**Differences between C and Python Implementations of SWHS**

Note: This does not include very basic syntax differences between the two languages.

|  |  |  |
| --- | --- | --- |
| **Module** | **C** | **Python** |
| **Input Parameters** | * Stored in a “struct” defined in a header file | * Stored in a dictionary defined in a Parameters class |
| **Input Format** | * Pi is defined outside of function * Loop through input file with while(1) loop, number of lines is hardcoded to determine when to break loop. (Originally used while(!feof) but that was not portable to Mac OS X). * Includes math.h to make use of pow function when calculating parameters. | * Pi comes from importing math.py * Loop through input file with for loop, which detects end of file and stops automatically. * Uses Python’s \*\* operator to calculate powers when calculating parameters. |
| **Input Verification** | * Error messages are printed using the standard printf function (C has no error raising function) * An integer error code is defined for each error to differentiate between them (makes it easy to determine if error was raised, just check if integer > 0) * Warning messages are printed using the standard printf function (C has no warning raising function) * Functions must return something since no errors/warnings are actually raised | * Python’s raise function is used to actually raise errors * A message is supplied to the raise function to differentiate between each error. * Imports warnings module to make use of warn function for raising warnings. * Functions do not return anything since errors/warnings are actually raised and can be detected. |
| **Temperature** | * Is a function required by ODE solver, defined in main.c because it needs to use the parameters structure * Defines expressions for the RHS of each ODE and assigns them to specific elements of matrices as required by the ODE solver (Uses actual values rather than strings) * Returns 0, as required by the ODE solver (function is tied to the solver and therefore follows the required template) | * Is a function required by ODE solver, defined in separate file because parameters can be referred to by strings which are tied to values using a dictionary in main.py * Simply defines strings for the RHS of each ODE, and ties them to their corresponding variable (also referred to by a string) in a dictionary. The ODE solver handles the reading of these strings * Returns the dictionary to be assigned to the ODE solver in the main.py module |
| **Event** | * Is a function required by the ODE solver, defined in main.c because it needs to use the parameters structure. * Defines expression that will equal 0 to trigger event, using actual values rather than strings. * Returns 0, as required by the ODE solver (function is tied to the solver and therefore follows the required template) | * Is a function required by ODE solver, defined in separate file because parameters can be referred to by strings which are tied to values using a dictionary in main.py * Defines strings of expressions that will equal 0 to trigger event * Makes the event using functions from the PyDSTool module, and returns it to be assigned to the ODE solver in main.py |
| **ODE Solver** | * CVODE * Defines new types (realtype, n\_vector) that must be used for all values. Any values that are not these types must be contained in the RCONST() macro. * ODE solver is initialized through a series of functions that take ODE parameters, such as the temperature and event functions, the initial conditions, tolerances, time step, and initial time as arguments. * Time range is not passed as argument (only initial time and time step). * Function defining Jacobian passed as an argument * The function that computes the ODE solution only does so for a single time step. Thus, the call to this function occurs inside a while loop. This is why a final time is not required; the solver can be stopped by using an appropriate condition for the while loop. This also means that if the final time is hit in the first or second ODE set, the remaining sets will be skipped as the condition for the while loop will not be met. * Events are signalled by way of a flag that is checked in each loop. * Solution values are automatically assigned to the variables that were fed to the ODE solver as arguments. | * PyDSTool * No new types defined (as expected, as Python does not require type declaration) * Parameters, temperature functions, initial conditions, tolerances, and initial time step are defined in dictionaries that tie actual values to strings. These are then individually assigned to an arguments structure for the ODE solver. * Time range is defined and added to the argument structure. * No Jacobian required * ODE solver is initialized in a single line of code that takes the arguments structure as an argument. * Solution to ODEs is computed outside of a loop. Thus, subsequent ODE sets are contained in if statements that depend on whether or not the terminal event was triggered. * Solution and event values must be accessed using the sample(), and getEvent functions, respectively. These values will be contained in dictionaries with the variable strings as keys. The lists are then extracted from the dictionary and assigned to separate variables for each result (i.e. time, water temp, PCM temp, water energy, PCM energy) |
| **Energy** | * Calculates energy for single time point, called in a loop in main.c (I had trouble returning a list from a function in C, so I did it this way instead) | * Loop within the function, calculates energy for all time points and stores in list |
| **Output Verification** | * Takes size of results (i.e. length of output lists) as an argument * Uses for loop to calculate sum of elements in list * Prints warnings using standard printf function * Has an integer to count the number of warnings called for testing purposes | * Does not need size of results as an argument * Uses sum function to calculate sum of elements in list * Imports warnings module and uses warn function to raise warnings. * Does not require warning counter because testing can detect raised warnings. |
| **Output Format** | * Size of results is an argument * No max\_width calculated * String formatting used with fprintf in for loop to write data to file. | * Size of results is not an argument * Calculates max\_width and uses ljust to display results in neatly aligned columns * No string formatting, each result written to file using write function, ljust, and adding the results into a single string for each time point (in for loop) |
| **Plotting Module** | * Gnuplot * C code pipes gnuplot commands | * Pyplot from matplotlib * No need to pipe commands since Pyplot is a python module. |
| **Testing** | * Unity framework * Requires test runners for each group of tests, and one more for the entire suite * No event tests (would not work given CVODE’s strict format for event functions) * Output files scanned using fscanf | * Unittest framework * Only one test runner to run every test * Event tests * Output files scanned using readlines() and then split() |

**General Comments**

* C code uses header files for each module, no equivalent in Python.
* Prototypes in C, not in Python.
* C requires “main” function, Python does not.
* Command line arguments are stored in argv for both implementations, but in C argv is declared as an argument for “main”, while in Python the sys module must be imported to access argv.
* Using for loops in C code requires some sort of iterator variable, but not in Python as Python automatically iterates through lists until their end. If Python needs to iterate through multiple lists at the same time, the zip function is used to create tuples out of the lists. Each item of the tuple can then be assigned to a variable in the for loop.
* Python can use the colon operator on indexes to access only certain parts of a list, C cannot.
* C requires type declaration, Python does not.
* C requires memory allocation, Python does not.
* Python can use negative indexes to access the last item in a list, C cannot.
* As a result of the above two points, a variable holding the size of the lists is carried around to many functions in the C implementation (output, verify\_output) but is not needed in the Python implementation.
* List comprehensions can be used in Python where loops are required in C (I used them to add lists element-wise when calculation total energy)
* Since we don’t know how many events will be triggered before running the program, and since C requires memory allocation beforehand, in the C implementation we have to transfer the results from the original lists to new, correctly sized lists after everything has been calculated. This is not necessary in the Python implementation.
* Calling functions from other modules in Python requires the module name and then the function name separated by a dot, while in C it only requires the function name.
* Python can use + operator on strings where C must use strcat() function.
* Python imports sys to add to path where necessary, whereas C uses the include flag when the compiler is called.
* The GCC compiler in general requires more flags than Python (which requires none). Source files, include directories, library directories and library links are all required for gcc.
* Some things are defined outside of functions in C using ‘#define’, such as pi and some structures needed by the ODE solver. There is no equivalent to this in Python, considering the entire control module is not a function.