ENGR 101 Project 4: Space Shuttle Braking Coefficient

Program Design Due: **Tuesday,11/2/2021**
Project Due: **Tuesday, 11/9/2021**

Engineering intersections: Aerospace Engineering/Computer Science

Implementing this project provides an opportunity to practice if/else statements and nested loops.

The program design is worth 10 points, the autograded portion of the final submission is worth 100 points, and the style and comments grade is worth 10 points.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Project Roadmap

This is a big picture view of how to complete this project. Most of the pieces listed here also have a corresponding section later on in this document that goes into more detail.

1. **Read through the project specification (DO THIS FIRST).**
2. **Ensure you understand the update equations for simulating 1D motion.**
   These are in the green box on page 4. If you do not understand them, visit us in office hours.
3. **Ensure you understand the concept of a binary search.**
   This is a critical component of the project.
4. **Complete the flowchart and submit it to Gradescope.**
   Go to office hours to get help if you're stuck on what goes where.
   Submit the flowchart and Honor Pledge to Gradescope (linked on Canvas).
5. **Write the `brakeCoef.cpp` program.**
6. **Verify that your code can reproduce the User Input and Terminal Output described in the Test Cases in the Appendix.**
7. **Check that your program is sufficiently and correctly commented.**
8. **Submit `brakeCoef.cpp` to the Autograder.**

Purpose

The goal of this project is to practice using C++ loops to solve equations numerically. These equations are of a category that cannot be solved analytically.

Background and Project Overview

It's been 30 years since the first settlement was established on Proxima b. The Proximabians want to encourage their burgeoning tourism industry, so they are installing a runway for a new re-usable passenger space shuttle. The aerospace engineers have not finalized the design of the shuttle because the design of the landing gear depends on the length of the runway...and the runway length has not been finalized because the civil engineers don't know the distance the shuttle needs to come to a safe and complete stop!

It is important that the braking system on the shuttle is designed to use as much of the runway as possible (Fig. 1). Stopping too quickly will cause high G-forces on the passengers of the shuttle. Stopping too slowly will cause the shuttle to run out of runway, a potentially fatal mistake.

To help optimize the braking coefficient/runway system, your solution will use a two step approach to determine how hard the brakes need to be applied - that is, solve for the **braking coefficient** - to ensure that the shuttle will stop safely under different landing conditions. The first step of the solution is an algorithm that will *simulate the shuttle's motion* under different initial conditions. In other words, the algorithm determines the **simulated distance** required to stop the shuttle when a particular braking coefficient is used. Once you have composed the algorithm, the second step is to write C++ code that *searches through a range of possible braking coefficients* to determine the **optimal braking coefficient** - the amount of brake application required to stop the shuttle exactly at the end of the runway.

Your Job
In this project, your job is to create a landing scenario program that calculates a passenger shuttle's optimal braking coefficient given different system parameters (initial velocity, air braking coefficient, and braking distance).

Simulating the Shuttle's Motion

To help the engineers finalize their designs for the shuttle and the runway length, you need to simulate the distance necessary for the shuttle to stop when given a number of conditions:

- v_0 - The initial velocity of the shuttle at the start of the runway.
- *b* - The braking coefficient (i.e. how hard do we apply the brakes).
- c_d - The atmospheric drag coefficient. This is an empirically determined value that depends on the shape of the shuttle, the drag created by its flaps, and the thickness of the Proximabian atmosphere. The aerospace engineers will test the shuttle design in a wind tunnel and let you know the proper value (i.e. it's a program input).

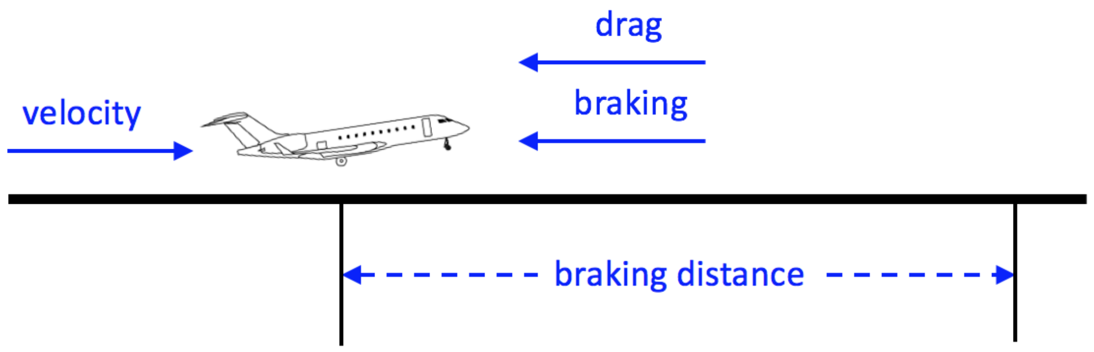


Figure 1. Schematic of the passenger shuttle landing simulation.

Both the braking coefficient and the atmospheric drag contribute to the shuttle's acceleration (here "deceleration", since it's negative). Because air resistance increases the faster the shuttle is moving, the contribution of the drag force depends on the shuttle's current velocity, *v(t)*, which varies with time *t*. The following equation determines *a(t)*, the acceleration of the shuttle at time *t*:

$$a(t) \;=\; -b - c_d\,(v(t))^2$$

Equation 1: The shuttle's acceleration at time *t*.

The **state** of the shuttle as it lands can be described by its current **distance** (from the start of the runway), **velocity**, and **acceleration**. You will use the variables *x*, *v*, and *a* to represent these parameters. These parameters are related as follows: velocity is the change of position over time, and acceleration is the change in velocity over time. Symbolically:

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = a$$

If acceleration *a* is constant, these differential equations may be solved in closed form, directly giving the shuttle's position. However, that's not the present case: the drag force and consequently the acceleration, depends on the current velocity. An analytical solution does not exist, requiring you to use C++ and **numerical methods** to simulate the shuttle's motion approximately.

Engineering Concept - Numerical Methods

In practice, engineers use numerical methods to simulate complex systems approximately when an analytic, closed form solution is not available. For this project, you will use Euler's Method to simulate the motion of the shuttle. The basic approach is to pick a small unit of time (e.g. Δ*t* = 0.00001 seconds) and during that small time, assume constant acceleration and constant velocity (any object essentially moves in a straight line if you observe for a small period of time).

Thus, one may write **update equations** for the shuttle's *new* velocity and *new* position after a single Δ*t* has elapsed, based on *old* acceleration, velocity, and position. The value of the shuttle's previous acceleration (a_{old}) depends on the shuttle's previous state as described in Equation 1 above.

Update Equations for Euler's Method to Simulate 1D Motion

$$a_{old} = -b - c_d v_{old}^2$$

$$v_{new} = v_{old} + a_{old}\Delta t$$

$$x_{new} = x_{old} + v_{old}\Delta t$$

Starting from the shuttle's **initial state** (position *x* = 0 at the start of the runway and velocity v = v_0), one can step forward in time units of Δ*t* calculating a new position and velocity from the old position and velocity. Once those are known, we could save the "new" position and velocity as the "old" position and velocity, then step forward another Δ*t*, and repeat.

Tip

The update equations in the green box above are math equations... *not* "code" equations. Think about how acceleration, velocity, and position will be stored as values in your computer program. Then, think about the order in which each value is updated according to our update equations. Use this information to help create the code that will calculate the acceleration, velocity, and position for each time step.

In this project, use the simulation to determine the total distance required for the shuttle to terminate forward motion. In other words, continue running the simulation (by iteratively updating the state of the shuttle) *until its velocity is no longer positive*. The value of t at this point will be the total time taken for the shuttle to stop. More importantly, the shuttle's position *x* will be the total distance traveled (i.e. the length of the runway required to come to a complete and safe stop).

This procedure allows you to find the **simulated stopping distance** given a set of initial conditions.

Note Use a fixed Δt = 0.00001 for this project.

Other examples of numerical methods are computational fluid dynamics, finite element analysis, and solutions to all manner of ordinary and partial differential equations.

Finding the Optimal Braking Coefficient

The ultimate goal is to determine the **optimal braking coefficient**, $b_{optimal}$, the minimum braking coefficient that will safely stop the shuttle before the runway ends. The minimum value is required since this will ensure the lowest G-forces passengers will endure. Another way to think about it is when $b_{optimal}$ is applied, the shuttle will come to a stop precisely at the end of the runway.

Consider the following: you know that $b_{optimal}$ lies in the range (b_{min}, b_{max}), but what is its value. One strategy would be to look through this space and check every *b* value by calculating the simulated stopping distance for that *b*. This is called a linear search, it would work, but would require checking millions of possible *b* values to determine the exact approximation of $b_{optimal}$ within precise bounds.

Engineering Concept - Binary Search

Instead of a linear search, you will employ a more efficient algorithm called a binary search. In a binary search, you systematically guess at the **midpoint** of the search range and use selection to test whether the desired value is "higher or lower" than the **midpoint**. This divides the search space in half, and may be used iteratively , until you have found the desired value. For example, let's say a friend asks you to guess a secret number between 0 and 20. Here's the binary search strategy:

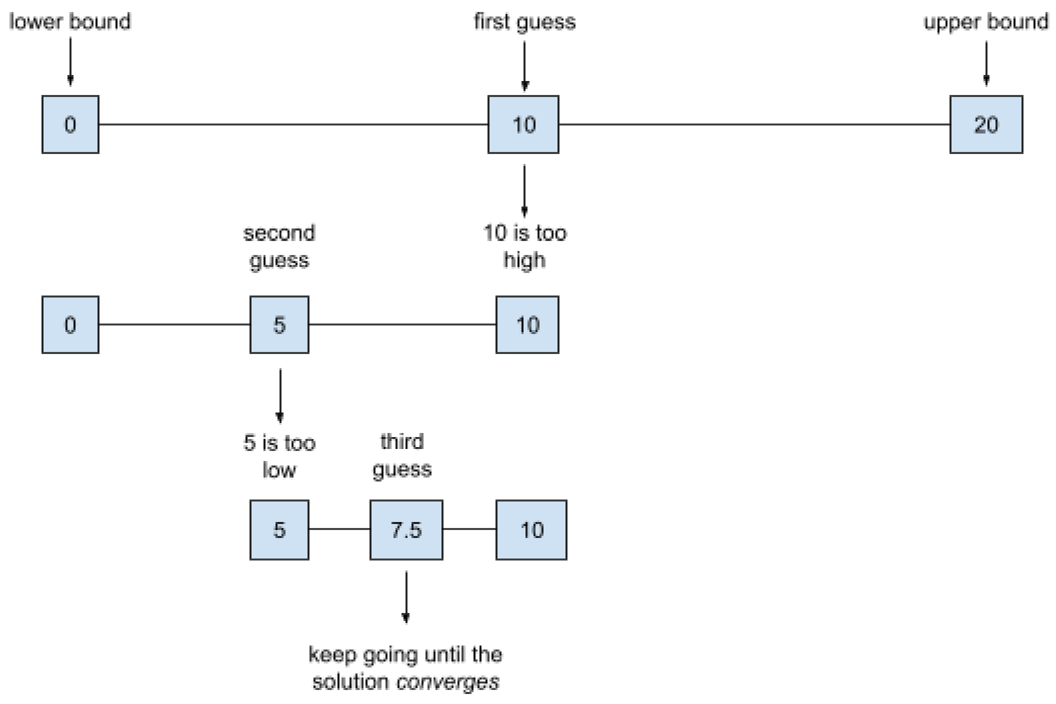


Figure 2. Visualization of how a binary search algorithm works.
Here's more detail. Watch a video of the concept in two dimensional form here.

To define the initial search space, you need lower and upper bounds for all possible values of the braking coefficient *b*. For the lower bound, if the braking coefficient *b* = 0, (your foot is off the brake pedal) the shuttle would travel the maximum distance and stop due to air drag.

For the upper bound, consider the case of no atmospheric air drag. This case would require the greatest braking, since air drag force cannot help slow down the shuttle. If we knew the braking coefficient required for this case, it would be the perfect upper bound for the *b* range. Let's call this b_{nodrag}.

Fortunately, we can solve for b_{nodrag} analytically since the shuttle's acceleration is now constant (with no air resistance). We've done this for you. Based on the initial velocity and runway length:

$$b_{nodrag} = \frac{v_0^2}{2*runwayLength}$$

To start the binary search, set the lower bound $b_{min} = 0$ and the upper bound $b_{max} = b_{nodrag}$.

Now, begin the process of repeatedly narrowing the search range. Pick a candidate braking coefficient using a *b* value midway between b_{min} and b_{max}. Let's call this midpoint b_{mid}. Using this candidate *b* value, compute the **simulated stopping distance** for the shuttle. Comparing this distance to *runwayLength* will tell you whether your guess was too high or too low. That is, you know whether the solution for $b_{optimal}$ falls in the range (b_{min}, b_{mid}) or in the range (b_{mid}, b_{max}). Adjust your lower and upper bounds accordingly and repeat this calculation over and over, each time narrowing the possible *b* range until its size is less than some tolerance. That is, stop when:

$$b_{max} - b_{min} < tolerance$$

When using a binary search, you say that your algorithm has *converged* on a solution when the size of the search space (e.g. the difference between the upper bound and the lower bound) is less than some specified tolerance. Because of the way we've been narrowing in on the solution, the midpoint of this range should be quite close to $b_{optimal}$!

Note
Use a tolerance of 0.000001 in your search for *b* so you get about 6 digits of accuracy in your solution.

Program Task

Using a binary search, determine $b_{optimal}$, the minimum braking coefficient needed such that the simulated braking distance is less than the runway length, *runwayLength*, given the initial landing velocity, v_0, and the shuttle's coefficient of drag, c_d.

To implement the binary search, use the following algorithm:

1. The search range of `b` is (`bMin`, `bMax`). Initially, this is (`0`, `bNoDrag`).
2. Set `bMid` = midpoint of range.
3. Calculate the simulated braking distance (`simDist`) assuming a braking coefficient of `bMid`. Use the numerical methods approach described above.
4. If `simDist` < `runwayLength`, then `bMid` is too high (we stopped earlier than we need - we could let up a bit on the brakes); set new range of `b` to be (`bMin`, `bMid`).
5. If `simDist` >= `runwayLength`, then `bMid` is too low (the shuttle will not stop in the required distance); set new range of `b` to be (`bMid`, `bMax`).
6. Repeat steps 2-5 until the range of `b` is less than the specified tolerance (see note on tolerance in previous section).

For step 3, see the Numerical Methods section -- calculate the distance that the shuttle would travel before its velocity **is no longer positive** by repeatedly calculating, in increments of Δ*t*, new values for the shuttle's acceleration, velocity, and position. When v_{new} is no longer positive, the value for x_{new} is the simulated braking distance, `simDist`.

The initial velocity, v_0, and the atmospheric drag coefficient, c_d, are to be entered by the user at the start of the program. See the User Input Section for more information.

Program Description

In general, you need to write a C++ program that:

1. Prompts the user for and reads in values for v_0, c_d, and *runwayLength* using `cout` and `cin` (see Required Program for more detail)
2. Implements the binary search algorithm described in Program Task section.
3. Outputs the solution for $b_{optimal}$, the minimum braking coefficient needed to stop within the distance, *runwayLength*.

Note: The above list can be considered to be the *start* of a program design for this project. **You will be required to submit a full program design as part of this project.** Refer to the Program Design/Flowchart Section for more details on what you need to submit.

Required Program

There is one required program file for this project: a C++ file that must be named `brakeCoef.cpp`. There is a starter file in the Project 4 folder on the Google Drive.

User Input

User input are values that will be read from the command line using `cin`. Your program must request inputs for v_0, c_d, and *runwayLength*, **in that order**. See the Test Cases for examples. The input prompt (what gets displayed to the user and tells the person what they are supposed to type) **must match the test cases *exactly*, including spaces, colons, etc**.

Terminal Output

Terminal output are values that will be displayed to the user via the terminal using `cout`. Your program **outputs the solution for *b***, the minimum braking coefficient needed to stop within the length of the runway (*runwayLength*). See the Test Cases for examples. The output **must match the test cases *exactly*, including spaces, colons, etc.**

Hints for a Successful Program

Here is some additional information that will help you be successful in this particular project.

Using Compiler Flags to Warn Yourself About Potential Bugs

We've seen how to compile a C++ program into an executable, and we've seen how to use the `-std=c++11` *compiler flag* to make sure that g++ uses the C++ 11 standards when compiling. There are two other compiler flags that you may find useful in this project:

Compiler Flag	Effect when including this flag
`-Wall`	**Checks for common things that may cause wrong behavior at runtime.** Examples include using a variable before it has been initialized and declaring or initializing a variable that never gets used. You can think of `Wall` as "all warnings". Including this flag will tell the compiler to warn you about a number of different things that may cause your program to produce wrong output even though it compiles. Super helpful!
`-pedantic`	**Checks that your code is compliant with C++ standards.** Including this flag will tell the compiler to warn you if you've included code that ~works~ but isn't compliant with C++ standards, which may cause you problems when you submit your code to the Autograder.

The point of compiler flags is to help you identify potential bugs in your code, especially those that occur even though your code compiles and runs. There are many, many other compiler flags that you can use when compiling your code; here is a resource that lists many flags and what they do. When compiling with the `-Wall` (all warnings) and `-pedantic` (for C++ standards) flags, you may come across some common warnings that should be addressed and could help with debugging.

Unused Variables

This project makes use of nested loops, and it can sometimes be a little tricky to figure out what the scope of each variable should be. You might get a warning like this:

```
bash-4.1$ g++ brakeCoef.cpp -std=c++11 -Wall -pedantic -o calcBrakeCoef
warning: unused variable 'accel' [-Wunused-variable]
    double accel = 0;
           ^
1 warning generated.
bash-4.1$
```

This warning is telling you that you created a variable but it doesn't get used anywhere. The bug might be that the variable's scope is too small or possibly too large. If you see this warning, find the line that shows you (in this case, it's the `double accel = 0;` line) and trace your code up and down to figure out why this variable isn't being used.

Uninitialized Values

We know that using uninitialized variables can lead to our programs using "memory junk" instead of the values that we intend for the variable to have. If your program is accidentally using a variable before it is initialized, you will see a warning like this:

```
bash-4.1$ g++ brakeCoef.cpp -std=c++11 -Wall -pedantic -o calcBrakeCoef
brakeCoef.cpp:31:12: warning: variable 'bmin' is used uninitialized whenever
function 'main' is called [-Wsometimes-uninitialized]
    double bmin;
    ~~~~~~~^~~~
brakeCoef.cpp:37:21: note: uninitialized use occurs here
    double bminBad = bmin;
                     ^~~~
brakeCoef.cpp:31:16: note: initialize the variable 'bmin' to silence this
warning
    double bmin;
               ^
                = 0.0
bash-4.1$
```

This warning is telling you to check on the variable listed (in this case, it's the `bmin` variable) to make sure you initialize this variable to SOMETHING before you use it. The compiler gives a suggestion of initializing `bmin` to `0.0` when you declare `bmin`, but you could initialize it to something else, too.

Submission and Grading

This project has two deliverables: the program design and the `brakeCoef.cpp` file. Grades are broken down into 3 categories: the program design, the autograder, and the style and comments.

This project will be graded in three parts. First, a grader will evaluate the *general completeness* of your program design that is submitted to Gradescope and will provide a maximum score of 10 points. Second, the Autograder will be responsible for evaluating your submission and will provide a maximum score of 100 points. Third, a grader will evaluate your submission for style and commenting and will provide a maximum score of 10 points. Thus, the maximum total number of points on this project is 120 points. Each test case on the Autograder is worth the same amount. Some test cases are public (meaning you can see your output compared to the solution as well as the inputs used to test your submission) while many are private (you are simply told pass/fail and the reason). The breakdown of points for each category is described in the next sections.

Program Design/Flowchart

Maximum Score: 10 points

This project implements a complex algorithm. Therefore, a flowchart is a required deliverable. Logic errors are easier to find on paper, in a flowchart, or in other types of program design. Flowcharts also help immensely when describing your program to someone; we will ask to see this program design document/flowchart if you come in to office hours, so keep it handy.

A **starter flowchart** is available in the Project 4 Google Drive folder. We have included the necessary shapes along with a few of the steps to guide you. You need to complete the flowchart by writing in the missing steps. Remember that a flowchart describes the program's control flow; the shapes describe the steps to be implemented, not the actual code itself. For example, if you need to check whether `x>2`, you would write "check that `x>2`" in the correct shape but you wouldn't write "`if (x>2) {`". If you have questions or want to get feedback on your flowchart, come to office hours!

To complete the flowchart, save a copy of the starter flowchart. Then, write in the missing steps. Make sure to complete the Name(s), Uniqname(s), Lab Section Number, and Date section. **If you are working with a partner, complete the flowchart together and put both partners' names, uniqnames, and lab section numbers on the flowchart.**

Submitting your assignment to Gradescope

This Gradescope assignment has two questions: the flowchart and the Honor Pledge. Create a separate document that states the Honor Pledge and includes your signature below the Honor Pledge (if your signature does not look like your name, also print your name).

If working alone:

I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code on this assignment.

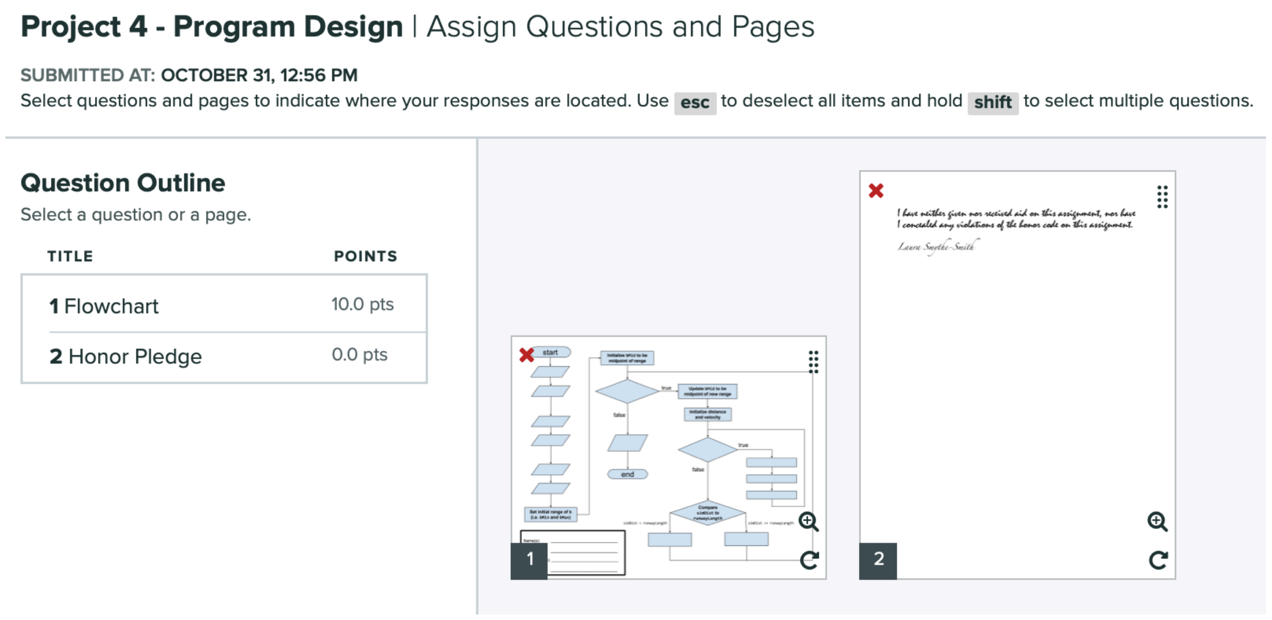
If working with a partner:

We have neither given nor received unauthorized aid on this assignment, nor have we concealed any violations of the Honor Code on this assignment.

The Honor Pledge and your signature must be *written*, not typed; if you are working in a partnership, both partners must sign the Honor Pledge. To create this written document, you have two options. Option 1: write the Honor Pledge and sign it on a piece of paper and then scan it to create an electronic copy; we recommend the Scannable app to get a high quality scan (low quality scans will not be accepted). Option 2: on a tablet or touchscreen, write the Honor Pledge, sign it, and save the file electronically. Please come to Office Hours if you need help with either of these options.

Create a document that contains both the Project 4 flowchart and a signed copy of the Honor Pledge. Save this document as a single `.pdf` (the name of this file does not matter).

When you upload your assignment, you will see something similar to what is shown below:



Project 4 - Program Design | Assign Questions and Pages

SUBMITTED AT: **OCTOBER 31, 12:56 PM**

Select questions and pages to indicate where your responses are located. Use **esc** to deselect all items and hold **shift** to select multiple questions.

Question Outline

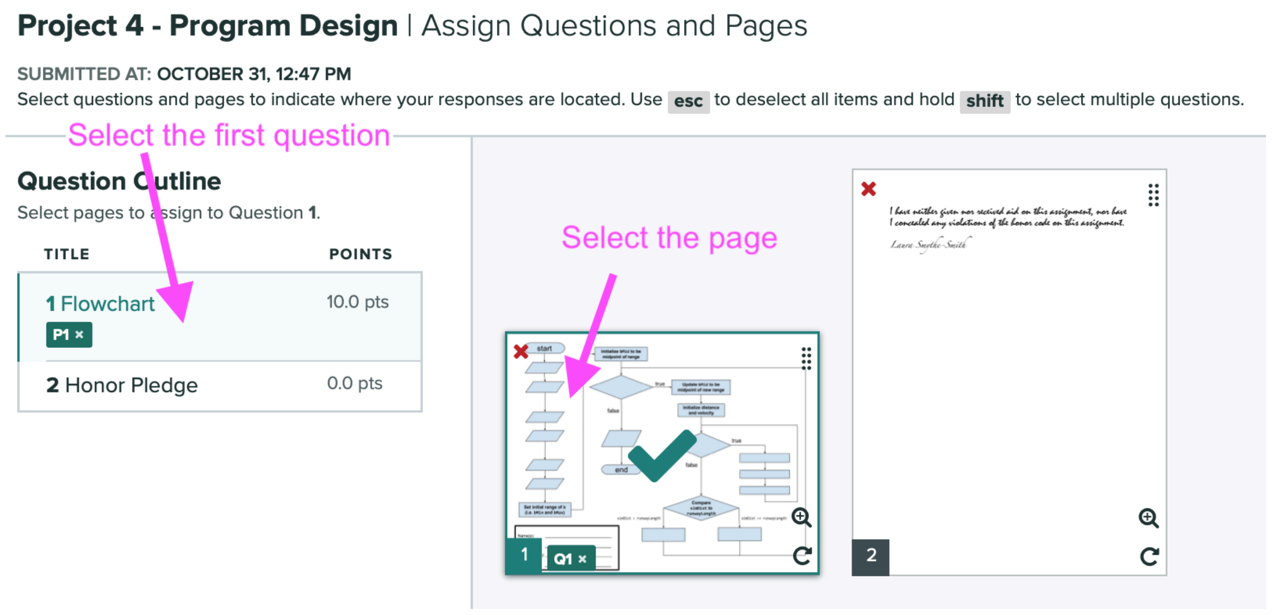
Select a question or a page.

TITLE	POINTS
1 Flowchart	10.0 pts
2 Honor Pledge	0.0 pts

I have neither given nor received aid on this assignment, nor have I concealed any violations of the honor code on this assignment.

Laura Smythe-Smith

You now need to assign which pages go with which question:



Project 4 - Program Design | Assign Questions and Pages

SUBMITTED AT: OCTOBER 31, 12:47 PM

Select questions and pages to indicate where your responses are located. Use esc to deselect all items and hold shift to select multiple questions.

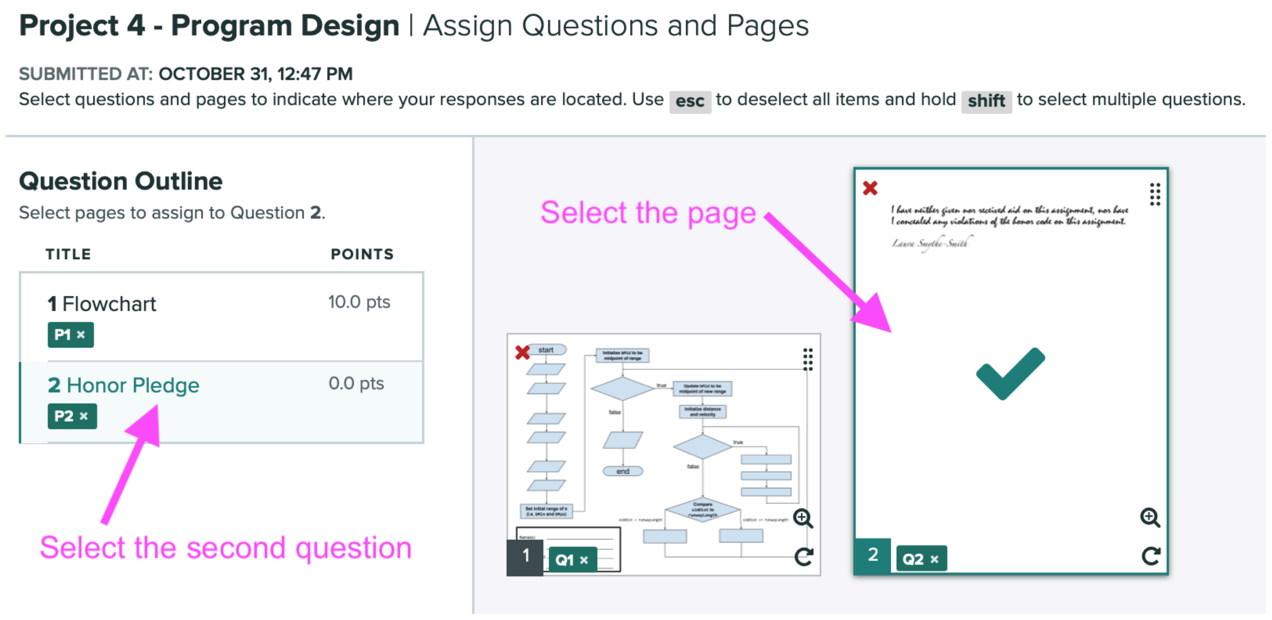
Question Outline

Select pages to assign to Question **1**.

TITLE	POINTS
1 Flowchart P1 ×	10.0 pts
2 Honor Pledge	0.0 pts

I have neither given nor received aid on this assignment, nor have I concealed any violations of the honor code on this assignment.

Laura Smythe-Smith



Project 4 - Program Design | Assign Questions and Pages

SUBMITTED AT: OCTOBER 31, 12:47 PM

Select questions and pages to indicate where your responses are located. Use esc to deselect all items and hold shift to select multiple questions.

Question Outline

Select pages to assign to Question **2**.

TITLE	POINTS
1 Flowchart P1 ×	10.0 pts
2 Honor Pledge P2 ×	0.0 pts

I have neither given nor received aid on this assignment, nor have I concealed any violations of the honor code on this assignment.

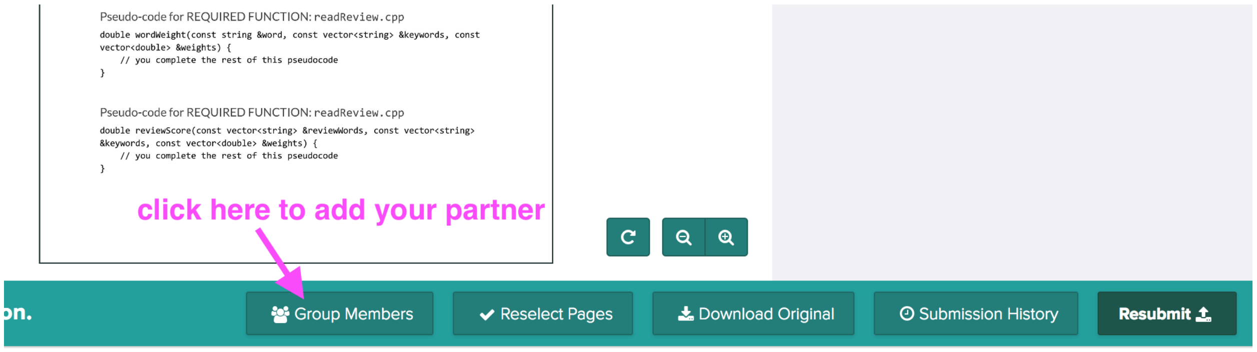
Laura Smythe-Smith

After you have assigned the correct pages to the correct questions, click the "Submit" button in the bottom right corner of the page.

Note: The program design is worth 9 points, and the signed Honor Pledge is worth 1 point (we haven't updated these screenshots yet).

Registering your partnership on Gradescope

If you are working with a partner, use "group submission" on Gradescope. Have one member of the partnership submit the flowchart and honor pledge as described above. After you submit the `.pdf` file, click "Group Members" to add your partner, as shown below:



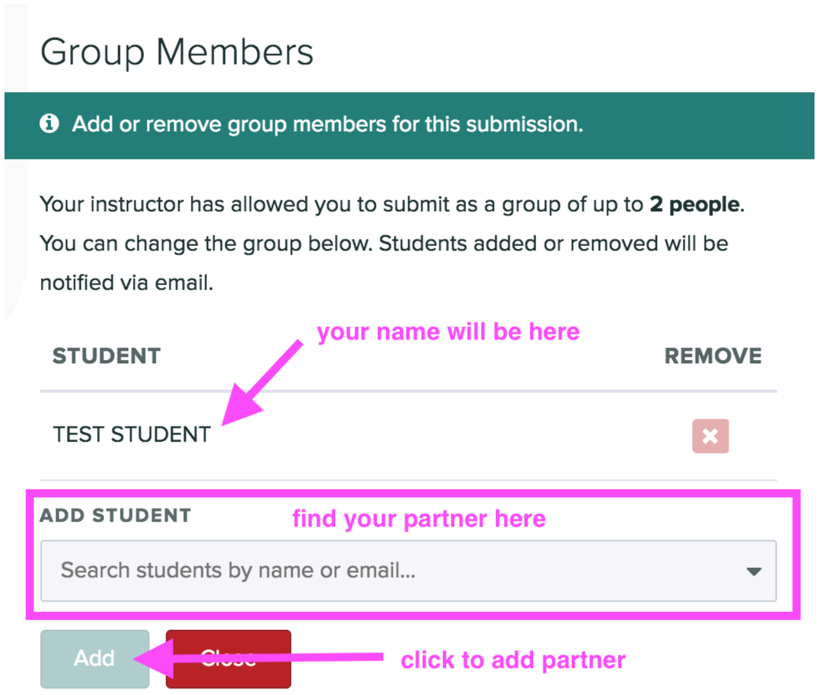
Pseudo-code for REQUIRED FUNCTION: `readReview.cpp`

```
double wordWeight(const string &word, const vector<string> &keywords, const
vector<double> &weights) {
    // you complete the rest of this pseudocode
}
```

Pseudo-code for REQUIRED FUNCTION: `readReview.cpp`

```
double reviewScore(const vector<string> &reviewWords, const vector<string>
&keywords, const vector<double> &weights) {
    // you complete the rest of this pseudocode
}
```

This will pop up a little box where you can search for your partner's name and then add them:



Group Members

Add or remove group members for this submission.

Your instructor has allowed you to submit as a group of up to **2 people**. You can change the group below. Students added or removed will be notified via email.

STUDENT	REMOVE
TEST STUDENT	✖

ADD STUDENT

Search students by name or email...

Your partner will receive an email verifying that they have been added to your group. Make sure your partner gets this email!

Autograder

Maximum Score: 100 points

Submit your `brakeCoef.cpp` file to the Autograder (autograder.io) for grading. The autograder provides you with a final score (out of 100 points) as well as information on ***a subset of its test cases***. We provide this autograder to give you a sense of your current progress on the project.

Submitting files to the Autograder

The Autograder will look for one file: `brakeCoef.cpp`.

Note: You don't *have* to wait until you have all the files ready before submitting to the Autograder. You can submit a subset of files and get feedback on the test cases related to those files. However, **to receive full credit, you need to submit all files and pass all test cases within a single submission.**

Project scoring and feedback on the Autograder

You are limited to 5 submissions on the Autograder per day. After the 5th submission, you are still able to submit, but all feedback will be hidden other than confirmation that your code was submitted. The autograder will only report a subset of the tests it runs. It is up to you to develop tests to find scenarios where your code might not produce the correct results.

You will receive a score for the autograded portion equal to the score of your best submission. Your latest submission with the best score will be the code that is style graded. For example, let's assume that I turned in the following:

Submission #	1	2	3	4
Score	50	100	100	50

Then my grade for the autograded portion would be 100 points (the best score of all submissions) and Submission #3 would be style graded since it is the latest submission with the best score.

Please refer to the syllabus for more information regarding partner groups and for general information about the Autograder.

Style and Commenting

Maximum Score: 10 points

2 pts - Each submitted file has Name, Partner Uniqname (or "none"), Lab Section Number, and Date Submitted included in a comment at the top
2 pts - Comments are used appropriately to describe your code (e.g. major steps are explained)
2 pts - Indenting and whitespace are appropriate (including functions are properly formatted)
2 pts - Variables are named descriptively
2 pts - Other factors (variable names aren't all caps, etc...)

Appendix: Test Cases

Here are four examples of running a sample solution with various inputs. For example, the first case represents a shuttle landing at 60 m/s with a coefficient of drag of 0.0005 / m and needing to stop in 2000 m.

```
bash-4.1$ g++ brakeCoef.cpp -std=c++11 -Wall -pedantic -o calcBrakeCoef
bash-4.1$ ./calcBrakeCoef
v0 (m/s): 60
c_d (1/m): .0005
runwayLength (m): 2000
b_optimal = 0.281732
bash-4.1$ ./calcBrakeCoef
v0 (m/s): 60
c_d (1/m): .0005
runwayLength (m): 1000
b_optimal = 1.04756
bash-4.1$ ./calcBrakeCoef
v0 (m/s): 80
c_d (1/m): .0005
runwayLength (m): 2000
b_optimal = 0.500857
bash-4.1$ ./calcBrakeCoef
v0 (m/s): 80
c_d (1/m): .0005
runwayLength (m): 1000
b_optimal = 1.86233
```

Note

Your program must match *exactly* all prompts and all output text as shown in the test cases above (not including the "bash-4.1$" part) AND your calculated values for *b* **must be within 0.01%** of the values shown in the test cases to receive full credit from the Autograder.