APPENDIX B

NUMERICAL CONTINUATION USING DDE-BIFTOOL

In the several figures, results from the delay-differential equation continuation software `DDE-BIFTOOL` are displayed. This appendix serves as a simple walk-through on how the software may be used to recreate the results in the above figure. Please note that the software package comes with an official series of tutorials and examples that are helpful in demonstrating even more functionality that has not been used in my analysis.

B.1 Numerical Continuation

The primary purpose of numerical continuation is to follow the behavior of a specific solution or bifurcation as parameters are varied in the system. Numerical continuation by definition makes use of software, the implicit function theorem, and the existence and uniqueness theorem for solutions to differential equations. A basic example of numerical continuation is in the context of ordinary differential equations. In this scenario, one is given a system

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \lambda), \quad \boldsymbol{x} \in \mathbb{R}^n, \quad \lambda \in \mathbb{R} \tag{B.1}$$

where $f(\mathbf{x}, \lambda)$ is a smooth function and λ is a parameter, that has an equilibrium solution

$$\boldsymbol{f}(\boldsymbol{x_0}, \lambda_0) = 0, \tag{B.2}$$

noting that this solution may vary in value with λ. Following the numerical value of this equilibrium solution is the basic goal of a numerical continuation routine. While doing this, bifurcations of the equilibrium point may be detected by also analyzing the spectrum of the equilibrium; stability information may also be calculated in this process.

The continuation process carried out on an equilibrium point is guaranteed to result in a branch of equilibrium points as long as the Jacobian matrix for the equilibrium point is nondegenerate, i.e. that it has maximal rank. For a one-dimensional differential equation, that means that there exists a branch $\boldsymbol{x}(\lambda)$ for which $\boldsymbol{x}(\lambda_0) = \boldsymbol{x_0}$ and $f(\boldsymbol{x}(\lambda), \lambda) = 0$. To "build up" the branch, a new parameter value close to the previous $\lambda_1 = \lambda_0 + \epsilon$ and the differential equation is evaluated at $\boldsymbol{x_0}$. A root-finding algorithm such as Newton's Method is applied in order to calculate the new position of the equilibrium point. Codimension-1 bifurcations may be detected along the branch by analyzing the Jacobian for singularities. In order to follow an equilibrium point around folds where the derivative is infinite, distance *along the branch* is used as the independent variable for a solution—the details of such a formulation are available from, e.g. Kuznetsov [24] §10.2.

The process may be extended to continuation of periodic orbits. Here, the computation is similar to the case of an equilibrium point, except that fixed points of the Poincaré Map are continued. The Floquet multipliers and period of the periodic orbit are also calculated in this process, so bifurcations in the limit cycle may be detected by inspection of those quantities.

In this work, numerical continuation is used as a tool to confirm perturbation results and explore exciting phenomena. Numerical continuation in delay-

differential equations is explored in detail by Engelborghs [14], who also wrote a package for MATLAB to perform these computations named DDE-BIFTOOL. The process of applying this package to the Rayleigh-Plesset Equation is described in detail in the following sections.

B.2 Installation

The installation files for DDE-BIFTOOL can be retrieved from http://twr.cs.kuleuven.be/research/software/delay/ddebiftool.shtml. The function and runtime files used in this walkthrough are available at http://www.math.cornell.edu/~rand/randdocs/Heckman_DDEBiftool.

Follow the link to the "warranty and download" page, and download the version 2.03 ZIP archive. Upon unzipping the file, a new directory "DDE-BIFTOOL_203" is created. In it, there are three more sets of archives, a readme, and an addendum manual. The set of ZIP archives named TW330.p*.zip contains the full manual for the tool, replete with descriptions of each major function included, data structures used, and a walk-through for the packaged demos in PostScript and PDF format. Reading through this manual is strongly recommended for the beginner in continuation, DDEs, or those relatively unfamiliar with the MATLAB programming language.

The next archive is named ddbiftool.zip. Unpackaging it will create a new directory named ddebiftool, which contains all of the resources that comprise the tool. This folder should be placed somewhere that is already in the MATLAB path, or should be added to the path manually. The way to accomplish this via the MATLAB GUI is dependent on the user's version of MATLAB,

but it may be also accomplished by using the `path` command (for more on the command, read the on-line help: `help path`).

Finally, the archive named `system.zip` contains a directory named `system`, under which there are three more directories full of demonstration systems. "System files" and runtime files (those extracted from `ddebiftool.zip`) should be kept in separate directories, and in particular the working directory should be set to the folder of the system on which the user is operating. Therefore, to run the main demo for `DDE-BIFTOOL`, the working directory should be changed to the `demo` folder under `system`. New systems may be created in arbitrary locations in the user's filesystem as a result.

B.3 System Functions

There are a number of resource files that must first be created in a separate directory before commands may be run in MATLAB to start the computation. Place the following files in their own folder, separate from the `DDE-BIFTOOL` runtime files. By default, the software unpacks a folder named `system` in which a number of tutorials are located; a new folder under this directory is an appropriate location for these new system files.

The first file that must be written is the system definition file, `sys_rhs.m`.

```
function f=sys_rhs(xx,par)

x = xx(1,1);                % state variable
xd = xx(2,1);               % first derivative of state variable
xl = xx(1,2);               % state variable, time lagged
xld = xx(2,2);              % derivative of time-lag state var.
```

```
e = 1;                          % arbitrary value of epsilon
k = 6.8915;                     % slightly-detuned from Hopf-Hopf

f(1,1) = xd;                    % the system of ODEs
f(2,1) = -(4/k)*xd - 4*x - (10/k)*xld + (e/k)*x.^3;

return;
```

sys_rhs.m

Another file needed is `sys_tau.m`. This file designates "which parameter" is the delay parameter in the system. Note that `DDE-BIFTOOL` can support multiple delay parameters. However, the software *does not have a central listing* of these parameters for each system; they are provided in an anonymous fashion within `sys_rhs` and are designated by handles e.g. `par(i)`, where `i` is consistent across all references to a particular parameter. In this system, there is only *one* parameter, and that parameter *is* the delay parameter. Therefore, the file `sys_tau.m` could not be any simpler:

```
function tau=sys_tau()

% T

tau=[1];

return;
```

sys_tau.m

The next file that must be established calculates the Jacobian (partial derivatives with respect to state variables and parameters). This has the file-

name `sys_deri.m`. In many cases, it suffices to use a default file provided with `DDE-BIFTOOL` that calculates these partial derivatives numerically (`df_deriv.m`, found in the package's root directory). I will make use of that method in my computations, rather than writing a custom Jacobian resource file.

For the continuation to work, the files `sys_ntau.m` and `sys_init.m` must also be copied. The latter file should be modified to have a correct directory traversal with respect to the current directory, as well as an appropriate "name" for the system. For this system, these files are:

```
1 function []=sys_ntau()

3 error('SYS_NTAU: This sys_ntau is a dummy file!');

5 return;
```

sys_ntau.m

```
function [name,dim]=sys_init()
2
name='dummy';
4 dim=2;
path(path,'../../ddebiftool/');
6
return;
```

sys_init.m

B.4 Runtime scripts

The simplest way to run the software package is by scripting the commands in an m-file. Below is a list of the commands used to generate the figure in this paper, with comments interspersed to explain what the commands are doing.

This clears the workspace and initializes the system handle:

```
clear all
[name,n]=sys_init;
```

Next, this identifies the bifurcation parameter continuation domain, and the initial step size to use for the bifurcation parameter. Note that since this system only uses the time delay as the bifurcation parameter, the variables begin with "delay;" this is not required.

```
delay_begin = 2;
delay_end = 4;
delay_step = 0.0001;
```

Beyond the creation of double-precision integer arrays, MATLAB also facilitates the organized storage of data by use of the `struct` class. Note that in the call to `stst.kind='stst'`, the *first* instance of `stst` is the *name* of the variable, and the *second* is the *kind* of object. From a MATLAB data structure perspective, the structure `stst` is being created, with one of its fields (`kind`) being set to the string `stst` as well.

```
stst.kind='stst';
```

```
stst.parameter=delay_begin; % first parameter value
stst.x=[0 0]';  % approx ic's for eq. pt.
method=df_mthod('stst');
[stst,success]=p_correc(stst,[],[],method.point);
```

Next, the field `paramter` is set for the variable `stst`, and it is specified as the parameter value previously assigned to `delay_begin`. Finally, the command `stst.x=[0 0]'` assigns an *approximate* location of an equilibrium point to the variable `stst`. It turns out in this case that this is the exact location of the equilibrium point, but this is only verified after `p_correc`, a function that is part of the tool. The command preceding sets the correction method to look for an equilibrium point.

```
method.stability.minimal_real_part=-50;
```

There are an infinite number of complex roots to a differential delay equation's characteristic equation. However, an infinite number will have negative real part, and only a finite number will have positive real part. Therefore, since we are mostly concerned about the roots as they cross the imaginary axis for stability purposes, we may sensibly ignore those roots that are very far away from it. This setting specifies the minimum real part needed (i.e. the largest negative real part) for `DDE-BIFTOOL` to calculate the root.

This command will calculate the stability of the equilibrium point `stst` and set `stability` as a new field of `stst` to the calculation output.

```
stst.stability=p_stabil(stst,method.stability);
```

The next command plots a locus of roots for the equilibrium point `ststst` in the complex plane. Note that first a predictive step for the roots is taken, followed by a corrective step. Red roots designate positive real part which lead to asymptotic instability. This is demonstrated in Figure B.1.

```
figure(1); clf; p_splot(ststst)
```

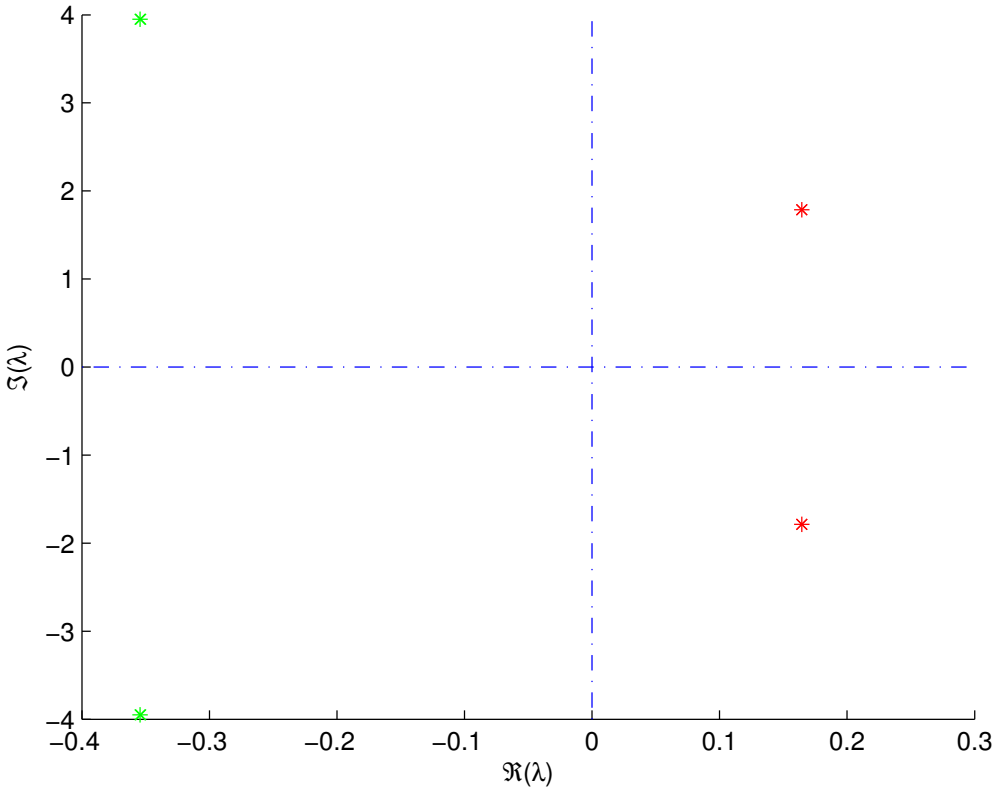

Figure B.1: Plot of eigenvalues of the origin in the complex plane as produced by p_splot during runtime.

Central to continuation is the concept of a "branch." This is a collection of solutions wherein the continuation parameter is varied slightly and the perturbed solution is calculated. This creates a sequence of objects (equilibrium points, limit cycles, etc.) that are topologically equivalent. Should branches have a definite "beginning" or "ending," they are located at bifurcation points.

The below sequence of commands is used to build up the branch of equilibrium points starting at `stst` via continuation. First, a branch object is created and named `branch1`. It is designated to have as the continuation parameter the "first parameter" in the list (in this system, there is only one parameter—the delay), and that it will be a branch of equilibrium points. The `max_bound` and `max_step` fields set the maximum bound and initial step size for the continuation parameter, respectively. Note the first entry in the input vectors are `1`, the "parameter position;" for continuation of the same system in more than one variable, other branches will have a different value depending on the parameter of interest.

```
branch1 = df_brnch(1,'stst'); % first (and only) parameter (delay)
branch1.parameter.max_bound = [1 delay_end]; % 1 is the parameter
    pos.
branch1.parameter.max_step  = [1 delay_step]; % same as above
branch1.point(1) = stst; % start with the steady state point
    determined
stst.parameter = delay_begin+delay_step;
[stst,success]=p_correc(stst,[],[],method.point);
branch1.point(2)=stst; % next branch point is as calculated
```

Next, the branch is given a starting point—in this case, the original trivial

equilibrium point designated earlier in `stst`. The data from the variable `stst` is copied into `branch1.point(1)`. With that information integrated into the branch, the next command increases the delay parameter slightly and the equilibrium, followed by recalculating the point position in case it has changed due to the new parameter value (in this case it will not move, since this is a trivial equilibrium point). This new equilibrium point is also copied into the branch as `branch1.point(2)`.

```
branch1.method.continuation.plot=0;
[branch1,s,f,r]=br_contn(branch1,50000);
```

The first command above turns off plotting when running continuation on this branch; this is sensible here because the branch is trivial, and the plot output would identify a solution with zero amplitude for the range of continuation.

The second command runs the continuation routine on the branch, for as many as 50,000 iterations or until the maximum parameter bound defined in `branch1.parameter.max_bound` is reached. This will populate the structure `branch1` with equilibrium points whose location is recalculated at each new parameter value, sufficing for trivial and nontrivial equilibrium points.

Many of the functions that act on `point` data types also have analogous counterparts for `branch` data types; for instance, whereas `p_stabil` calculates the stability of a single point, `br_stabl` calculates the stability of each point that is within a branch. This will assign new stability information to the branch point-by-point, and is used to identify bifurcation points. Note that the prepackaged version of the function contains a safety check that has been disabled for this calculation.

```
branch1 = br_stabl(branch1,0,1);
```

Equipped with the stability of the equilibrium point along the branch, we know that changes in stability will correspond to bifurcations. The function to locate Hopf points is `p_tohopf`; it takes an "initial guess" of an equilibrium point that is undergoing Hopf bifurcation as input, and as output returns a machine-precision approximation for the location of a Hopf point. Here, it is output first as the variable `hopf`, and then renamed to `first_hopf` to distinguish itself from later Hopf bifurcations. Note that if there are multiple Hopf points close to one another, the initial approximation point will have to be precisely chosen.

```
hopf=p_tohopf(branch1.point(10));
method=df_mthod('hopf');
[hopf,success]=p_correc(hopf,1,[],method.point);
first_hopf=hopf;
```

With a Hopf point identified, a natural next step would be to characterize the periodic solution that bifurcates from the equilibrium point. In particular, what is the amplitude-parameter dependence of the limit cycle? Whereas this is a computationally expensive and tedious exercise using numerical integration, continuation is uniquely equipped to quickly calculate limit cycle profiles via calculating fixed points of Poincaré maps.

Two separate points along the periodic solution are required to build up a branch of periodic solutions. In the below code, those two points are `psol` and

`deg_psol`. The latter corresponds to the periodic solution *exactly corresponding* to the Hopf point `first_hopf`, and the former to a slightly detuned point from the Hopf. Lines 1–3 above establish a point along the periodic solution branch `psol` by making a guess at the periodic solution using a polynomial of degree 3. The function `p_corrc` is then employed to correct the location of the periodic solution. Separately, a new (empty) branch for the periodic solution `branch2` is initialized and parameter bounds are set. Finally, `deg_psol` is calculated, coincident with the Hopf point. The data from `deg_psol` and `psol` are copied to `branch2`, and the function `br_contn` runs the continuation calculation.

The result is a branch full of points corresponding to the periodic solution that had bifurcated from `first_hopf`; this is shown in Figure B.2.

```
intervals=20;
degree=3;
[psol,stepcond]=p_topsol(first_hopf,1e-4,degree,intervals);
method=df_mthod('psol');
[psol,success]=p_correc(psol,1,stepcond,method.point); % correction
branch2=df_brnch(1,'psol');
branch2.parameter.max_bound=[1 delay_end];
branch2.parameter.max_step=[1 .01]; % custom
deg_psol=p_topsol(first_hopf,0,degree,intervals);
deg_psol.mesh=[]; % save memory by clearing the mesh field
branch2.point=deg_psol;
psol.mesh=[];
branch2.point(2)=psol;
figure(23);
[branch2,s,f,r]=br_contn(branch2,100);
```

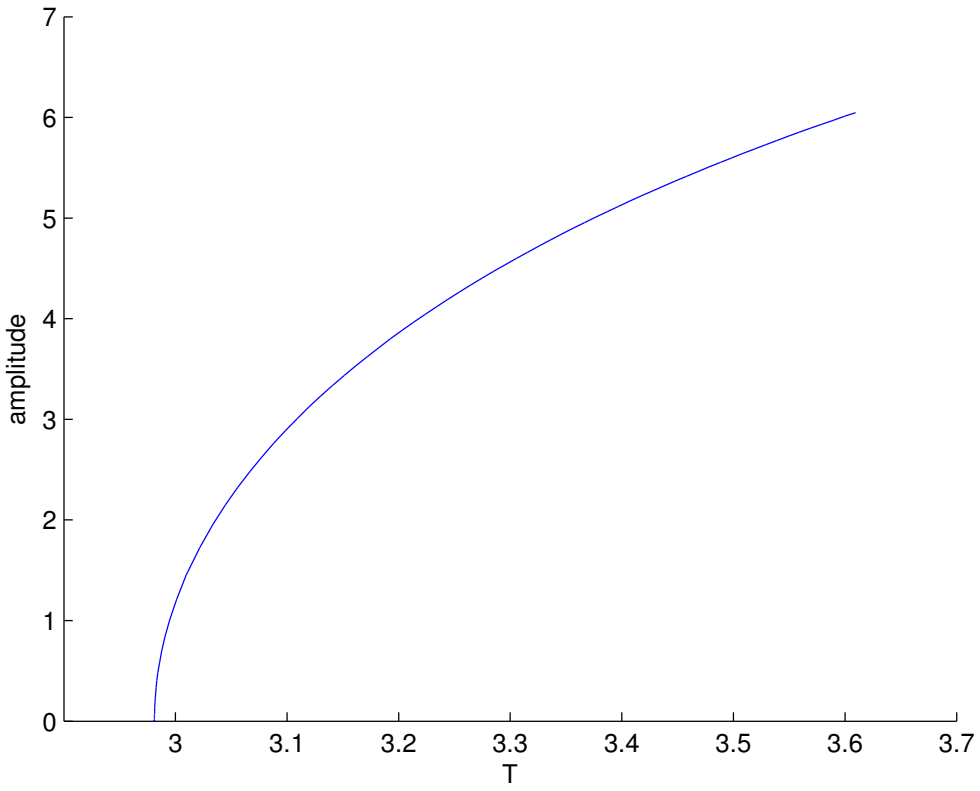

Figure B.2: Continuation output of the first nontrivial branch as generated by br_contn.

This particular system is almost degenerate—the first and second Hopf bifurcations are tuned to occur at parameter values that are very close to one another. It turns out too that the numerical algorithm that locates the Hopf bifurcations (`p_tohopf`) often settles on one Hopf bifurcation far more often than the other. As a result, finding *both* Hopf points can be difficult. The script `find_hopf.m` is a suggestion of how to find a Hopf point unique from the first one by comparing the frequencies, and is provided below.

```
pt = 1;
w0 = first_hopf.omega;
second_hopf.omega = w0;
```

```
while abs(second_hopf.omega - w0) < 0.1 && pt < length(branch1.point)
    pt = pt + 1;
    hopf = p_tohopf(branch1.point(pt));
    method=df_mthod('hopf');
    [hopf,success]=p_correc(hopf,1,[],method.point);
    second_hopf = hopf;
    hopf.stability=p_stabil(hopf,method.stability);
end
```

find_hopf.m

With the second Hopf point found at index `pt`, the branch is then built up as before; first, the new Hopf point is identified and corrected.

```
hopf = p_tohopf(branch1.point(pt));
method=df_mthod('hopf');
[hopf,success]=p_correc(hopf,1,[],method.point);
second_hopf = hopf;
hopf.stability=p_stabil(hopf,method.stability);
```

The rest of the continuation follows exact as that done on `branch2` above, except all bifurcating from `second_hopf` rather than `first_hopf`. Below is the script that accomplishes this, and the output is displayed in Figure B.3.

```
intervals=20;
degree=3;
[psol,stepcond]=p_topsol(second_hopf,1e-4,degree,intervals);
method=df_mthod('psol');
[psol,success]=p_correc(psol,1,stepcond,method.point); % correction
branch3=df_brnch(1,'psol');
```

```
branch3.parameter.max_bound=[1 delay_end];
branch3.parameter.max_step=[1 .01];
deg_psol=p_topsol(second_hopf,0,degree,intervals);
deg_psol.mesh=[]; % save memory by clearing the mesh field
branch3.point=deg_psol;
psol.mesh=[];
branch3.point(2)=psol;
figure(23);
[branch3,s,f,r]=br_contn(branch3,100);
```

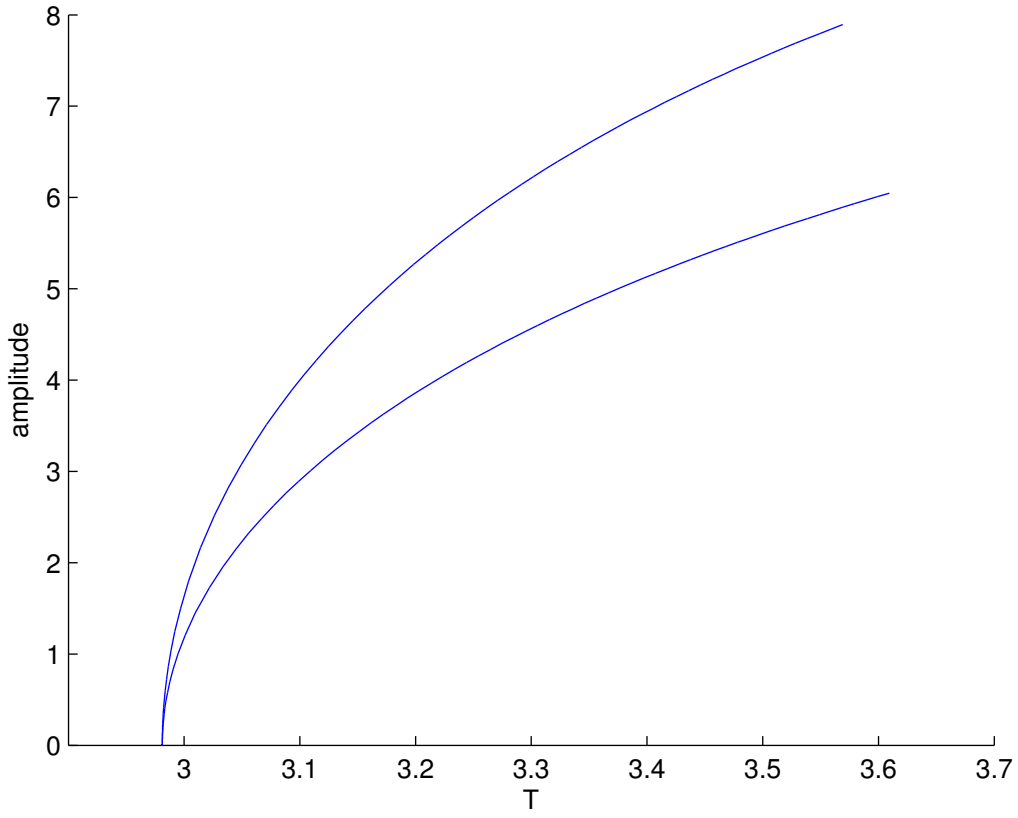

Figure B.3: Continuation output of the second nontrivial branch as generated by br_contn. Note that this branch bifurcates from the same Hopf point but generates a different amplitude prediction, due to the Hopf point's degeneracy.