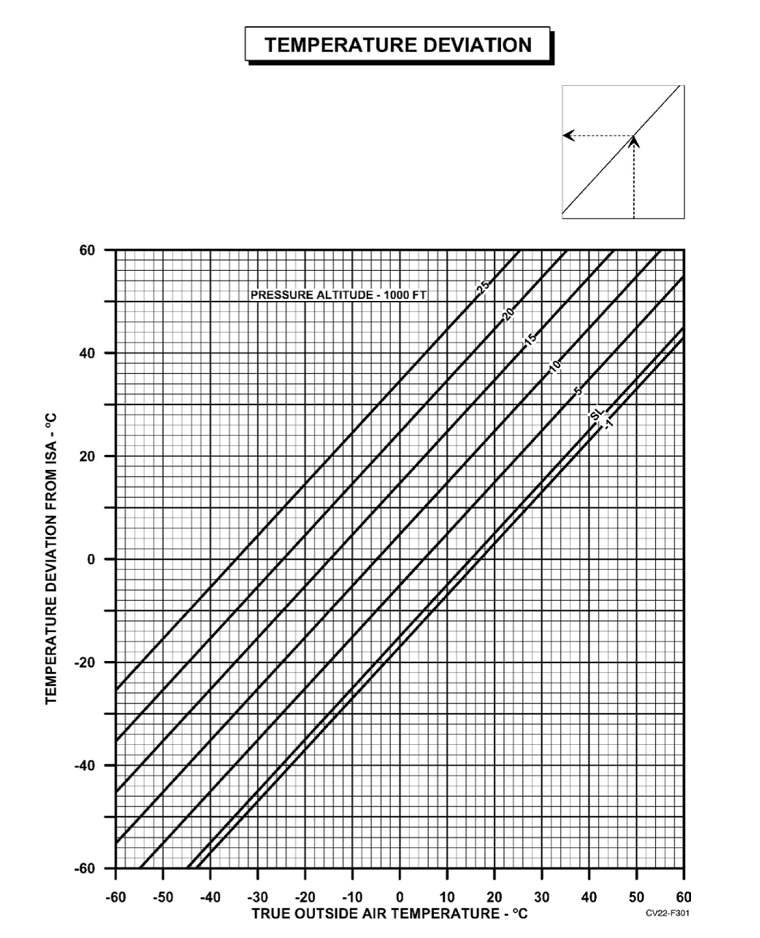
**Chart Studio**  
Dan Teel

Chart Studio is a tool that allows the virtual modeling of aircraft performance charts and the processes used with them. It’s kind of like a visual programming language.

It allows the modeling of 4 ‘types’ of charts, they are

* Linear charts, the most common, come into the chart on the x or y axis until you reach the desired line, and then the output value is the other axis value of that point.
* Trend charts, the second most common, these charts are ran by coming straight in on one axis, meeting a pair of lines, and following the ‘trend’ of the lines until a certain point, and the output value is where it currently lies on the input axis.
* Clamp charts, these are usually compounded on top of Linear charts to restrict the output value to between a minimum and maximum line.
* Polygon charts, rarely used, but the output value is the value of the polygon in which an X, Y pair lies.

Linear Chart

A linear chart is one where you start on one axis at a certain point, trace up or horizontally, and then trace out on the other axis.

As an example, the figure to the right, if you were trying to find the temperature deviation from ISA when the true outside air temperature is 10 degrees and at 5000ft pressure altitude, start at point x: 10, y: -60, trace up to the 5000ft line, and then read out on the left. The result ends up being about 5.

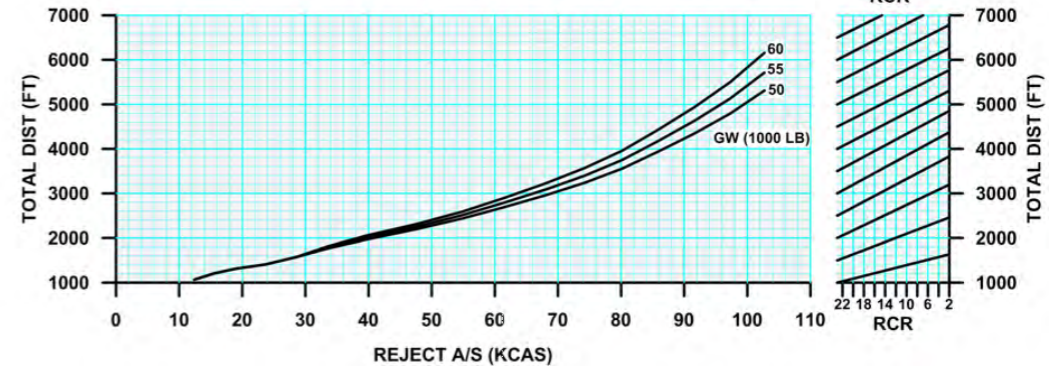
This chart takes its input value on its X axis. If we came up from the left and then down, it would takes it input on the Y axis.

Trend Chart

A trend chart starts and finishes on the same axis, but over the course of its trace, it will follow 2 lines it lies between to alter its value.

In the figure below, the small chart on the right is a trend chart. It takes its input on the Y axis, starts following the trend lines at RCR 23. It will follow to the relevant trend lines until it stops at whatever RCR the user has inputted.

Below is an example of it being ran. Start at 4000ft and RCR 23, follow the closest trend lines all the way to RCR 2, and then that’s the result. 4000ft with an RCR of 2 turns into 5700ft.



Besides just charts, the program also allows you to model Tables and program scripts.

* Tables will interpolate between X amount of values based on a separate value. For instance, if you have 3 charts, one for sea level, one for 5000ft, and one for 10000ft, you can include all three of those charts output values along with their reference values (0, 5000, or 10000) to interpolate a result for an altitude of anywhere in-between 0ft and 10000ft.
* Scripts allow you to programmatically include any sort of logic to produce a value. Two common uses are to apply Altimeter Setting corrections for Pressure Altitude, and another is to accomplish a binary search to run charts backwards, that aren’t meant to be run backwards. But it can be used for almost anything, as long as you can program it.

Lastly, the program has Inputs and Constants.

* Input objects allow a user to enter values into the program for reference by all the other Chart/Table/Script objects, these commonly included Temperature, Elevation, Gross Weight, Runway Condition Rating (RCR), and others. Inputs utilize same scripting language as Scripts to turn the inputted values (strings) into number values or null if the input is invalid.
* Constants are simply an object that holds an immutable number value. A common example is the number 23, which is the highest RCR.

**Chart Scripting Language** (not a very good name I know =\)

CSL is very loosely based on C and is statically typed.

Lines are terminated with a semi colon (;), e.g.

string name = “Dan Teel”;  
exit name+” is my name”;

Comments only support one style, ‘//’. Anything after // until a new line is ignored by the interpreter.

It can do functions and nested functions within. It can handle recursion, but does not allow for closures. You cannot store a reference to a function, it can only be invoked by its functionName( <arguments here> ).

double squaredTwice( double value ) {

double squared( double value ) {

return value^2;

}

return squared(squared(value));

}

In the previous example, the only named thing in the global scope is squaredTwice. Inside of the squared function inside squaredTwice, the value parameter makes the value parameter of squaredTwice inaccessible. If they were named differently, you would be able to access both.

One downside to this language is that it does not currently allow forward declarations. If a function lives below where you want to use it, you can’t use it.  
  
**Types**

There are no implicit type conversion except inside the condition expression of if/while/foor/loops.

This is not allowed

bool heHasACar =false;  
bool sheHasACar=true;  
double numberOfCars=heHasACar+sheHasACar; //Expected doubles, but got booleans  
string howManyCars=”They got this many cars: “+numberOfCars;//Expected string to concatenate, but it got a double.

To convert to any one type, simply use double(<expression>), bool(<expression>), or string(<expression>)

This is allowed

bool heHasACar=false;  
bool sheHasACar=true;  
double numberOfCars=double(heHasACar)+double(sheHasAcar);  
string howManyCars = “They got this many cars: “+string(numberOfCars);

In addition to each type holdings its type, they can also hold null, which enables optional like programming techniques. Beware trying to operate on null values though (null+5 will throw an exception and fail the script, as will min(null, 5) ).  
  
Boolean values can hold the named constant ‘true’, ‘false’, or ‘null’.

Double types can hold a number, e.g. (-5, 123.456, -.3, 8), or ‘null’.

Strings can hold values by way of string literals. Two styles of string literals, one using the “ and one using the ‘. E.g. “I’m a string!”, ‘Im also a string!’. String escapes are not currently provided. Strings as well can also hold ‘null’.

**Operators**

The operator order of precedence list in descending order:

parentheses **(** <any> **)**unary minus **-** <double>  
unary not **!** <bool>is null unary operator **?** <any>  
  
exponentiation <double> **^** <double>  
  
multiply <double> **\*** <double>  
divide <double> **/** <double>  
remainder <double> **%** <double>  
  
add <double> **+** <double>  
subtract<double>  **-** <double>  
  
equals <any> **==** <any>  
not equals <any> **!=** <any>  
greater than <any> **>** <any>  
greater than or equals <any> **>=** <any>  
less than <any> **<** <any>  
less than or equals <any> **<=** <any>  
and <bool> **&&** <bool>or <bool> **||** <bool>

ternary <bool> **?** <any> **:** <any>

**If – Else**

Much like C, an if statement will execute code in its block if the expression it evaluates is true. If you do not use braces, only the immediate statement after the expression will be executed (if true). Below are examples  
  
 if (boolVariable) a=a+1;  
  
 if (boolVariable == true){  
 a=a+1;  
 b=b-1;  
 } else {  
 a=a-1;  
 b=b+1;  
 }

**For loop**

Much like C, but different, the for loop looks like   
  
 for (<assignment or declare>; <bool expression> ; <assignment>){  
 <code to execute here>  
 }

The braces are optional if you just want to execute one statement in the loop. Use break to immediately exit the loop.  
  
 Examples:

for (double i=0; i<10; i=i+1){  
 notice( string( i ) );  
 }

for (double i=0; i<10; i=i+1) print( string( i ) );  
  
 for (;;){  
 i=i+1;  
 //Do something cool here  
 if (i==9) break;  
 }

**While loop**

The while loop is pretty standard, as long as the condition is true, it will keep looping. Again, the braces are optional if you only want to loop one line. Use break to immediately exit the loop.Example  
 while (true){  
 //Never ending loop, don’t do this.  
 }

**Loop-While loop**

This loop is almost the same as the while loop, except that it will run the loop body at least once, and then it will start evaluating the condition to determine if it will keep looping. Use break to immediately exit the loop.

Example  
   
 loop {  
 //Do something you want ran at least once in here  
 } while (a>10)

**Exit statement**

When a script is ran, its under the assumption that it should return a value. In ChartStudio, all of the scripts (along with Input scrips) expect a double value or null to be returned. To immediately exit script execution and pass back a value, use the exit statement. If you don’t explicitly exit with a value, the interpreter returns null as its exit value.  
  
 Examples  
   
 exit 100; //Returns a double

exit; //Returns null

**Function call statement**  
  
 Every function declared is supposed to return a double, bool or string. That doesn’t mean you have to capture that value though and they can be treated like subroutines and invoked outside of an assignment statement.  
  
 Example  
   
 bool exitWithValuesAdded( double value1, double value2 ){  
 exit value1 + value2;  
 }

exitWithValue(); //Calling the function and not capturing the return value.

**Declaring variables**

To declare a variable, simply put the type of variable you want, and then the name of it. If you want to declare multiple variables, separate the names by a comma. You can also immediately assign a value during declaration by putting an = and an expression.   
  
 If you do not assign a value, the variable will default to null.  
  
 Variables are only accessible in the scope its declared, or children scopes.  
  
 Valid variable names can start with an alphabetic character or an underscore. After the initial character, the rest of the characters can be alphabetic, numeric, or underscore.  
  
 Examples

string a;  
 bool b = true;  
 double c, d=10, e;

**Declaring functions**

Much like C, we declare our functions by first writing the return type, the function name, and its argument list with types. Function name rules are the same as variables.

Examples

string hisName(string name){  
 return “His name is “+name;  
 }  
  
 double sumOfValues( double value1, double value2, bool floorTheValue){  
 return floorTheValue ? floor(value1+value2) : value1+value2;  
 }  
  
 bool not( bool valueToNot){  
 return !valueToNot;  
 }

**Imported variables**

The interpreter can have variables passed to it. After script execution, these variables can be used by the parent program. This allows the script to return data without using exit.  
  
 For Script objects, all Chart object calculation results are accessible through its name, but they are constant and cannot be written too, only read. The value in an exit statement sets the Script objects value.  
  
 For Inputs, the string value of the input is imported in a read-only variable called ‘input’. The exit statement sets the value of the input.

To highlight that the input field has invalid data, there is also a variable imported called ‘invalid’. Setting this to true will allow the parent program highlight to the user that they put in bad data.

**Imported functions**

Much like imported variables, functions can be passed into the script. Two functions imported to Script objects and Inputs are bool alert( string ) and bool notice( string ).

Calling alert will tell the parent program to display a message with the string value passed, with terrible connotations.

Calling notice will tell the parent program to display a message with the string value passed, but in a friendly way.