

Assignment 4

2.086/2.090 Spring 2013

Released: *Friday, 5 April, at 5 PM.*

Due: *Friday, 19 April, at 5 PM.*

Upload your solution as a zip file “YOURNAME_ASSIGNMENT_4” which includes for each question the `AxQy` script *as well as* any MATLAB functions (of your own creation) which are called by your script. Note the MATLAB functions of your own creation may include both specific MATLAB functions requested in the question statement but also any other MATLAB functions (called directly or indirectly by your script or by other functions) which you choose to develop as part of your answer. Both the scripts and (requested) functions must conform to the formats described in **Instructions** and **Questions** below. You should also include in your folder all the `grade_o_matic` .p files for Assignment 4.

Instructions

Before embarking on this assignment you should

- (1) Complete the Textbook reading for Unit IV and review the Lecture Notes for Unit IV
- (2) Execute (“cell-by-cell”) the two MATLAB Tutorials for Unit IV Passing Functions (Handles), Anonymous Functions, and MATLAB `ode45`; MATLAB `eig`. (Note Chapter 6, in particular Sections 6.5 and 6.6, also addresses relevant MATLAB issues.)
- (3) Download the `Assignment_4_Materials` folder. This folder contains a template for the script associated with each question (`A4Qy_Template` for Question `y`), as well as a template for each function which we ask you to create (`func_Template` for a function `func`). The `Assignment_4_Materials` folder also contains the `grade_o_matic` codes needed for Assignment 4. (Please see Assignment 1 for a description of `grade_o_matic`.)

We indicate here several general format and performance requirements:

- (a.) Your script for Question `y` of Assignment `x` *must* be a proper MATLAB “.m” script file and *must* be named `AxQy.m`. In some cases the script will be trivial and you may submit the template “as is” — just remove the `_Template` — in your “YOURNAME_ASSIGNMENT_4” folder. But note that you still must submit a proper `AxQy.m` script or `grade_o_matic_A4` will not perform correctly.
- (b.) In this assignment, for each question `y`, we will specify inputs and outputs both for the script `A4Qy` and (as is more traditional) any requested MATLAB functions; we shall denote the former as script inputs and script outputs and the latter as function inputs and function outputs. For each question and hence each script, and also each function, we will identify *allowable instances* for the inputs — the parameter values or “parameter domains” for which the codes must work.
- (c.) Recall that for scripts, input variables must be assigned *outside* your script (of course before the script is executed) — *not* inside your script — in the workspace; all other

variables required by the script must be defined *inside* the script. Hence you should test your scripts in the following fashion: `clear` the workspace; assign the input variables in the workspace; run your script. Note to test your MATLAB functions you need not take such precautions: all inputs and outputs are passed through the input and output argument lists; a function enjoys a private workspace.

- (d.) We ask that you not end any of your script names or function names with the suffix `_ref` in order to prevent conflicts with scripts or functions provided in the folder `Assignment_4_Materials` (and needed by `grade_o_matic_A4`).
- (e.) We ask that in the submitted version of your scripts and functions you suppress all display by placing a “;” at the end of each line of code. (Of course during debugging you will often choose to display many intermediate and final results.) We also require that **before** you upload your solution you should run `grade_o_matic_A4` (from your `YOURNAME_ASSIGNMENT_4` folder) for final confirmation that all is in order.

Testing: Manufactured Solutions

In the course thus far it was fairly easy to develop test cases for your codes. In particular, it was often simple to identify instances for which the solution is known; you could then test your implementation relative to this solution. In Unit IV and beyond we will increasingly encounter problems for which it is difficult to develop instances for which the (exact) solution is known.

For example, consider a linear ODE IVP of the form

$$\frac{dw}{dt} = Aw + F(t), \quad 0 \leq t \leq t_f, \quad (1)$$

subject to initial conditions $w(0) = w_0$. Here w may be a scalar or more generally an $n \times 1$ state vector; A is an $n \times n$ (time-independent) matrix and F is an $n \times 1$ (time-dependent) vector. (We can also directly extend the “manufactured solutions” framework to nonlinear problems — and indeed it is for nonlinear problems that the framework is most useful.) We might develop (say) a “home-grown” Euler Backward scheme for approximation of this equation (as in Question 1 below), or perhaps we might apply the MATLAB `ode45` function based on a RK4 approximation of this equation (as in Question 2 below). In either case we would like to test the implementation. We should certainly look for w_0 or F for which we can develop a simple expression for the exact solution $w(t)$. But in general this might not be possible or perhaps these exact solutions will be too simple to test all aspects of the code.

It is thus often useful to also pursue the method of “Manufactured Solutions”: you start with an *assumed solution* — some prescribed relatively simple expression $w(t)$; you then insert this assumed answer $w(t)$ into (1) to find the *question* — the w_0 and in particular the $F(t)$ — which yields the chosen $w(t)$; finally, you now test your code for this w_0 and $F(t)$ to ascertain that the Euler Backward approximation approaches $w(t)$ — which, by construction, you know! — at the correct rate as (say) J , the number of timesteps, increases (for fixed t_f). Note it is not sufficient to obtain an answer which is “close”: it is important to test convergence, and even convergence rate, in order to flush out certain kinds of errors. We emphasize that we do not modify A as in some sense it is the implementation of A that we wish to confirm.

Finally, it is important to note that the method of Manufactured Solutions only confirms (at best) that the code implements the desired numerical scheme for the given “dynamics” A . In par-

ticular, it does not tell you for some particular “non-manufactured” w_0 and F of interest whether for a given J your numerical solution $\tilde{w}(t)$ is sufficiently close to the exact solution $w(t)$. For that purpose you must use your insight, as well as quantitative measures informed by *a priori* and *a posteriori* error estimates.

Questions

1. (25 points) We consider here the scalar ODE IVP

$$\begin{cases} \frac{du}{dt} = \lambda u + f(t), & 0 < t \leq t_f \\ u(t=0) = u_0 \end{cases}, \quad (2)$$

for $\lambda \leq 0$. We recall that this equation is the lumped model for the temperature evolution of a body: u is the temperature (measured relative to ambient temperature), in $^{\circ}C$; u_0 is the initial temperature (measured relative to ambient temperature), in $^{\circ}C$; λ is the (negative of the) heat transfer coefficient times the area of the body divided by the specific heat times the mass of the body, in units of s^{-1} ($-\lambda$ is hence an inverse time constant of our first-order system); and $f(t)$ is the heat generation (in Watts) divided by the specific heat times the mass of the body, in units of $^{\circ}Cs^{-1}$. We note that λ is *negative*.

The Euler Backward discretization of (2) will yield an approximate solution $\tilde{u}^j = \tilde{u}(j \Delta t) (\approx u(j \Delta t))$, $0 \leq j \leq J$, for $\Delta t = t_f/J$. In this question we would like you to implement the Euler Backward method in a MATLAB function with signature

```
function [u_vec] = Euler_Backward(u_0,lambda,f_source,t_final,J)
```

in order to obtain the approximate temperature history of the body, $\mathbf{u_vec}(j) = \tilde{u}((j-1) \Delta t)$, $1 \leq j \leq J+1$, for prescribed initial condition $\mathbf{u_0}$, parameter \mathbf{lambda} , “source” function $\mathbf{f_source}$, and final time $\mathbf{t_final}$ ($= t_f$). Recall that the function “signature” refers to the function name and the input and output lists.

The function must be named `Euler_Backward` and furthermore must be stored in a file named `Euler_Backward.m`. The function takes five function inputs. The first input is the scalar $\mathbf{u_0}$ which corresponds to the initial condition u_0 in (2); the set of allowable instances, or parameter domain, is not restricted (any finite value is admissible). The second input is the scalar \mathbf{lambda} which corresponds to λ in (2); the set of allowable instances, or parameter domain, is the *negative* real numbers. The third input is the MATLAB function $\mathbf{f_source}$ which “implements” in MATLAB the source function $f(t)$ in (2); the function $\mathbf{f_source}$ should have signature template `function [f_val] = f_source(time)` where the input to function $\mathbf{f_source}$ is the scalar \mathbf{time} and the output from function $\mathbf{f_source}$ is the scalar $\mathbf{f_val}$ such that $\mathbf{f_val} = f(\mathbf{time})$.¹ The fourth input is the scalar $\mathbf{t_final}$ which corresponds to the final time t_f in (2); the set of allowable instances, or parameter domain, is the positive real numbers (since our initial time here is, for simplicity, fixed as zero). The fifth input is the

¹Note we prefer to include $\mathbf{f_source}$ as an argument to `Euler_Backward` rather than “hardwire” the source function inside `Euler_Backward` so that we do not need to re-write, re-debug, and re-test a new Euler Backward code for each new desired source function $f(t)$ — re-use is one of the great strengths, along with modularity and locality, of the “function” concept in programming languages.

scalar J which corresponds to J as defined in the Euler Backward discretization; the set of allowable instances, or parameter domain, is the positive integers. (Note that Δt is not an input but rather should be calculated (within your function `Euler_Backward`) as $\Delta t \equiv t_f/J$.) The function yields a single function output: the output is the $J+1 \times 1$ vector `u_vec` which corresponds to $\tilde{u}(j \Delta t)$, $0 \leq j \leq J$.²

The script for this question is provided in `A4Q1_Template.m`; you should just remove the `_Template` but you should *not* modify the body of the script in any way or `grade_o_matic_A4` will be very unhappy and you will receive no credit, and furthermore for this particular question you should not even *run* the script `A4Q1` — rather, you should directly test your `Euler_Backward` code as described in “Guidelines and Hints” below. We also provide a template for the Euler Backward code, `Euler_Backward_Template`. Note the only deliverables for this problem are your script `A4Q1` and your `Euler_Backward` function. In particular, any “`f_source`” functions (see Guidelines and Hints below) which you create to test your `Euler_Backward` code are for your own purposes and should not be uploaded in `YOURNAME_ASSIGNMENT_4`; `grade_o_matic` will create its own instances (unknown to you) of “`f_source`” with which to test your `Euler_Backward` code.

Guidelines and Hints. We recall that the particular names chosen for the inputs and outputs in the function signature/body of a function is not important: it is only the number and order of the inputs and outputs which matters to ensure correct instantiation of the function inputs, and correct assignment of the function outputs, when the function is called by another program.³ (In contrast, in a script, which does not have a private workspace, the specific names of the variables are important.) In particular, we know that the names of the variables in the argument list of the function call need not be in any way related to the names of the “dummy” variables inside the function definition.

This concept also applies to inputs which are function (handles): there is thus no reason why the name of the function which implements $f(t)$ and which will appear in the argument list of the *call* to `Euler_Backward` should be the same as the name of the second “dummy” argument in the `Euler_Backward` definition — and indeed, this would be very cumbersome. Instead, you may create several different MATLAB functions, say `f_1` and `f_2`, which implement different sources $f(t)$, say $f(t) \equiv f_1(t) \equiv \sin(t)$ and $f(t) \equiv f_2(t) \equiv t$, respectively, and then call `Euler_Backward(u_0,lambda,@f_1,t_final,J)` and `Euler_Backward(u_0,lambda,@f_2,t_final,J)`, respectively, to obtain the corresponding numerical solutions. We thus recognize that the signature we provided earlier for `f_source` is more properly viewed as a signature *template* which indicates the input and output lists and desired functionality: you are free to choose any names for the inputs and outputs and indeed the function name.⁴ Note you may also find it very convenient for simple source terms to call the Euler Backward code with anonymous or “in-line” functions, for example `Euler_Backward(u_0,lambda,@(t)t,t_final,J)`; this anonymous function shares the signature template of `f_source` except that no output name is required as there is perforce

²Note it may be helpful for debugging purposes to plot your solution; this may be effected as `plot(linspace(0,t_final,J+1)',[u_vec])`.

³In fact, more advanced MATLAB argument handling procedures even permit flexibility in the number and order of inputs and outputs: the call to the function selectively instantiates certain inputs (with others set to defaults) and selectively chooses certain outputs.

⁴Often the function name may be prescribed by the “client,” which for you is effectively `grade_o_matic`; but for this particular question `grade_o_matic` will call `Euler_Backward` with its own source functions and associated MATLAB implementations.

a single output (and indeed even a function name is optional if the anonymous function is defined directly in the call to `Euler_Backward`).

Challenge 1. This challenge relates to Question 1. Develop an *a posteriori* estimate for the error $|u(t_f) - \tilde{u}(t_f)|$ — which of course should *not* require knowledge of the exact solution. Keywords (from the textbook, Unit I and Unit IV, and the Unit IV Lecture Notes): truncation error and discretization error; Euler Backward; *a priori* error bound; finite difference approximation to second derivative. (Note in practice these concepts are often taken further: the development of adaptive schemes which decrease or increase the timestep as the integration proceeds in order to achieve, but not over-achieve, a given error tolerance.)

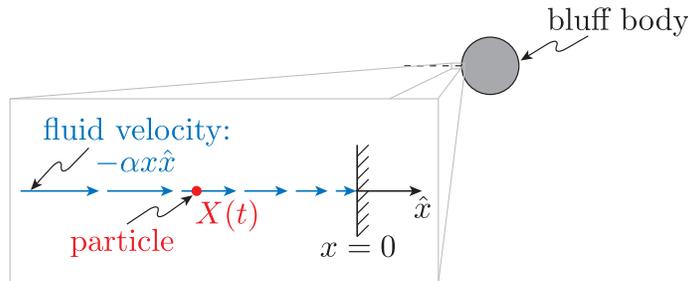


Figure 1: Particle in a stagnation flowfield.

2. (25 points) A small spherical particle immersed in fluid stream approaches (from the left) a wall at $x = 0$. As a first approximation, the governing equation for the position (in x) of the particle as a function of time, $X(t)$, is

$$\begin{cases} \frac{d^2 X}{dt^2} + b \frac{dX}{dt} + b\alpha X = 0, & 0 < t \leq t_f \\ X(0) = -L, \quad \frac{dX}{dt}(0) = V \end{cases}, \quad (3)$$

where b is related to the Stokes (low-Reynolds number) drag force on the particle (divided by the mass of particle) and has units s^{-1} , α is related to the gradient of the flow in the vicinity of the wall (divided by the mass of the particle) and also has units s^{-1} , and finally $-L$ (in m) and V (in m/s) are the initial position and initial velocity of the particle. (We neglect gravitational effects.)

We note that (3) assumes the fluid flow takes the form $-\alpha x$ for x a coordinate normal to and decreasing away from the wall: this represents a stagnation flow along a line of symmetry (for example, if $x = 0$ corresponds to the stagnation point at the “nose” of a bluff body); we depict the situation in Figure 1. The solution is only relevant for times t such that $X(t) \leq 0$ — before (if) the particle reaches the wall — however you may continue the calculation beyond contact and then inspect the solution to detect the time at which the particle (first) collides with the wall.⁵ This simple model (typically extended to two or three spatial dimensions) is relevant in many industrial processes such as particle separation.

⁵Note in fact our problem has another interpretation: we again consider a stagnation flow but now view x as the coordinate *along* the wall such that the stagnation point $x = 0$ is a particle “trap”; in this case the particle will oscillate about and gradually approach $x = 0$. In this case the solution is relevant both for x negative and x positive.

We denote our state variable as $w \equiv (w_1 \ w_2)^T$ for $w_1 \equiv X$ and $w_2 \equiv \frac{dX}{dt}$. It is then possible to express (3) as

$$\begin{cases} \frac{dw_1}{dt} = g_1(t, w, b, \alpha) \\ \frac{dw_2}{dt} = g_2(t, w, b, \alpha) \end{cases}, \quad 0 \leq t \leq t_f, \quad (4)$$

or even more succinctly as

$$\frac{dw}{dt} = g(t, w, b, \alpha), \quad 0 \leq t \leq t_f, \quad (5)$$

where $g(t, w, b, \alpha) \equiv (g_1(t, w, b, \alpha) \ g_2(t, w, b, \alpha))^T$. You will need to derive the form of $g(t, w, b, \alpha)$ from (3). We provide w with the initial conditions prescribed in the problem statement,

$$w(t=0) \equiv w_0 = (-L \ V)^T. \quad (6)$$

We would like you to write a script which solves (approximately) (5) with the MATLAB function `ode45`.

The script takes six script inputs. The first script input is the scalar damping coefficient b which must correspond in your script to MATLAB variable `b`; the set of allowable instances, or parameter domain, is $0.01 \leq b \leq 100$. The second script input is the scalar velocity parameter α which must correspond in your script to MATLAB variable `alpha`; the set of allowable instances, or parameter domain, is $0 \leq \alpha \leq 1$. The third input is the scalar initial velocity of the particle V which must correspond in your script to MATLAB variable `V`; the set of allowable instances, or parameter domain, is $0 \leq V \leq 1.0$. The fourth input is the scalar (negative of the) initial position of the particle L which must correspond in your script to MATLAB variable `L`; the set of allowable instances, or parameter domain, is $0.1 \leq L \leq 1.0$. The fifth input is the scalar final time t_f which must correspond in your script to MATLAB variable `t_final`; the set of allowable instances, or parameter domain, is $0.1 \leq t_f \leq 60$. The sixth input is the error tolerance for the `ode45` integrator which must correspond in your script to the scalar `max_err_tol`; the set of allowable instances, or parameter domain, is $1e-8 \leq \text{max_err_tol} \leq 1e-4$. The script yields two script outputs. The first script output is the `ode45` approximation to $X(t_f)$ — the position of the particle at the final time (even if this position corresponds to positive x and hence beyond the wall) — which must correspond in your script to the scalar `X_at_t_final`. The second script output is t_{wall} — the time at which the particle contacts the wall — which must correspond in your script to the scalar `t_wall`; note if the particle does not reach the wall by t_f then your script should set `t_wall` to `-1`.⁶

A template for the script for this question is provided in `A4Q2_Template`. For this question you must upload both your script `A4Q2` and your function `g_dyn_Q2` (see Guidelines and Hints below) in your `YOURNAME_ASSIGNMENT_4` folder.

Guidelines and Hints. The MATLAB `ode45` code (which you will call from within your script `A4Q2`) will require, in addition to the six script inputs to `A4Q2` described above, a MATLAB anonymous function $\mathcal{O}(t, w)$ which implements the “dynamics” $g(t, w, b, \alpha)$ of (5) (which

⁶To calculate t_{wall} (in the case of contact) we ask that you find the smallest index j such that $\tilde{X}(t^j) \geq 0$ and then set $t_{\text{wall}} = t^j$; here $\tilde{X}(t^j)$ is the `ode45` prediction for the displacement.

you must derive) for the given script inputs b and α ; the signature template for this (anonymous) function is described in the [ode45](#) documentation. For this simple problem you could in fact define this anonymous function directly, however we ask that you follow the approach required more generally for more complicated problems: you first create a standard MATLAB function `g_dyn_Q2(t,w,b,alpha)` which implements $g(t, w, b, \alpha)$; you then in (the input list of) your call to `ode45` create the anonymous function `@(t,w)g_dyn_Q2(t,w,b,alpha)` which has the signature template demanded by `ode45`. (You can not simply call `ode45` with `g_dyn_Q2` as then `b` and `alpha` will not be specified.) Note you may in principle choose any name for this dynamics function; we ask you to call the function `g_dyn_Q2`, and we provide a template in `Assignment_4_Materials`. Recall you *are* required to include your `g_dyn_Q2` function in your `YOURNAME_ASSIGNMENT_4` submission (as when `grade_o_matic_A4` runs your script `A4Q2` your call to `ode45` will involve your `g_dyn_Q2` function — `grade_o_matic_A4` will not provide the `g_dyn_Q2` function).

Challenge 2. This challenge relates to Question 2. You will find that your script `A4Q2` is quite slow for b/α and bt_f very large — and slower and slower as b/α and bt_f get larger and larger. Identify the cause of this difficulty and propose and implement a numerical scheme which addresses the issue: an alternative to explicit high-order Runge-Kutta (the method implemented in `ode45`) which, for the same accuracy, requires much less computation time. Keywords (in the textbook, Unit IV): stiff equations; absolute stability diagrams; explicit Runge-Kutta; implicit methods; overdamped oscillator.

3. (15 points) We now reconsider the problem of Question 2 but rather than a low-Reynolds number Stokes approximation for the drag on the particle we instead consider a high-Reynolds number (bluff-body) approximation for the drag. The governing equation for $X(t)$ is then

$$\begin{cases} \frac{d^2 X}{dt^2} + b \left(\frac{dX}{dt} + \alpha X \right) \left| \frac{dX}{dt} + \alpha X \right| = 0, & 0 < t \leq t_f \\ X(0) = -L, \quad \frac{dX}{dt}(0) = V \end{cases}, \quad (7)$$

where b — related to the high-Reynolds number drag force (divided by the mass of the particle) — now has units of m^{-1} . This equation is of course now *nonlinear*.

We would like you to repeat Question 2 but now consider this nonlinear model. We emphasize that all the allowable instances, deliverables, and Guidelines and Hints from Question 2 should be directly transferred here to Question 3. The “only” change will be to the $g(t, w, b, \alpha)$ which you should implement in MATLAB as `g_dyn_Q3` (we provide a template in `Assignment_4_Materials`). In actual practice, you need only construct `g_dyn_Q3` and then modify very slightly your script `A4Q2` — a small change to the `ode45` call — to obtain your new script `A4Q3` (for this reason we leave `A4Q3_Template.m` blank). You must upload both your script `A4Q3` and your function `g_dyn_Q3` in your `YOURNAME_ASSIGNMENT_4` folder.

Challenge 3. This challenge relates to Question 3. Derive an expression for the exact solution to (7) for the particular case in which $\alpha = 0$ and compare your exact solution to the `ode45` approximation obtained in Question 3.

4. (25 points) In this question we shall consider the stability of a spinning parallelepiped (or “book” for short). We show in Figure 2 our book with semi-axes a , b , and c and principal

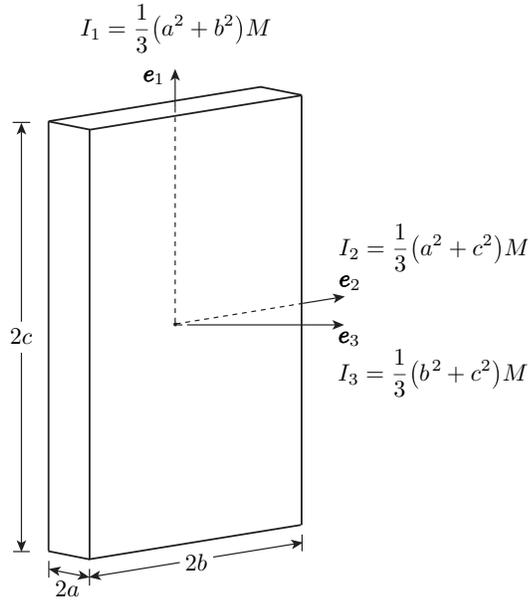


Figure 2: Book Geometry

moments of inertia I_1, I_2 , and I_3 in respectively the $\mathbf{e}_1, \mathbf{e}_2$, and \mathbf{e}_3 directions. In general, the principal moments of inertia are related to the dimensions of the book by

$$\begin{aligned}
 I_1 &= \frac{1}{3}(a^2 + b^2)M \\
 I_2 &= \frac{1}{3}(a^2 + c^2)M \\
 I_3 &= \frac{1}{3}(b^2 + c^2)M,
 \end{aligned} \tag{8}$$

where M is the mass of the book. We shall presume that in general $a < b < c$ such that $I_1 < I_2 < I_3$. We shall consider the particular case in which $a = 1$ cm, $b = 10$ cm, $c = 15$ cm, and $M = 55$ g (in terms of which we can then calculate I_1, I_2 , and I_3 , in units of $\text{g}\cdot\text{cm}^2$).

Euler's equations (in the book frame) for torque-free motion are given by

$$\begin{aligned}
 I_1 \frac{d\omega_1}{dt} &= -\omega_2\omega_3(I_3 - I_2) \\
 I_2 \frac{d\omega_2}{dt} &= -\omega_3\omega_1(I_1 - I_3) \\
 I_3 \frac{d\omega_3}{dt} &= -\omega_1\omega_2(I_2 - I_1),
 \end{aligned} \tag{9}$$

where $\omega = \omega_1\mathbf{e}_1 + \omega_2\mathbf{e}_2 + \omega_3\mathbf{e}_3$ is the angular velocity vector in the book frame. (Hence, for example, ω_1 represents the rotation about the \mathbf{e}_1 axis.)

Assume now that we are given a time-independent, or *equilibrium*, solution to (9), $\bar{\omega}$. We recall the process by which we determine the *stability* of this steady solution: we write

$\omega(t) = \bar{\omega} + \omega'(t)$; we insert our expression for $\omega(t)$ into (9) and neglect all products of (the assumed small) “prime” terms⁷ to arrive at the linearized equations

$$B \frac{d\omega'}{dt} = A\omega'(t) \quad (10)$$

for the 3×1 vector ω' (note that A and B are both 3×3 matrices); we assume temporal behavior of the form $\omega'(t) = ve^{\lambda t}$ to arrive at the eigenvalue problem

$$Av = \lambda Bv \quad (11)$$

for eigenvalue λ and (3×1) eigenvector v (there will be three eigenvalues and three associated eigenvectors); we solve our eigenproblem (11) with a call to MATLAB built-in function `eig`⁸; and finally, we interpret λ to assess stability.

As regards the stability interpretation, we recall that if the *real part* of any of the three eigenvalues λ is positive then the system is unstable — the amplitude of $\omega'(t)$ is exponentially growing in time — and will rapidly depart from the equilibrium solution $\bar{\omega}$. On the other hand, if the real part of all three eigenvalues λ is negative, then the steady solution is stable and will persist. The neutral or marginal case — in which the real part of the eigenvalue λ with largest real part is zero — would require further attention to better understand dissipation and also possibly higher-order corrections.⁹

We would like you to write a script which calculates the three eigenvalues λ and renders a stability verdict — stable, marginally stable, or unstable — for each of the three equilibrium solutions: $\bar{\omega}^1 \equiv (1 \ 0 \ 0)^T$ (rotation about the principal direction e_1 , which has the smallest moment of inertia); $\bar{\omega}^2 \equiv (0 \ 1 \ 0)^T$ (rotation about the principal direction e_2 , which has the intermediate moment of inertia); and $\bar{\omega}^3 \equiv (0 \ 0 \ 1)^T$ (rotation about the principal direction e_3 , which has the largest moment of inertia). (It is simple to deduce that each of these equilibria is indeed a *time-independent* solution of (9).) We are also interested, for each equilibrium, in \mathcal{F} , the imaginary part of the eigenvalue λ with largest imaginary part; \mathcal{F} is related to the frequency of oscillations about the equilibrium. (Note that \mathcal{F} is a real number — the real number which multiplies $i \equiv \sqrt{-1}$ in λ .)

As a first step you will need to perform the requisite linearizations of (9) to deduce the matrices A and B of (10) and (11) — note that A and B are different for each of the three equilibrium solutions. We would also invite you as an electively final step to confirm your stability conclusions (or in the case of marginal stability, resolve your stability quandary) based on experiments with the official soft-matter “2.086 book” with the particular moments of inertia given above.

Your script will take no script inputs (you should assign the indicated moments of inertia, in units of $\text{g}\cdot\text{cm}^2$, within your script). The script should yield six script outputs. The first output is a scalar \mathcal{F}^1 which corresponds to \mathcal{F} (defined above) associated with the first equilibrium $\bar{\omega}^1$ and which must correspond in your script to MATLAB variable `lam_imag_max_1`; the second output is an integer `stability_equi_1` which is -1 if this first equilibrium (based

⁷Note by definition of an equilibrium, the products of “bar” terms will also vanish; we consider particular equilibria below.

⁸Note that (11) is a generalized eigenproblem and hence you will need to call `eig` with two arguments, `eig(A,B)`.

⁹For our problem here, the addition of drag terms to our lossless model (9) would most likely shift eigenvalues to the left in the complex plane and thus “stabilize” a marginally stable equilibrium. However, you should not make this presumption in your analysis.

on the linear stability analysis) is stable, 0 if this equilibrium is marginally stable, or 1 if this equilibrium is unstable, and which must correspond in your script to MATLAB variable `stability_equi_1`. The third output is a scalar \mathcal{F}^2 which corresponds to \mathcal{F} (defined above) associated with the second equilibrium $\bar{\omega}^2$ and which must correspond in your script to MATLAB variable `lam_imag_max_2`; the fourth output is an integer `stability_equi_2` which is -1 if this second equilibrium (based on the linear stability analysis) is stable, 0 if this equilibrium is marginally stable, or 1 if this equilibrium is unstable, and which must correspond in your script to MATLAB variable `stability_equi_2`. Finally, the fifth output is a scalar \mathcal{F}^3 which corresponds to \mathcal{F} (defined above) associated with the third equilibrium $\bar{\omega}^3$ and which must correspond in your script to MATLAB variable `lam_imag_max_3`; the sixth output is an integer `stability_equi_3` which is -1 if this third equilibrium (based on the linear stability analysis) is stable, 0 if this equilibrium is marginally stable, or 1 if this equilibrium is unstable, and which must correspond in your script to MATLAB variable `stability_equi_3`.

A template for the script for this question is provided in `A4Q4_Template`.

5. (10 points) This question relates to Question 1 for the particular case in which we take $\lambda = -0.1$, $f(t) \equiv e^{\sigma t}$ for $\sigma = 2$, $t_f = 2$, and $u_0 = 1$. We ask that you deduce the answers from theoretical considerations but of course we also encourage you to then corroborate and confirm your conclusions with your script of Question 1.

(i) (2.5 points) The exact solution is given by

$$(a) \quad u(t) = \frac{e^{\sigma t} - 1}{\sigma} + e^{\lambda t}$$

$$(b) \quad u(t) = \frac{e^{\sigma t} - e^{\lambda t}}{\sigma - \lambda} + e^{\lambda t}$$

$$(c) \quad u(t) = \frac{e^{\sigma t}}{\sigma}$$

$$(d) \quad u(t) = \frac{e^{\sigma t}}{\sigma - \lambda} + e^{\lambda t}$$

(ii) (5 points) The *a priori* bound for the error $|u(t_f) - \tilde{u}(J\Delta t)|$ suggests that in order to obtain an error of roughly 0.01 the timestep Δt should be chosen as

$$(a) \quad 1.2 \times 10^{-8}$$

$$(b) \quad 1.9 \times 10^{-4}$$

$$(c) \quad 5.4 \times 10^{-2}$$

$$(d) \quad 0.2$$

(Note since the *a priori* estimate is an upper bound for the error, the Δt predicted by the theory will in fact be conservative — a bit too small. You can confirm this claim empirically — with your code — if you wish.)

(ii) (2.5 points) As $J \rightarrow \infty$ the error $|u(t_f) - \tilde{u}(J\Delta t)|$ will

- (a) tend to infinity
- (b) asymptote to a constant
- (c) decrease as $1/J$
- (d) decrease as $1/J^2$

The template `A4Q5_Template.m` contains the multiple-choice format required by `grade_o_matic_A4`. Please make sure to use lower-case letters for your multiple-choice selections.

MIT OpenCourseWare
<http://ocw.mit.edu>

2.086 Numerical Computation for Mechanical Engineers
Spring 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.