Assignment 4 — 2.086 Fall 2014

Due: *Wednesday, 19 November at 5 PM.*

Upload your solution to course website as a zip file "`YOURNAME_ASSIGNMENT_4`" which includes the script for each question *as well as* all MATLAB® functions (of your own creation) called by your scripts; both scripts and functions must conform to the formats described in **Instructions** and **Questions** below.

Instructions

Download (from the course website Assignment 4 page) the `Assignment_4_Templates` 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_Templates` folder also contains the `grade_o_matic` files for Assignment 4 (please see Assignment 1 for a description of `grade_o_matic`[1]) as well as all `.mat` files which you will need for Assignment 4.

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` 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 for 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 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 to course website you run `grade_o_matic` (from your YOURNAME_ASSIGNMENT_4 folder) for final confirmation that all is in order.

[1]Note that, for display in verbose mode, `grade_o_matic` will "unroll" arrays and present as a row vector.

Note that, in Assignment 4, the templates provide rather little in the way of hints: you must largely design your own code. You can start with the mathematical statement of the problem and, as appropriate, the numerical method for approximation or estimation or solution. You can then design the logic (or "flow") for your code: the reduction of your method to a sequence of steps — an algorithm. Finally, you should consider the particular MATLAB implementation: the capabilities of MATLAB which you will exploit, and the associated syntax and "built-in" functions.

Questions

1. (*10 points*) This question is intended to exercise the concept of passing one MATLAB function — more precisely, a *function handle* — as an input argument to another MATLAB function. As the vehicle, we will ask you to write a MATLAB function which implements the rectangle, right integration rule to calculate an approximation, I_h, to the integral

$$I = \int_{x_{\min}}^{x_{\max}} f(x)\,dx \;, \tag{1}$$

for any given function f of interest. We ask that you consider a discretization of the interval $(x_{\min}, x_{\max})$ defined by equispaced points, $x_{\min} \equiv x_1, x_2, \ldots, x_N \equiv x_{\max}$: for $h = (x_{\max} - x_{\min})/(N-1)$, points $x_i = x_{\min} + (i-1)h, 1 \le i \le N$, induce segments of length h, $S_i = (x_i, x_{i+1}), 1 \le i \le N-1$. (See the nutshell *Integration* for a full description of the rectangle, right rule.)

In particular, we ask you to create a function rect_right_rule with signature[2]

```
function [I_h]=rect_right_rule(integrand_func,x_min,x_max,N)
```

which evaluates I_h for any given function f, embodied in the function integrand_func, limits of integration x_min and x_max, and number of discretization points N.

The function rect_right_rule takes four function inputs: integrand_func, x_min, x_max, and N. The first argument to rect_right_rule, integrand_func, is a function handle for the (MATLAB embodiment of the) function f to be integrated. Note that MATLAB function integrand_func is the implementation of the mathematical function f in the sense that integrand_func(x) = f(x). The MATLAB function integrand_func may be provided in several forms, as described in the Appendix. In all cases, the function integrand_func must take a single function input, x_vec, and yield a single output, f_vec = integrand_func(x_vec): the input x_vec is a $M \times 1$ array of real numbers, for $2 \le M \le 10000$; the output y_vec is the $M \times 1$ array with entries y_vec(i) = f(x_vec(i)), i= $1, \ldots, M$. The second and third inputs to rect_right_rule are, respectively, $x_{\min}$ (MATLAB scalar x_min), the lower limit of the integration, and $x_{\max}$ (MATLAB scalar x_max), the upper limit of the integration; the input parameters x_min and x_max must be real numbers and satisfy x_min <x_max. The fourth argument is N (MATLAB scalar N), the number of points in our discretization; allowable instances must satisfy $2 \le N \le 10000$.

[2] We recall that the particular names chosen for the inputs and outputs in the function signature/body of a function are 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. (In contrast, in a script, which does not have a private workspace, the specific names of the variables are important.)

This function `rect_right_rule` has a single function output, I_h (MATLAB `I_h`), which is the rectangle, right rule approximation of the integral I defined in (1). Recall that we consider the particular case of equispaced points and hence equisized segments.

A script template is provided in `A4Q1_Template`: you should not modify this template; you must only remove the `_Template` and upload in your `YOURNAME_ASSIGNMENT_4` folder. We also provide a function template in `rect_right_rule_Template`. We emphasize that your function `rect_right_rule` should perform correctly for any set of inputs (and hence any admissible function, `integrand_func`). You should yourself devise several test cases for which you can anticipate the correct answers and hence test `rect_right_rule`; we provide some suggestions in the Appendix. Note the only deliverables for this problem are your script `A4Q1` and your function `rect_right_rule`. In particular, any `integrand_func` functions which you create to test your `rect_right_rule` 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 "`integrand_func`" with which to test your `rect_right_rule` code.

2. (*20 points*) We consider here the scalar first-order ODE IVP

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

for $\lambda \le 0$. 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 $°C$; u_0 is the initial temperature (measured relative to ambient temperature), in $°C$; λ is the (negative of the) heat transfer coefficient times the surface area of the body divided by the heat capacity of the body, in units of s^{-1} ($-\lambda$ is hence the inverse time constant of our first-order system); and $f(t)$ is the heat generation (in Watts) divided by the heat capacity of the body, in units of $°Cs^{-1}$. We note that λ is *non-positive*. The Euler Backward discretization of (2) will yield an approximate solution $u^{\tilde{}j} = u^{\tilde{}}(j\,\Delta t)(\approx u(j\,\Delta t))$, $0 \le j \le J$, 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, `u_vec(j)` $= \tilde{u}($`(j-1)`$\,\Delta t)$, `1` $\le$ `j` $\le$ `J+1`, for prescribed initial condition `u_0`, parameter `lambda`, "source" function `f_source`, and final time `t_final` ($= t_f$).

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 u_0 (MATLAB scalar `u_0`), the initial condition in (2); the set of allowable instances, or parameter domain, is not restricted (any finite value is admissible). The second input is λ (MATLAB scalar `lambda`), the inverse time constant in (2); the set of allowable instances, or parameter domain, is the *non-positive* real numbers. The third input is the MATLAB function `f_source` which "implements" the source function $f(t)$ in (2) in the sense that `f_source(t)` $= f($`t`$)$. The function `f_source` must take a single input `t_vec` and yield a single output, `f_vec`: the input `t_vec` is a real $M \times 1$ array; the output `f_vec` is the $M \times 1$ array with entries `f_vec` (i)= f(`t_vec(i)`), $1 \leq$ `i` $\leq M$. The fourth input is the t_f (MATLAB scalar `t_final`), the final time of integration 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 J (MATLAB scalar `J`), the number of intervals 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 $\tilde{u}(j\,\Delta t)$, $0 \leq j \leq J$ (MATLAB `J+1` $\times$ 1 array `u_vec`), approximate solution to our ODE IVP.[3]

A script template is provided in `A4Q2_Template`: you should not modify this template; you must only remove the `_Template` and upload in your `YOURNAME_ASSIGNMENT_4` folder. We also provide a function template in `Euler_Backward_Template`. We emphasize that your function `Euler_Backward` should perform correctly for any set of inputs (and hence any admissible function `f_source`). You should yourself devise several test cases for which you can anticipate the correct answers and hence test `Euler_Backward`. Note the only deliverables for this problem are your script `A4Q2` and your `Euler_Backward` function. In particular, any "`f_source`" functions which you create — named or anonymous — 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.

3. (*20 points*)

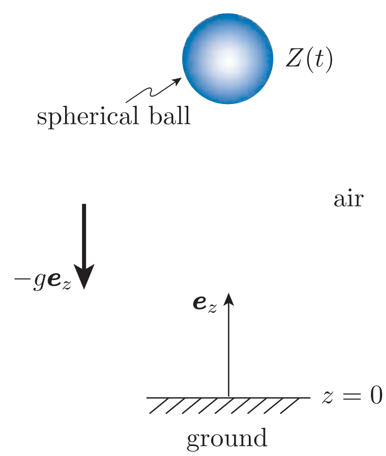

Figure 1: A ball falling under the action of gravity and subject to aerodynamic drag.

We consider the trajectory of a spherical ball falling under the action of gravity, as shown in Figure 1. We choose our coordinate system such that the acceleration of gravity is given by $-g\boldsymbol{e}_z$ for $g = 9.81$ m/s. The position (height) of the ball as a function of time t, relative to the ground at $z = 0$, is then denoted $Z(t)\boldsymbol{e}_z$. The position $Z(t)$ can be reasonably described

[3]Note 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)`.

by the second-order ODE IVP

$$\begin{cases} \dfrac{d^2Z}{dt^2} + \alpha \, \dfrac{dZ}{dt} \, \left| \dfrac{dZ}{dt} \right| = -g, \quad 0 < t \le t_f \\ Z(0) = Z_0, \; \dfrac{dZ}{dt}(0) = \dot{Z}_0 \end{cases} ; \tag{3}$$

we shall consider initial conditions $Z_0, \dot{Z}_0$ and final times t_f such that $Z(t) > 0$ for all $0 < t \le t_f$. The first and third terms in the ODE should look familiar; the second term, $\alpha \frac{dZ}{dt} \left| \frac{dZ}{dt} \right|$, models the effect of aerodynamic drag on the ball. The parameter α, related to the "ballistic coefficient," is given by

$$\alpha \equiv \frac{\frac{1}{2} C_D \, \rho_{\text{air}} \, A_{\text{frontal}}}{m} ,$$

where C_D is the drag coefficient, ρ_{air} is the density of air, A_{frontal} is the projected area of the ball in the direction of motion (hence $A_{\text{frontal}} \equiv \pi r_{\text{ball}}^2$, for r_{ball} the radius of the ball), and m is the mass of the ball. (We presume the density of the ball is large compared to the density of the air.) We shall consider the high–Reynolds numbers flow regime for which the drag coefficient C_D is relatively insensitive to velocity and furthermore roughly equal to 1/2. We shall measure time in seconds (s) and position Z in meters (m); the units of α are then m^{-1}.

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

$$\begin{cases} \dfrac{dw_1}{dt} = g_1(t, w, \alpha) \\ \dfrac{dw_2}{dt} = g_2(t, w, \alpha) \end{cases} , \; 0 \le t \le t_f \; , \tag{4}$$

or even more succinctly as

$$\frac{dw}{dt} = g(t, w, \alpha) \; , \; 0 \le t \le t_f \; , \tag{5}$$

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

$$w(t = 0) \equiv (Z_0 \;\; \dot{Z}_0)^{\text{T}} \; . \tag{6}$$

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

Your script must take three script inputs. The first script input is the coefficient α, which must correspond in your script to MATLAB scalar variable `alpha`; the set of allowable instances, or parameter domain, is $0 \le \alpha \le 2000$. The second input is the position and velocity of the ball at release, $(Z_0, \dot{Z}_0)$, which must correspond in your script to MATLAB 2×1 array `w_0`; allowable instances must satisfy $.5 \le Z_0 \le 4.0$ m and $-2 \le \dot{Z}_0 \le 2$. The third input is the final time t_f, which must correspond in your script to MATLAB scalar variable `t_final`; the set of allowable instances, or parameter domain, is $.2 \le t_f \le 20$. The script yields a single script output: the output is the `ode45` approximation to $Z(t_f)$ — the position of the particle at the final time — which must correspond in your script to the MATLAB variable `Z_at_t_final`.

The MATLAB ode45 code (which you will call from within your script `A4Q3`) will require, in addition to the three script inputs to `A4Q3` described above, a MATLAB anonymous function `@(t,w)` which implements the "dynamics" $g(t, w, \alpha)$ of (5) for the given script input α. We suggest that you first create a standard named MATLAB function `g_falling_ball(t,w,alpha)` which implements $g(t, w, \alpha)$; then, directly in the input list of your call to ode45, you form the anonymous function `@(t,w)g_falling_ball(t,w,alpha)`. (You can not simply call ode45 with `g_falling_ball` as then `alpha` will not be specified.)

A template for the script for this question is provided in `A4Q3_Template`; we also provide a function template in `g_falling_ball_Template`. For this question you must upload in your `YOURNAME_ASSIGNMENT_4` folder both your script `A4Q3` and your function `g_falling_ball`; `grade_o_matic_A4` will *not* provide the `g_falling_ball` function.

The Last Instance. In this question there are three `grade_o_matic` grader input instances: the first two are each worth 5 points, the third is worth 10 points, for a total of 20 points. There are only two student input instances, in particular, the first two grader instances. However, we give you here an alternative fashion by which to (more or less) confirm that, for the third grader instance, your code yields the correct output: comparison with experiment.

We again invoke the falling ball experimental data — height as a function of time t for $0 < t \leq t_f^{\text{exp}} \equiv 0.6$ s — developed by Dr James Penn. Dr Masa Yano has performed a nonlinear least squares procedure on the data to obtain optimal values of Z_0^{opt}, $\dot{Z}_0^{\text{opt}}$, and α^{opt} for which the corresponding solution to (3), $Z(t)$, very well replicates the experiment: $Z_0^{\text{opt}} = 1.997$ m, $\dot{Z}_0^{\text{opt}} = -0.428$ m/s, and $\alpha^{\text{opt}} = 0.0444$ m^{-1}. The non-zero value of $\dot{Z}_0^{\text{opt}}$ reflects an actual release time a fraction of a second prior to time $t = 0$. The value of α^{opt} can in fact be independently reproduced, to within several percent, from first principles. We can thus be reasonably confident that there is a physical basis for the optimal parameter values obtained.

The script `The_Last_Instance_Q3.m` plots three quantities as a function of time for $0 < t \leq t_f = t_f^{\text{exp}} \equiv 0.6$ s: the experimental measurements; $Z(t)$ of (3) for the optimal (physical) parameters $Z_0 = 1.997$ m, $\dot{Z}_0 = -0.428$ m/s, $\alpha = 0.0444$ m^{-1}, and $t_f = 0.6$ s as predicted by *your* script `A4Q3` — which we denote "simulation: drag included"; and the solution to (3) for the good initial height and velocity, $Z_0 = 1.997$ m, $\dot{Z}_0 = -0.428$ m/s, but α set to zero — which we denote "simulation: drag neglected." (Note the drag-neglected solution can be expressed in closed form: $Z(t) = 1.997 - 0.428t - (1/2)gt^2$.) We also directly plot your script output as the point (0.6 s, `Z_at_t_final`). In all cases we take for the magnitude of the acceleration gravity $g = 9.81$ m/s.

It is very good evidence that your code is correctly predicting the output for the third grader input instance if the "simulation: drag included" results in the figure produced by `The_Last_Instance_Q3` lie directly on top of the experimental measurements. You will also notice that, as expected, the "simulation: drag neglected" result is noticeably below the experimental measurements.

4. (*5 points*) In this question we would like you to reconsider Question 3 but now apply, in the place of ode45, the MATLAB ODE integrator ode23s. In other words, this question, Question 4, is exactly Question 3 but now with "ode45" replaced everywhere by "ode23s". Note you need not in any way change `g_falling_ball`.

   A template for the script for this question is provided in `A4Q4_Template` — though in fact it is better that you take as your point of departure `A4Q3`, which then requires only very minor modification to form A4Q4. Note your Question 4 you need only upload your script A4Q4 since you will have already provided function g_falling_ball in YOURNAME_ASSIGNMENT_4 folder as part of the deliverable for Question 3.

Elective Exercise: (*0 points*) The (grader, and student) grade_o_matic input instances are the same for Question 3 and Question 4. You should find in Question 3, for the second (student) input instance, that many time steps are required by ode45. In contrast, you should find in Question 4, for this same second (student) input instance, that very few time steps are required by ode23s (to obtain roughly the same accuracy). Can you explain this difference? Note that in the second input instance $\alpha = 1000$. Hint: You may wish to look at the temporal evolution of $\frac{dZ}{dt}$, the ball velocity, in particular for very short times.

5. (*25 points)* We consider the motion of a particle in an air flow in two space dimensions, as shown in Figure 2. We denote the particle position as a function of time t by the 2-vector $(X(t), Y(t))$ which we can also express as $X(t)\boldsymbol{e}_x + Y(t)\boldsymbol{e}_y$ for $\boldsymbol{e}_x$ and $\boldsymbol{e}_y$ the unit vectors associated with the x and y Cartesian coordinates, respectively. We denote the prescribed steady fluid velocity field by the 2-vector $\mathbf{U}(x, y) \equiv (U_x(x, y), U_y(x, y))$ which we can also express as $\mathbf{U}(x, y) = U_x(x, y)\boldsymbol{e}_x + U_y(x, y)\boldsymbol{e}_y$. It is important to note that $(X(t), Y(t))$ is the position of the particle at time t — a particular point in space — whereas $(U_x(x, y), U_y(x, y))$ is the velocity *field* — the (steady) fluid velocity vector at all points (x, y) in space.

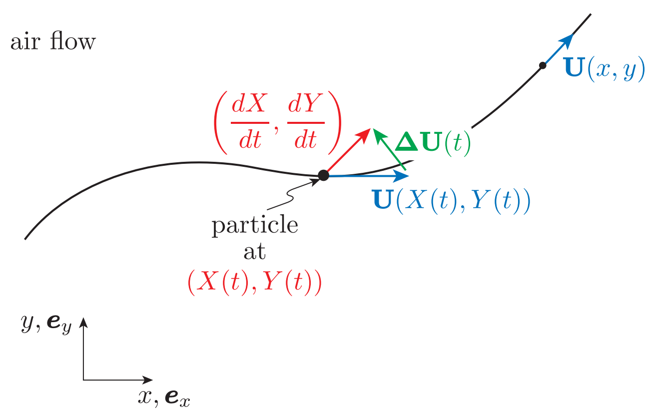

Figure 2: A particle at position $(X(t), Y(t))$ in an air flowfield $\mathbf{U}(x, y)$.

The motion of the particle can be reasonably described by the pair of coupled second-order ODE IVPs

$$\begin{cases} \dfrac{d^2X}{dt^2} + \alpha\, \boldsymbol{\Delta}\mathbf{U}_x \, |\boldsymbol{\Delta}\mathbf{U}| \;=\; 0, \\[2ex] \dfrac{d^2Y}{dt^2} + \alpha\, \boldsymbol{\Delta}\mathbf{U}_y \, |\boldsymbol{\Delta}\mathbf{U}| \;=\; 0 \end{cases} \qquad 0 < t \le t_f \,, \tag{7}$$

subject to initial conditions

$$X(0) = X_0,\ \frac{dX}{dt}(0) = \dot{X}_0\,;\ Y(0) = Y_0,\ \frac{dY}{dt}(0) = \dot{Y}_0\,. \tag{8}$$

Here α, related to the ballistic coefficient, is the ratio of drag forces to inertial effects , and

$|\mathbf{\Delta U}|$ is the velocity difference vector

$$\mathbf{\Delta U}(t) \equiv (\mathbf{\Delta U}_x, \mathbf{\Delta U}_y) \equiv \left(\frac{dX}{dt} - U_x(X(t), Y(t))\, , \, \frac{dY}{dt} - U_y(X(t), Y(t)) \right) \, .$$

In words, and as depicted in Figure 2, $\mathbf{\Delta U}(t)$ is the different between the velocity (vector) of the particle and the velocity (vector) of the fluid at the particular point in space at which the particle is located at time t, $(X(t),\ Y\ (t))$. At higher Reynolds number, the drag force is proportional to the square of the magnitude of $\mathbf{\Delta U}$, $|\mathbf{\Delta U}|^2$ (which we can also write as$\|\mathbf{\Delta U}\|^2$), and in the direction $-\mathbf{\Delta U}$, as summarized in (7).[4]

We make several remarks about this model. Prediction of the motion of particles in flows is important in many technological applications such as industrial separators (and, more mundanely, vacuum cleaners). For simplicity we do not include gravity in our model, though in actual practice "settling" (say in the third direction) can play an important role in separation applications. Our model for the forces on the particle is reasonable under certain assumptions: the density of the particle should be large compared to the density of air (such that we may ignore buoyancy); the particle should be of size small compared to the length scale over which the flow field varies (such that we may ignore unsteady drag effects). Finally, we note that (7) is nondimensionalized based on a characteristic length scale and characteristic velocity scale associated with the fluid flow.

We denote our state variable as $w \equiv (w_1\ \ w_2\ \ w_3\ \ w_4)^{\mathrm{T}}$ for $w_1 \equiv X$, $w_2 \equiv \frac{dX}{dt}$, $w_3 \equiv Y$, $w_4 \equiv \frac{dY}{dt}$. It is then possible to express (7) as

$$\left\{ \begin{aligned} \frac{dw_1}{dt} &= g_1(t, w, \alpha, \mathbf{U}) \\ \frac{dw_2}{dt} &= g_2(t, w, \alpha, \mathbf{U}) \\ \frac{dw_3}{dt} &= g_3(t, w, \alpha, \mathbf{U}) \\ \frac{dw_4}{dt} &= g_4(t, w, \alpha, \mathbf{U}) \end{aligned} \right. \qquad 0 < t \le t_f, \tag{9}$$

or even more succinctly as

$$\frac{dw}{dt} = g(t, w, \alpha, \mathbf{U})\ ,\ 0 \le t \le t_f\ , \tag{10}$$

where $g(t,\, w,\, \alpha,\, \mathbf{U}) \equiv (g_1(t,\, w,\, \alpha,\, \mathbf{U})\ g_2(t,\, w,\, \alpha,\, \mathbf{U})\ g_3(t,\, w,\, \alpha,\, \mathbf{U})\ g_4(t,\, w,\, \alpha,\, \mathbf{U}))^{\mathrm{T}}$; you will need to derive the form of $g(t,\, w,\, \alpha,\, \mathbf{U})$ from (7). We provide w with the initial conditions prescribed in the problem statement,

$$w(t = 0) \equiv (X_0\ \ \dot{X}_0\ \ Y_0\ \ \dot{Y}_0)^{\mathrm{T}}\ . \tag{11}$$

In fact, we shall exclusively consider a particular form of the initial conditions (11) in which the initial partial velocity is the velocity of the fluid at $(x,\, y) = (X_0,\, Y_0)$,

$$w(t = 0) \equiv (X_0\ \ \ U_x(X_0, Y_0)\ \ Y_0\ \ U_y(X_0, Y_0))^{\mathrm{T}}\ . \tag{12}$$

[4]Note that for smaller particles and slower flows, we can replace our quadratic drag model with a linear Stokes model.

Our interest is the trajectory of the particle but more particularly in

$$R_f \equiv \sqrt{(X(t_f))^2 + (Y(t_f))^2}\,, \tag{13}$$

which is the distance of the particle from the origin at the final time.

We would like you to write a "driver" function with signature

```
function [final_radial_position] = ...
                         driver_func_Q5(alpha,X_0,Y_0,t_final,Flowfield,vis_true)
```

which solves (approximately) (10) with the MATLAB built-in function `ode45` to yield the `final_radial_position` for given parameter `alpha`, initial particle location (`X_0`,`Y_0`), final time `t_final`, and fluid velocity field $\mathbf{U}$ (embodied in MATLAB function `Flowfield`). (The final input, `vis_true` will allow you to visualize your solution if you are so inclined. The visualization includes the particle trajectory but also flow streamlines, which you can interpret as the trajectory of fluid elements. The visualization capability was developed in collaboration with Dr Masa Yano.)

Your function `driver_func_Q5` will take six function inputs. The first input is the parameter α (MATLAB scalar variable `alpha`); the set of allowable instances, or parameter domain, is $.01 \leq \alpha \leq 10$. The second and third inputs are the position in x and y of the particle at release, X_0 and Y_0 (MATLAB scalar variables `X_0` and `Y_0`, respectively); the set of allowable instances, or parameter domain, is $-6 \leq X_0 \leq 2$ and $-1 \leq Y_0 \leq 1$. The fourth input is the scalar final time t_f (MATLAB scalar variable `t_final`); the set of allowable instances, or parameter domain, is $.1 \leq t_f \leq 150$. The fifth input is the MATLAB function `Flowfield` which implements the fluid velocity field $\mathbf{U}$ in the sense that `Flowfield(x,y)` $= \mathbf{U}$(`x`,`y`). The function `Flowfield` must take two inputs, `x` and `y`, and yield a single output, `Vel_Vect`: the inputs `x` and `y` are each scalars; the output `Vel_Vect` is the 1×2 array `[`U_x(`x`,`y`)$,U_y$(`x`,`y`)`]`. The sixth and final input, `vis_true`, is a logical MATLAB variable which, if set to `true`, provides a plot of the particle trajectory; this capability is included solely for your viewing pleasure and perhaps debugging assistance. Note that `grade_o_matic` will set `vis_true = true`. The function `driver_func_Q5` yields a single output: this single output is R_f of (13) (MATLAB scalar variable `final_radial_position`), the `ode45` approximation to the radial position of the particle at the final time.

The MATLAB code `ode45` (which you will call from within your function `driver_func_Q5` will require a MATLAB function which implements the "dynamics" $g(t, w, \alpha, \mathbf{U})$ of (10). We suggest that you first create a standard named MATLAB function

```
function [w_dot] = g_particle_in_flow(t,w,alpha,Flowfield)
```

which implements $g(t, w, \alpha, \mathbf{U})$; then, directly in the input list of your call to `ode45`, you may form the dynamics function (of the input-output form required by `ode45`) as

```
@(t,w)g_particle_in_flow(t,w,alpha,Flowfield) % note Flowfield, not @Flowfield
```

for the particular value of `alpha` and the particular function `FlowField` of interest. (Note that you require `Flowfield`, not `@Flowfield`, in this anonymous function definition: Flowfield in `driver_func_Q5` is already a function handle.)

A template for the script for this question is provided in `A4Q5_Template`: you should not modify this template; you must only remove the `_Template` and upload in your `YOUR_ASSIGNMENT_4` folder. We also provide you with function templates in `driver_func_Q5_Template` and `g_particle_in_flow_Template`. Note that for this question you must upload in your folder `YOURNAME_ASSIGNMENT_4` your script `A4Q5` as well as your functions `driver_func_Q5` and `g_particle_in_flow`; `grade_o_matic_A4` will *not* provide either the `driver_func_Q5` function or the `g_particle_in_flow` function.

In this problem we appreciate that it will be difficult for you to create your own velocity fields with which to test your code. Thus in this question the grader and student input instances coincide. The three `grade_o_matic` input instances correspond to potential flows: flow past a cylinder, a sink-vortex flow, and a stagnation flow. In all three cases, `grade_o_matic` will present you with visualizations of the flow field as streamlines. (Although these flowfields are not overly realistic, the particular particle trajectories tested by `grade_o_matic` are in fact quite reasonable.) We ask that you retain the MATLAB files `Cylinder_PF.m` and `Cyclone_PF.m` in your `YOURNAME_ASSIGMENT_4` folder (in order that `grade_o_matic` can function properly).

6. (*20 points*)

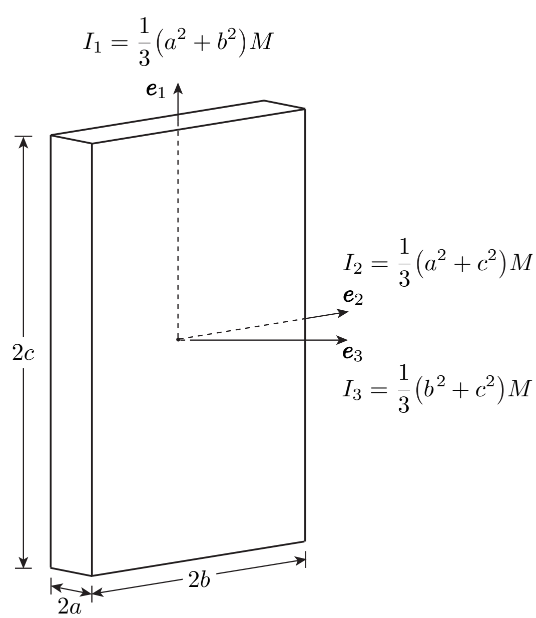

Figure 3: Local coordinate frame and moments of inertia for a parallelepiped book.

In this question we shall consider the stability of a spinning parallelepiped (or "book" for short). We show in Figure 3 our book with semi-axes a, b, and c and principal moments of inertia I_1, I_2, and I_3 in respectively the $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, and $\boldsymbol{e}_3$ directions. In general, the principal moments of inertia are related to the dimensions of the book by

$$\begin{aligned} I_1 &= \frac{1}{3}(a^2 + b^2)M \\ I_2 &= \frac{1}{3}(a^2 + c^2)M \\ I_3 &= \frac{1}{3}(b^2 + c^2)M, \end{aligned} \tag{14}$$

where M is the mass of the book. The units for length and mass are cm and g such that the units for the moment of inertia are g-cm^2.

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

$$\begin{aligned} \frac{d\omega_1}{dt} &= -\omega_2\omega_3(I_3 - I_2)/I_1 \\ \frac{d\omega_2}{dt} &= -\omega_3\omega_1(I_1 - I_3)/I_2 \\ \frac{d\omega_3}{dt} &= -\omega_1\omega_2(I_2 - I_1)/I_3 \end{aligned} \tag{15}$$

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

Assume now that we are given a time-independent, or *equilibrium*, solution to (15), $\overline{\omega}$; in fact, as we describe below, there are three equilibrium solutions to (15). We recall the process (see the nutshell video *Stability*) by which we determine the *stability* of an equilibrium solution $\overline{\omega}$: we write $\omega(t) = \overline{\omega} + \omega'(t)$; we insert this expression for $\omega(t)$ into (15); we neglect all products of (the assumed small) "prime" terms[5] to arrive at the linear(ized) equations

$$\frac{d\omega'}{dt} = A\omega'(t) \tag{16}$$

where ω' is 3×1 vector and A (which will depend on $\overline{\omega}$) is a 3×3 matrix; we assume temporal behavior of the form $\omega'(t) = ve^{\lambda t}$ to arrive at the eigenvalue problem

$$Av = \lambda v \tag{17}$$

for complex eigenvalue λ and (3×1) complex eigenvector v (there will be three eigenvalues and three associated eigenvectors); we solve our eigenproblem (17) with a call to MATLAB built-in function `eig`; 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 corresonding equilibrium solution $\overline{\omega}$. 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

[5] Note by definition of an equilibrium, the products of two "bar" terms will also vanish; we consider particular equilibria below.

dissipation and also possibly higher-order (nonlinear) corrections. (For our problem here, the addition of drag terms to our lossless model (15) would most likely shift eigenvalues to the left in the complex plane and thus "stabilize" a marginally stable equilibrium. However, we should not make this presumption without further investigation.) In practice, we might also be interested in the imaginary part of the eigenvalues, which is related to the frequency of oscillations about the equilibrium, or "wobble."

We would like you to write a script which calculates the eigenvalues λ for each of the three equilibrium solutions of (15): $\omega^1 \equiv (1\ 0\ 0)^{\mathrm{T}}$ (rotation about the principal direction $\boldsymbol{e}_1$, which has the smallest moment of inertia); $\omega^2 \equiv (0\ 1\ 0)^{\mathrm{T}}$ (rotation about the principal direction $\boldsymbol{e}_2$, which has the intermediate moment of inertia); and $\omega^3 \equiv (0\ 0\ 1)^{\mathrm{T}}$ (rotation about the principal direction $\boldsymbol{e}_3$, which has the largest moment of inertia). (It is simple to deduce that each of these equilibria is indeed a *time-independent* solution of (15).) Hence as a first step you will need to perform the requisite linearizations of (15) to deduce the matrix A of (16) and hence the eigenproblem (17) — note that A will be different for each of the three equilibrium solutions.

The script will take four script inputs: real positive numbers a, b, c, and M, which must correspond to MATLAB (scalar) variables `a`, `b`, `c`, and `M`, respectively; allowable instances must satisfy $0 < a \leq b \leq c$ and $0 < M$.

The script will yield three script outputs. For each equilibrium, $\overline{\omega}^i$, $i = 1, 2, 3$, we will obtain three eigenvalues, λ^i_j, $j = 1, 2, 3$. We may then define

$$G^i = \max_{j \in \{1,2,3\}} \Re(\lambda^i_j), \quad i = 1, 2, 3\ , \tag{18}$$

where $\Re(z)$ refers to the real part of a complex number z. (In words, G^i is the maximum real part — which governs stability — over all three eigenvalues associated with the i^{th} equilibrium.) Our three outputs are then the three real numbers G^1, G^2, and G^3, which must correspond to MATLAB (scalar, real) variables `G_1`, `G_2`, and `G_3`, respectively.

A template for the script for this question is provided in `A4Q6_Template`.

Elective Exercise. (*0 points*) We would also invite you as an elective final step to confirm your stability conclusions — as to which equilibria will spin stably and which will rapidly deviate into apparently "random" motion— by experiments with a physical parallelepiped. You might also attempt to confirm your predictions for the frequency of stable wobble. To avoid damage to a book or injury to your person we provide in Rm 3-264 an official soft-matter 2.086 pseudo-book with dimensions $a = 1$ cm, $b = 10$ cm, $c = 15$ cm, and mass $M = 55$ g.

Appendix to Question 1: Function Handles

We suggest you test `rect_right_rule` — and your understanding of function handles — in several ways. We consider below the case in which the function we wish to integrate, $f(x)$, is given by $f(x) = x^2$. The input `integrand_func` to `rect_right_rule` (which of course need not be named `integrand_func`) can be created in several ways.

First, you can create a "named" function `myfunc_1` defined in a file `myfunc_1.m` given by (say)

```
function [integrand_values] = myfunc_1( x_vec )
integrand_values = x_vec.^2;
end
```

and then call `rect_right_rule(@myfunc_1,0,pi,200)`. Here `@myfunc_1` is the handle (a specific instance of `integrand_func`) of the function `myfunc_1`; in essence, the function handle `@myfunc_1` tells MATLAB where to find the code which implements the function `myfunc_1`.

As an alternative to a named function, we may also create an anonymous function

```
myfunc_2 = @(x) x.^2
```

and then call `rect_right_rule(myfunc_2,0, pi,200)` — note the absence of "@" since for an anonymous function the "name" is directly the handle.

Finally, we may create an anonymous (or named) function with "additional" parameters,

```
myfunc_3 = @(x,p) x.^p
```

and then call `rect_right_rule(@(x)myfunc_3(x,2),0,pi,N)` — we create the specific anonymous function, with inputs expected by `rect_right_rule`, directly in the call to `rect_right_rule`. Note the latter is a convenient way to effectively pass parameter values to functions within functions.