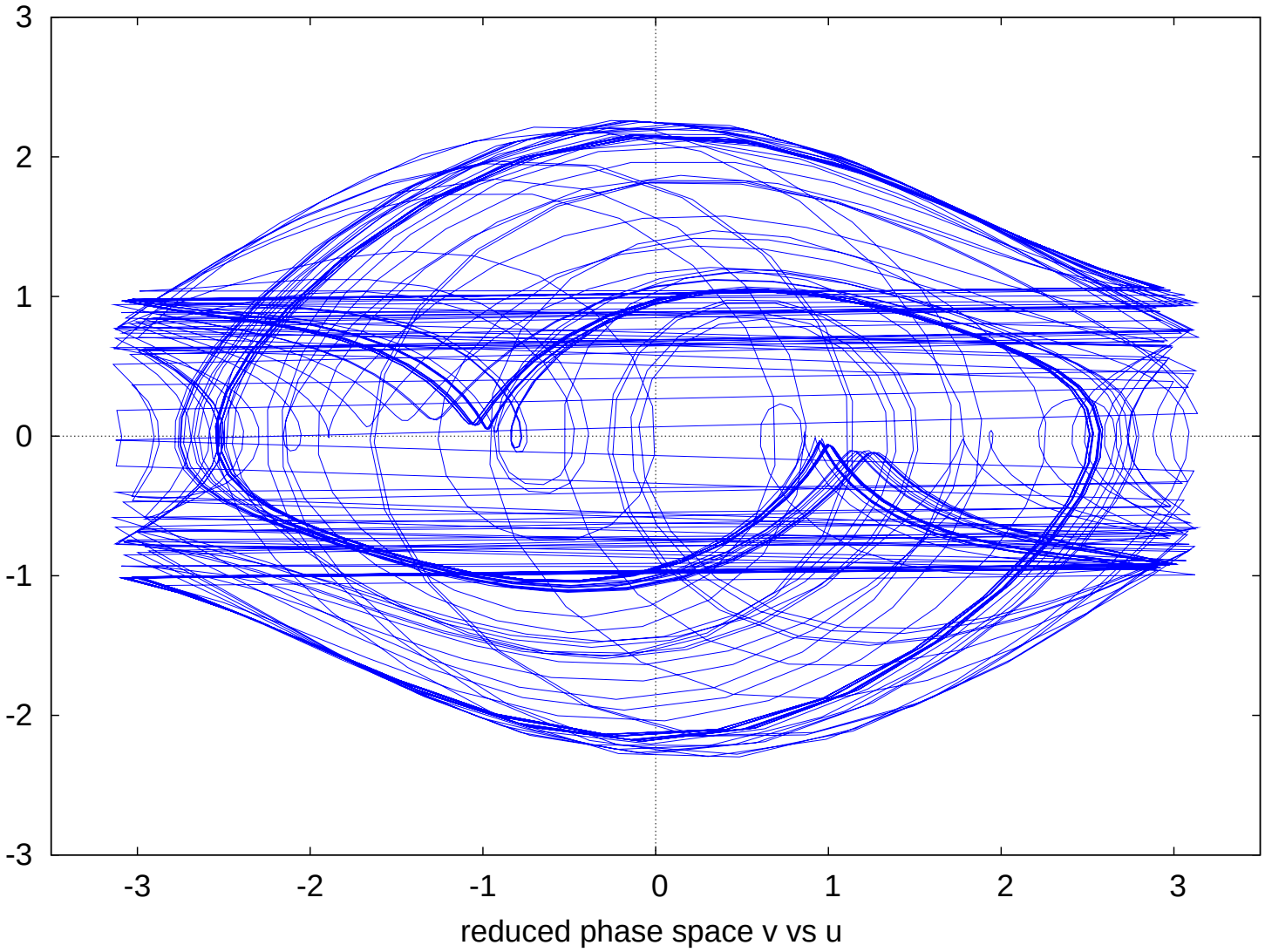

Figure 35: 3000 points reduced phase space plot

Poincare Plot

We now construct the Poincare section plot as before, using all the points available in **uvL_red**.

```
(%i33) pL : makelist(1+10*nsteps + j*nsteps, j, 0, ncycles - 10)$
(%i34) %, fll;
(%o34)                          [311, 7441, 231]
(%i35) poincareL : makelist(uvL_red[i], i, pL)$
(%i36) %, fll;
(%o36)      [[- 2.2070801, 1.3794391], [1.2417605, 0.937008], 231]
(%i37) plot2d ( [discrete,poincareL],[x,-3,3],[y,-4,4],
                  [style,[points,1,1,1]],
             [ylabel," "],[xlabel," Poincare Section v vs u "] )$
```

which produces the plot

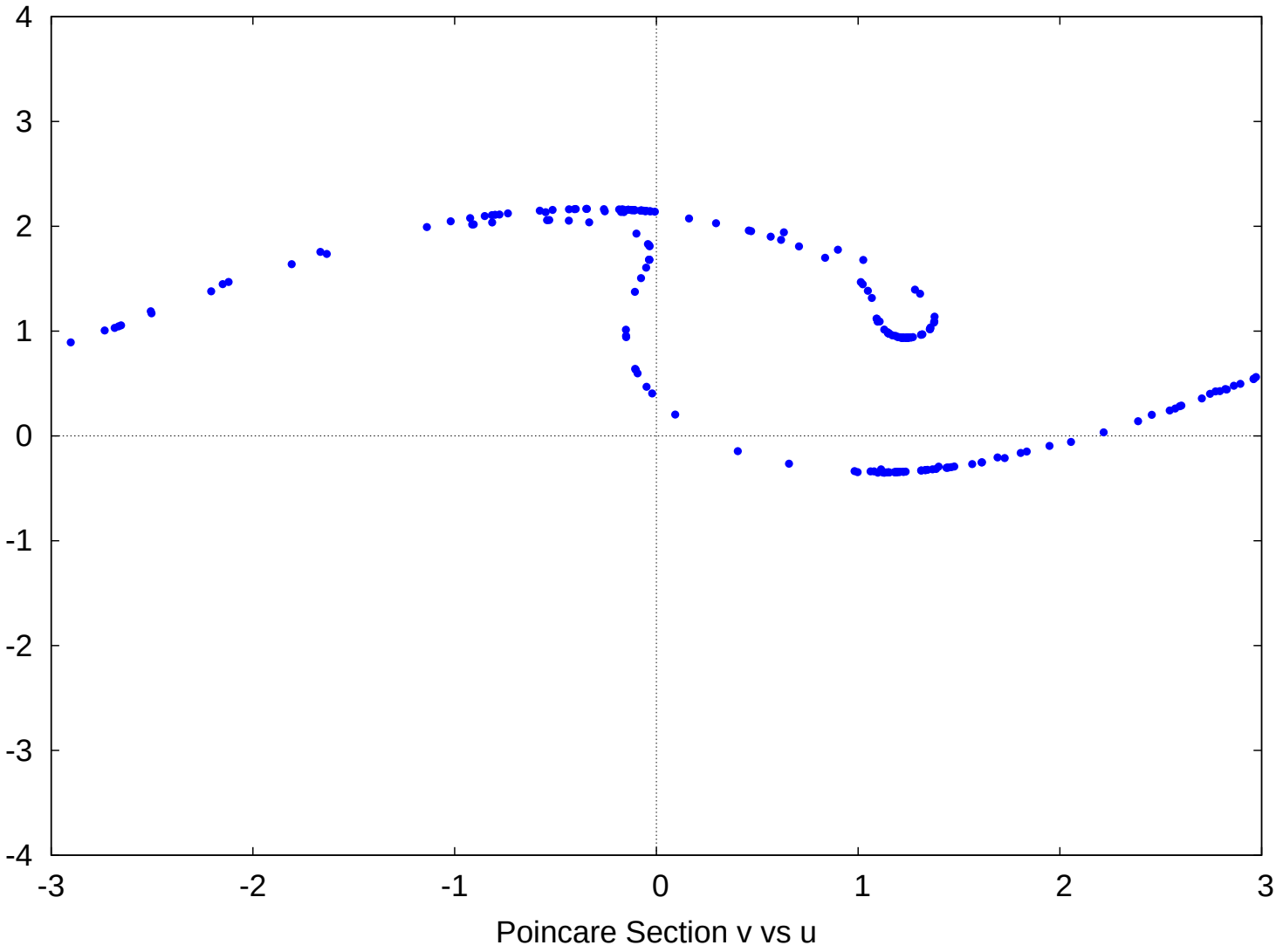

Figure 36: 231 poincare section points

3.5 Using contrib_ode for ODE's

The syntax of **contrib_ode** is the same as **ode**. Let's first solve the same first order ODE example used in the first sections.

```
(%i1) de : 'diff(u,t)- u - exp(-t);
                                 du          - t
(%o1)                            -- - u - %e
                                 dt
(%i2) gsoln : ode2(de,u,t);
                                        - 2 t
                                      %e          t
(%o2)                       u = (%c - -------) %e
                                        2
(%i3) contrib_ode(de,u,t);
                                    du          - t
(%o3)                contrib_ode(-- - u - %e   , u, t)
                                    dt
(%i4) load('contrib_ode);
(%o4) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/contrib/diffequations/c\
ontrib_ode.mac
(%i5) contrib_ode(de,u,t);
                                        - 2 t
                                      %e          t
(%o5)                      [u = (%c - -------) %e ]
                                        2
(%i6) ode_check(de, %[1] );
(%o6)                                   0
```

We see that **contrib_ode**, with the same syntax as **ode**, returns a list (here with one solution, but in general more than one solution) rather than simply one answer.

Moreover, the package includes the Maxima function **ode_check** which can be used to confirm the general solution.

Here is a comparison for our second order ODE example.

```
(%i7) de : 'diff(u,t,2) - 4*u;
                                    2
                                   d u
(%o7)                              --- - 4 u
                                     2
                                   dt
(%i8) gsoln : ode2(de,u,t);
                                    2 t          - 2 t
(%o8)                     u = %k1 %e    + %k2 %e
(%i9) contrib_ode(de,u,t);
                                    2 t          - 2 t
(%o9)                    [u = %k1 %e    + %k2 %e     ]
(%i10) ode_check(de, %[1] );
(%o10)                                  0
```

Here is an example of an ODE which **ode2** cannot solve, but **contrib_ode** can solve.

```
(%i11) de : 'diff(u,t,2) + 'diff(u,t) + t*u;
                                  2
                                 d u   du
(%o11)                           --- + -- + t u
                                   2   dt
                                 dt
(%i12) ode2(de,u,t);
(%o12)                               false
```

```
(%i13) gsoln : contrib_ode(de,u,t);
                                3/2
                     1   (4 t - 1)                            - t/2
(%o13) [u = bessel_y(-, ------------) %k2 sqrt(4 t - 1) %e
                     3        12
                                          3/2
                                 1   (4 t - 1)                          - t/2
                          + bessel_j(-, ------------) %k1 sqrt(4 t - 1) %e     ]
                                 3        12
(%i14) ode_check(de, %[1] );
(%o14)                                  0
```

This section will probably be augmented in the future with more examples of using **contrib_ode**.