Assignment 1 — 2.086 Fall 2014

Due: *Monday, 22 September, at 5 PM.*

Upload your solution to course website as a zip file named "`YOURNAME_ASSIGNMENT_1.zip`" which includes the script for each question in the proper format as well as any MATLAB® functions (of your own creation) your scripts may call.

Instructions

You should download (from the course website Assignment 1 page) the `Templates_Assignment_1` folder. This folder contains a script template for each of the questions in Assignment 1: the script template `AxQy_Template.m` will serve as the point of departure for your own script for Question `y` of Assignment `x`. The templates will help you follow the required format and furthermore provide specific implementation hints; the "?" in the templates indicate the additional pieces which you must provide. In later assignments we will provide function templates as well. Note the Templates_Assignment_1 folder will also contain several other files: `grade_o_matic.m`, `grade_o_matic_function_.p`, and `g_o_m_assignment1.dat`. You will directly use `grade_o_matic` — an automatic grading program — as described below; you will not directly use either `grade_o_matic_function_.p` or `g_o_m_assignment1.dat`.

You should perform all your work for Assignment 1 in the folder `Templates_Assignment_1`: this is important to ensure that MATLAB can find all the correct files for execution of `grade_o_matic` for Assignment 1 (in particular, `grade_o_matic_function_.p` and `g_o_m_assignment1.dat`). To develop your solution to Question `y`, first open the file `AxQy_Template.m` and "save as" `AxQy.m`. Next, make your insertions and modifications to `AxQy.m` to create your script for Question `y`; you will certainly need to test and debug your code, a process in which `grade_o_matic` can play a role. Finally, when you have completed all the questions, you should rename the folder `Templates_Assignment_1` to `YOURNAME_ASSIGNMENT_1`, run `grade_o_matic` one last time (from the `YOURNAME_ASSIGNMENT_1` folder), compress your `YOURNAME_ASSIGNMENT_1` folder with the "zip" utility, and then upload the `YOURNAME_ASSIGNMENT_1.zip` folder to course website.

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`. (Hence our suggestion that, to start, you should open `AxQy_Template.m` and "save as" `AxQy.m`.)

(b.) For each question and hence each script we will identify *input parameters*; we will also identify corresponding *allowable instances* for the parameters — the parameter values or "parameter domains" for which the code must work. More strictly speaking, inputs are typically defined for a function (in an argument list), which we will learn about shortly, rather than a script; however, less strictly speaking, we can also define inputs for scripts — variables which must be assigned values (in the workspace) before the script is executed. Your script should perform correctly for any allowable instances.[1] We may similarly and informally associate to each

[1] Note that most properly, and certainly if you write code for third parties, you should always confirm that any

script an *output* or *outputs*.

(c.) We shall define the inputs and outputs and associated specific MATLAB variable names for each question as we proceed. The 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.

(d.) Finally, we ask that in the submitted version of your script you suppress all display by placing a "`;`" at the end of each line of your script. (Of course during debugging you will often choose to display many intermediate and final results.)

So what is this thing called `grade_o_matic`? The `grade_o_matic` software will be used both by you, the student, and by the graders: in the former case, "student mode," `grade_o_matic` runs your code for a (relatively small) set of "student instances" of the input parameters; in the latter case, "grader mode," `grade_o_matic` runs your code for a (larger and more rigorous) set of "grader instances" of the input parameters. Note that the student instances of the input parameters are in all cases a subset of the grader instances of the input parameters. The student mode serves to help you develop your code; the grader mode — not accessible to students — serves to assign and record your scores. In general, for each question, the total points available are allocated uniformly over all the outputs and all the grader instances of the input parameters.

In student mode, you can take advantage of `grade_o_matic` (*i*) to verify that your inputs and outputs are in the proper format, [2] and (*ii*) to confirm that your code is performing correctly —for the *student* instances of the input parameters. You can run `grade_o_matic` for any single auto-gradable question, [3] say Question 3, as `grade_o_matic(3)`, or with more verbose display as `grade_o_matic(3,'v')`; alternatively, you can run `grade_o_matic` for all auto-gradable questions as simply `grade_o_matic` (with no arguments). The command `grade_o_matic(3,'v')` will indicate (*a*) for each student instance, whether or not your script `A1Q3.m` yields the correct value for the output(s),(*b*) the total number of points earned over all the student instances of Question 3, and (*c*) the number of points associated with the remaining grader instances (not present in the student instances subset) of Question 3.

As an example of the differences between the student mode and grader mode of `grade_o_matic` we consider Question 3, worth 8 total points associated with four grader instances (hence each grader instance is worth 2 points). Let us say you run the student mode of `grade_o_matic`.

You may receive the message

```
Question 3 score:  2.00 of 2.00 (6.00 more points available in grader mode).
```

This message is interpreted as follows: your answer to Question 3 has been checked for the (in this case, single) student input instance and you will receive 2 of 2 points for the student instance since

inputs constitute an allowable instance and if not the code should set an appropriate "error flag." We will not require you to include such error flags in your codes: we will only grade your code based on allowable instances as prescribed in the problem statement.

[2] As already indicated earlier, we strongly suggest that when you have completed the assignment and placed all your scripts and functions in `YOURNAME_ASSIGNMENT_1` and **before** you zip and upload your folder to course website you run `grade_o_matic` with no arguments (from your `YOURNAME_ASSIGNMENT_1` folder) for final confirmation that no last-minute errors have penetrated your defenses.

[3] Some questions, such as questions with multiple choice answers (in student mode) or questions which involve graphics (in student or grader mode), are not auto-gradable.

your code correctly predicts the output for this student input instance. The student mode cannot provide any further information about your total score for Question 3. However, the grader mode will have access to (in this case) three additional input instances each worth 2 points which you will earn if your code correctly predicts the output.

You may also receive the message

```
Question 3 score:  0.00 of 2.00 (6.00 more points available in grader mode).
```

This message is interpreted as follows: your answer to Question 3 has been checked and you do not receive any points for the student instance since your code incorrectly predicts the output for this student instance. (Clearly, this suggests that you should debug your code.) Again, the student mode cannot provide any further information about your total score for Question 3. However, the grader mode will have access to (in this case) three additional input instances each worth 2 points which you will earn if your code correctly predicts the output.

It should be clear that you should not rely exclusively on the student mode ("student instances") `grade_o_matic` assessment to determine if your code is working properly. Successful evaluation for the student instances is a necessary *but not sufficient* condition for correct performance, and in particular your code may succeed for all the student instances yet still fail for a grader instance — such that you will not receive the full points available for the question. You should thus confirm, independently of `grade_o_matic`, the "logic" of your codes, and you should furthermore conduct your own tests for carefully chosen instances of the input parameters designed to flush out flaws either for special cases or for representative generic cases. Note the input parameter domains — allowable input parameter instances — will most often be intervals or regions and hence it will typically not be possible to conduct exhaustive brute force testing. In 2.086 the consequences of a faulty code or poorly prepared input parameters, and in particular insufficient testing, are relatively benign: you might receive a low grade on a question due to a bug not detected by the "student instances" but subsequently caught by the "grader instances"; more optimistically, you might receive unearned credit for a bug caught by neither the "student instances" nor the "grader instances." In contrast, in real life, the failure of a code or a computational method can lead to engineering disasters and indeed loss of life.

Questions

Questions 1–5 are based on the Fibonacci sequence. We recall that the Fibonacci sequence is defined by

$$F_n = \begin{cases} 1, & n = 1; \\ 1, & n = 2; \\ F_{n-1} + F_{n-2}, & n \geq 3, \end{cases}$$

which is strictly increasing for $n > 2$. In actual practice we will consider only a finite number of terms in this sequence. **Note**: in Question 1, Question 2, and Question 3, you should not use any single-index or double-index arrays as these first few questions are intended to exercise your "scalar" skills.

1. (8 points) Write a script which, given a positive integer N, calculates F_N. Our input parameter is N and must correspond in your script to MATLAB variable `Nfib`; the allow-

Cartoon by Sidney Harris removed due to copyright restrictions.

Figure 1: Cartoon by Sidney Harris

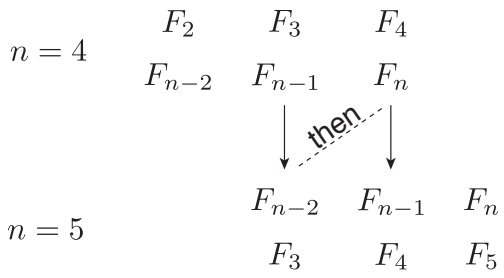

Figure 2: Visualization of the "shuffle" sequence.

able instances, or input parameter domain, is given by $1 \leq N \leq 25$. Our output is F_N and must correspond in your script to MATLAB (scalar) variable `F_Nfib`: your script must assign the result F_N — the N^{th} term in the Fibonacci sequence as defined in the problem statement above — to the MATLAB variable `F_Nfib`.

We ask that you use a `for` loop. Note that at any given time you should only store the three active members of the sequence, which you will re-assign — shift, or "shuffle" — appropriately. Figure 2 may help you visualize this shuffle.

2. (8 points) Write a script which, given a positive integer $F_{\text{upper limit}}$, finds N^* such that $F_{N^*} \leq F_{\text{upper limit}}$ and $F_{N^*+1} > F_{\text{upper limit}}$. Note that from the monotonicity properties of the Fibonacci sequence we may interpret our task more intuitively: N^* is the number of terms in the sequence which are less than or equal to $F_{\text{upper limit}}$. Our input is $F_{\text{upper limit}}$ and must correspond in your script to MATLAB variable `F_upper_limit`; the allowable instances, or input parameter domain, is given by $2 \leq F_{\text{upper limit}} \leq 70,000$. Our output is N^* and must correspond in your script to MATLAB (scalar) variable `N_star`: your script should assign the result N^* — as defined in the problem statement above — to the MATLAB variable `N_star`.

   We ask that you use a `while` loop and a "counter" variable. Figure 3 may help you visualize the process.

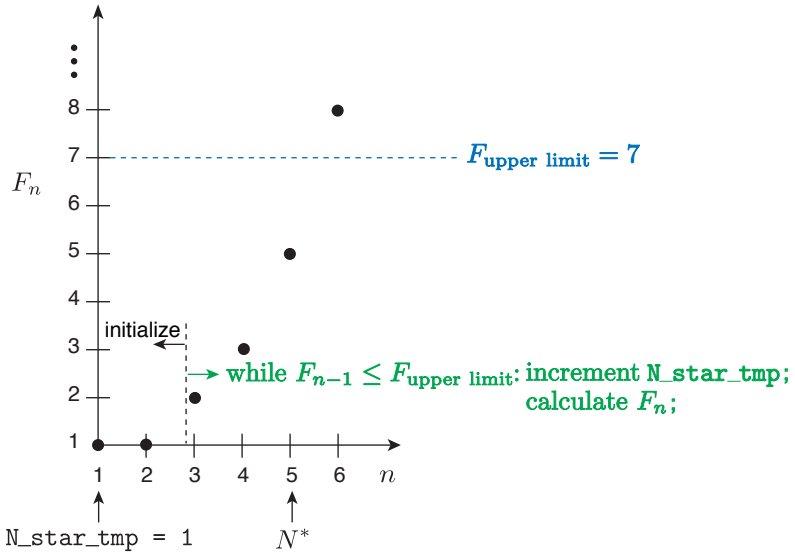

Figure 3: Visualization of a `while` loop and a "counter" variable.

3. (8 points) Write a script which, given any positive integer N, finds the sum S_N of those F_n, $1 \leq n \leq N$, for which the digit in the "ones place" is either 0 or 1: [4]

$$S_N = \sum_{n=1}^{N} \begin{cases} F_n & \text{if digit in the "ones place" of } F_n \text{ is either 0 or 1} \\ 0 & \text{if digit in the "ones place" of } F_n \text{ is neither 0 nor 1 .} \end{cases}$$

Our input is N and must correspond in your script to MATLAB variable `Nfib`; the allowable instances, or input parameter domain, is given by $1 \leq N \leq 25$. Our output is S_N and must correspond in your script to MATLAB (scalar) variable `fibsum`: your script should assign the result S_N — as defined in the problem statement above — to the MATLAB variable `fibsum`.

We ask that you use a `for` statement, a (scalar) `if` statement, and a summation (or "accumulation") variable. Note that to obtain the digit in the ones place of an integer you will find the MATLAB built-in function `mod` very convenient, as suggested in `A1Q3_Template.m`: note that for a positive integer `k`, `mod(k,10)` returns `k - 10*n` where `n` is the largest integer such that `k - 10*n` is non-negative; if `k` is exactly divisible by `10`, then `mod(k,10)` returns `0`.

4. (8 points) Write a script which, given a positive integer N, constructs a single-index (row) array `Ffib` which contains the first N terms of the Fibonacci sequence — $[F_1, F_2, \ldots, F_N]$. For example, for $N = 3$, `Ffib` will be given by $[1, 1, 2]$. Our input is N and must correspond in your script to MATLAB variable `Nfib`; the allowable instances, or input parameter domain, is given by $1 < N \leq 25$. Our output is $[F_1, F_2, \ldots, F_N]$ and must correspond in your script to MATLAB single-index row array `Ffib`: your script should assign the result $[F_1, F_2, \ldots, F_N]$ — the first N terms in the Fibonacci sequence as defined in the problem statement above — to the MATLAB variable `Ffib`; note that `length(Ffib)` will be `Nfib`.

We ask that you use a `for` loop and a single-index array `Ffib` initialized (for the proper length) to all zeros.

[4] The digit in the ones place is of course the digit in the base-10 representation which appears just before the decimal; for the Fibonacci series we consider only integers, and thus the digit in the "ones place" is the last digit.

5. (8 points) Write a script which, given a positive integer $F_{\text{upper limit}}$, constructs a single-index (row) array `Ffib_star` which contains the first N^* terms of the Fibonacci sequence — $[F_1, F_2, \ldots, F_{N^*}]$ — for N^* such that $F_{N^*} \leq F_{\text{upper limit}}$ and $F_{N^*+1} > F_{\text{upper limit}}$. Our input is $F_{\text{upper limit}}$ and must correspond in your script to MATLAB variable `F_upper_limit`; the allowable instances, or input parameter domain, is given by $2 \leq F_{\text{upper limit}} \leq 70,000$. Our output is $[F_1, F_2, \ldots, F_{N^*}]$ and must correspond in your script to MATLAB single-index (row) array `Ffib_star`: your script should assign the result $[F_1, F_2, \ldots, F_{N^*}]$ — as defined in the problem statement above — to `Ffib_star`; note that `length(Ffib_star)` will be N^* and is not known *a priori*.

   Use a `while` loop and a single-index array which is initialized and subsequently "grown" by horizontal concatenation.[5]

6. (8 points) Write a script which, given a real number r and an integer N, calculates the sum of a geometric series with $N+1$ terms,

$$G_N = \sum_{i=0}^{N} r^i = 1 + r + r^2 + r^3 + \cdots + r^N \ .$$

   Note that we may also write the first term in the sum as $1 = r^0$, which shall prove useful subsequently. Our inputs are N and r which must correspond in your script to MATLAB variables `Ngeo` and `rgeo`, respectively; the corresponding allowable instances, or input parameter domains, are $1 \leq N \leq 20$ and $0 < r < 1$, respectively. Our output is G_N and must correspond in your script to MATLAB variable `G_Ngeo`: your script should assign the result G_N — as defined in the problem statement above — to the MATLAB variable `G_Ngeo`.

   Do *not* use a `for` loop or a `while` loop. Instead, we ask that you use the MATLAB built-in function `ones`, the `colon` operator, array "dotted" operators, and the MATLAB built-in function `sum`. In fact, a single line of MATLAB code should suffice, though we suggest you break the nested operations into several steps for improved readability.

7. (10 points) Write a script which, given a vector of points `x_vec` $= [x_1, x_2, \ldots, x_N]$ and a point x , finds the index i^* such that $x_{i^*} \leq x$ and $x_{i^*+1} > x$. Our inputs are `x_vec` and x. The input `x_vec` must correspond in your script to a MATLAB single-index row array `x_vec` of `length` $N \geq 3$; as regards allowable instances, you may assume that the points in `xvec` are distinct and ordered, $x_i < x_{i+1}$, $1 \leq i \leq N-1$, but *you should not assume in this or subsequent questions that the points are equi-distantly spaced*. The input x must correspond in your script to MATLAB (scalar) variable `x`; the allowable instances, or input parameter domain, is $x_1 \leq x < x_N$. (Note that N is not an input, however we do require that `x_vec` is chosen such that $N \geq 3$.) Our output is i^* and must correspond in your script to MATLAB (scalar) variable `i_star`: your script should assign the result i^* — as defined in the problem statement above — to the MATLAB variable `i_star`.

   Do *not* use a `for` loop or a `while` loop. Instead, we ask that you use array relational operators, the MATLAB built-in `find`, and then the MATLAB built-in function `max` (or `length`).[6] You

[5] When we get to double-index arrays we will emphasize the importance of initializing arrays (with `zeros` and later `spalloc`). Initialization ensures that you control the size/shape of the array and also is the most efficient way to allocate memory. On the other hand, concatenation can be very useful in dynamic contexts (in which array size may not be known *a priori*) or in situations in which a large array is most easily expressed in terms of several smaller arrays. But use concatenation sparingly in particular for very large arrays.

[6] Note the algorithm suggested here requires $O(N)$ FLOPs. A better (but more complicated) approach can be developed which requires only $O(\log N)$ FLOPs.

may test your code for various choices of `x_vec` and `x`: possible choices for `x_vec` include (for some given `N`) `(1/(N-1))*[0:N-1]`, `linspace(-3.,1.,N)`, and `sort(rand(1,N))` (this last option will create points which are not equi-distantly spaced).

8. (10 points) In this question we are interested in the interpolant of some univariate function $f(x)$. Write a script which, given a vector of distinct ordered points `x_vec` $= [x_1, x_2, \ldots, x_N]$, a vector of associated function values `f_vec` $= [f(x_1), f(x_2), \ldots, f(x_N)]$, and a point x, finds the piecewise-linear interpolant of f at x, $(\mathcal{I}f)(x)$ as defined in the Interpolation nutshell. Our inputs are `x_vec`, `f_vec`, and x. The inputs `x_vec` and x must correspond in your script to MATLAB variables `x_vec` and `x`, respectively; these inputs are already described in Question 7. The input `f_vec` must correspond in your script to a MATLAB single-index row array `f_vec` of `length` N; we do not specify an input parameter domain for `f_vec`, however we anticipate that the convergence rate of $(\mathcal{I}f)(x)$ to $f(x)$ will depend on the smoothness of the function f which engenders `f_vec`.[7] Our output is $(\mathcal{I}f)(x)$ and must correspond in your script to MATLAB variable `Interp_f_h`: your script should assign the result $(\mathcal{I}f)(x)$ — as defined in the problem statement above — to the MATLAB variable `Interp_f_h`.

   We ask that you build your interpolation script on your "find segment" script of Question 7. You may test your code for various choices of `x_vec` (and `x`) as described in Question 7. To form `f_vec`, choose some appropriate function $f(x)$ and then form `f_vec(i)` $= f(x_i)$, $1 \leq i \leq N$. Note the latter can be facilitated by dotted operators and MATLAB built-in functions: for example, for $f(x) = x^p$ for some given p, you may write `f_vec = x_vec.^p`; for $f(x) = e^x$, you may write `f_vec = exp(x_vec)`.

   A general comment on testing: In problems in which you consider a numerical method which itself represents an approximation it can be difficult to confirm that the implementation is correct: it is possible to conflate bugs with legitimate numerical errors. Hence it is always good to start with a very small case which you can check by hand. Then next proceed to an instance in which the numerical (approximation and discretization) errors are zero, or in any event on the order of machine precision — this test can readily identify logical or programming errors. Then finally proceed to a case in which you can anticipate the convergence rate of the numerical method — this test will flush out more subtle, or higher-order, errors. You can apply this approach to Question 8, Question 10, and Question 11.

9. (6 points) (Driscoll 5.1[8]) Write a script to plot, on a single figure, the functions $\exp(-x)$, $\exp(-(x^2))$, $(\exp(-x))^2$, and $x\exp(-x)$ over the interval $-1 \leq x \leq 1$. Plot each function at 10 equi-spaced points in x (over the interval $-1 \leq x \leq 1$); choose for $\exp(-x)$ a red line and no mark at each point, $\exp(-(x^2))$ a black dotted line with no mark at each point, $(\exp(-x))^2$ a green line with a circle at each point, and $x\exp(-x)$ no line with a blue $\times$ at each point; create a legend for the four curves; label your axes as appropriate; and provide the figure with a title. There are no inputs or outputs for this question.

   Create the set of points in x and the corresponding vectors of function values as described in Question 8.

[7] In this case we clearly see why exhaustive testing is impossible: the dimensionality of the input parameter domain can be very high. More importantly, this example illustrates the distinction between a code which executes the intended steps correctly and a code which executes the intended steps correctly *and also yields a suitably accurate answer to the question posed*; the former relates to implementation while the latter relates to the computational method implemented and the allowed instances (as well as the implementation in some cases).

[8] Derived from *Learning* MATLAB by Tobin Driscoll.

10. (10 points total) In this question we are interested in the first derivative of some univariate function $f(x)$. Write a script which, given a vector of distinct ordered points `x_vec` $= [x_1, x_2, \ldots, x_N]$ and a vector of associated function values `f_vec` $= [f(x_1), f(x_2), \ldots, f(x_N)]$, finds

    (a.) (5 points) `fprime_h_vec`, the forward difference approximation to $f'(x)$ at the set of points x_i, $1 \leq i \leq N-1$—specifically $[f'_h(x_1), f'_h(x_2), \ldots, f'_h(x_{N-1})]$;

    (b.) (5 points) `xneg_fprime_h_vec`, the set of points x_i at which $f'_h(x_i)$ is negative.

    The forward difference approximation $f'_h(x)$ is defined in the Differentiation nutshell. Note that if $f'_h(x_i) < 0$ for all x_i, $1 \leq i \leq N-1$, then `xneg_fprime_h_vec` will be identical to `x_vec`; conversely, if $f'_h(x_i) \geq 0$ for all x_i, $1 \leq i \leq N-1$, then `xneg_fprime_h_vec` will be the "empty matrix." Our inputs are `x_vec` and `f_vec` which must correspond in your script to MATLAB variables `xvec` and `fvec` (already described in Question 8), respectively. Our outputs are `fprime_h_vec` and `xneg_fprime_h_vec` as defined above which must correspond in your script to MATLAB single-index row arrays `fprime_h_vec` and `xneg_fprime_h_vec`, respectively; note that `length(fprime_h_vec)` is $N-1$ but `length(xneg_fprime_h_vec))` is not known *a priori* and will be determined by the particular instance studied.

    We ask that you use dotted arithmetic operators and array relational operators. You may test your code for various choices of `x_vec` and `f_vec` as described in Question 7 and Question 8.

11. (6 points) In this question we are interested in the integral of some univariate function $f(x)$. Write a script which, given a vector of distinct ordered points `x_vec` $= [x_1, x_2, \ldots, x_N], x_1 = 0, x_N = 1$, and a vector of associated function values `f_vec` $= [f(x_1), f(x_2), \ldots, f(x_N)]$, finds the "rectangle, left rule" approximation, I_h, to the integral $I \equiv \int_0^1 f(x)\,dx$. The "rectangle, left rule" approximation is defined in the Integration nutshell. Our inputs are `x_vec` and `f_vec` which must correspond in your script to MATLAB variables `x_vec` and `f_vec`, respectively; the corresponding allowable instances are already described in Question 8, except that here in Question 11 we additionally require $x_1 \equiv 0$ and $x_N \equiv 1$. Our output is I_h and must correspond in your script to MATLAB (scalar) variable `Integral_f_h`: your script should assign the result I_h — as defined in the problem statement above — to the MATLAB variable `Integral_f_h`.

    We ask that you use dotted arithmetic operators. You may test your code for various choices of `x_vec` and `f_vec` as described in Question 7 and Question 8.

12. (6 points total) A surface of revolution (see Figure 4) is described by a function $f(x)$ for $0 \leq x \leq 1$: here x represents the axial coordinate and $f(x)$ represents the radius. We shall suppose that $f(x)$ is strictly greater than zero for all x in our interval $0 \leq x \leq 1$, that $f(x)$ is very smooth, and in particular that $f'(x)$ — the derivative of the radius function with respect to axial coordinate — is always finite. It follows that the surface area of the surface can then
be expressed as

$$A = \int_0^1 2\pi \left(1 + (f'(x))^2 \right)^{1/2} f(x)\,dx \ . \tag{1}$$

    (Note that we consider only the lateral area: we do not include the surface area of the end plates at $x = 0$ and $x = 1$, respectively.)

    We now divide the interval $0 \leq x \leq 1$ into $N-1$ segments, S_i, $1 \leq i \leq N-1$, defined by the N points x_i, $1 \leq i \leq N$: $S_i \equiv [x_i, x_{i+1}]$. Note the points are ordered, $x_i < x_{i+1}$,

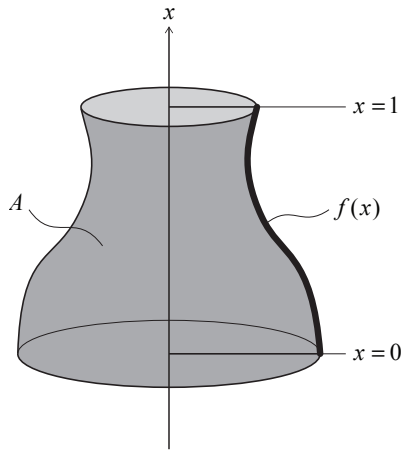

Figure 4: A surface of revolution described by the function $f(x)$ for $0 \leq x \leq 1$.

$1 \leq i \leq N-1$, and also equi-spaced, $x_{i+1} - x_i \equiv h$, $1 \leq i \leq N-1$. It follows that all the segments S_i, $1 \leq i \leq N-1$, are of the same length, h.

We now write

$$A = \sum_{i=1}^{N-1} A^i \ ,$$

where

$$A^i \equiv \int_{x_i}^{x_{i+1}} 2\pi \big(\, 1 + (f'(x))^2 \, \big)^{1/2} f(x) \, dx$$

is the contribution to the surface area from segment S_i for $1 \leq i \leq N-1$. Next, we introduce A_h given by

$$A_h = \sum_{i=1}^{N-1} A_h^i \quad , \tag{2}$$

where

$$A_h^i = 2\pi \left(\, 1 + \left(\frac{f(x_{i+1}) + c_1 f(x_i)}{h} \right)^2 \right)^{1/2} \left(\frac{1}{2} f(x_i) + c_2 f(x_{i+1}) \right) h \ . \tag{3}$$

Here c_1 and c_2 are to be chosen to ensure that the approximation given by equations (2) and (3) converges to A of (1) as N tends to infinity and h tends to zero (for *any* f which satisfies our assumptions above).

(i) (2 points) The constant c_1 should be chosen as

(a) -1

(b) $-\frac{1}{2}$

(c) 0

(d) $\frac{1}{2}$

(e) 1

(*ii*) (2 points) The constant c_2 should be chosen as

(a) -1

(b) $-\frac{1}{2}$

(c) 0

(d) $\frac{1}{2}$

(e) 1

Now assume that the constants c_1 and c_2 have been chosen correctly.

(*iii*) (2 points) The statement "A_h given by equations (2) and (3) will *exactly* equal A given by equation (1) for $f(x) = 1 + 2x$" is

(a) TRUE

(b) FALSE

You may assume here that all floating point calculations are performed exactly without any round-off or other finite-precision effects.

There are no inputs or outputs for this question. Your answer should be a one-line script as indicated in `A1Q12_Template.m`.

2.086 Numerical Computation for Mechanical Engineers
Fall 2014