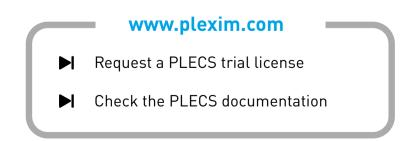# PLECS

Tutorial

Introduction to the C-Script Block

Implementation of a digital and analog PI controller

Tutorial Version 1.0

1 Introduction

The C-Script block is a versatile tool in the PLECS component library that can be used for implementing custom controllers and components. The advanced capabilities of the C programming language combined with the flexible sample time settings in the C-Script block allow almost any custom component model, from a simple mathematical function to a complex state machine, to be implemented. The C-Script block can also simplify the workflow when writing C code for DSP controllers since the code can be reused for the DSP. The key skills that you will learn in this exercise are:

- Understand how the C-Script block interfaces with the simulation engine through function calls.
- Understand the different time settings available in the C-Script block.
- Use the C-Script block for implementing a mathematical function.
- Use the C-Script block for implementing a discrete and continuous PI controller.

Before you begin Ensure the file `buck_converter.plecs` is located in your working directory. You should also have the reference files that you can compare with your own models at each stage of the exercise.

2 Function Call Interface

The C-Script block interfaces with the simulation engine using a number of predefined function calls. These function calls are depicted in Fig. 1. Each function call corresponds to a code window in the C-Script editor, which is accessed from a pull-down menu. The most commonly used code windows are described below, and for a complete description, one can refer to the PLECS User Manual or the block's documentation, which is accessed by clicking the **Help** button.

code declarations() In addition to the function windows, a **Code declarations** window is provided for defining global variables, macros and helper functions to be used in the C-Script block functions. The code declarations code is essentially a global header file. All variables and functions defined within this window are globally visible to all functions within the C-Script block.

start() The **Start function code** window is for initializing the simulation. Internal state variables, for example, should be initialized here.

output() The **Output function code** window is designed to contain the functional code for the C-Script block. A call is made to this function at least once during each simulation time step. For this reason, any internal states or persistent variables should be updated in the update function.

update() If a model contains discrete internal states, a call is made to the code in the **Update function code** window directly after the output function has been executed. When the C-Script contains internal states, they should be updated in this function rather than in the output function to ensure they are updated only once during each time step.

3 Parameters

When you open the C-Script block the **Setup** tab of the code editor window as shown in Fig. 2 will appear. In this window you can configure the parameters. You will actually write your C code within the function windows contained in the **Code** tab.

3.1 Sample time parameter

The **Sample time** setting is a key parameter that controls when the C-Script block is called. The sample time can be inherited from the simulation engine or controlled by the C-Script block itself. A description of the possible sample time settings is given below:
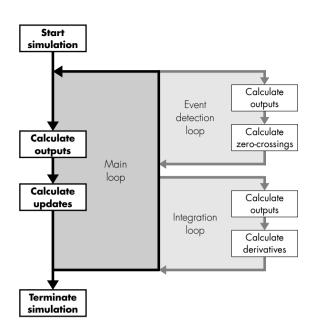
Figure 1: Function calls made during operation of the C-Script block. The update function is called when discrete states are defined and the derivative function is called when continuous states are defined.

Figure 2: C-Script editor window for configuring parameters and writing code.

Continuous   The continuous time setting is selected by entering 0 into the sample time dialog. With the continuous time setting, the time steps are inherited from the solver. Every time the solver takes a step, the C-Script block is executed.

Discrete   The discrete-periodic time setting is selected by entering a positive number into the sample time dialog. The C-Script block is executed at discrete regular intervals defined by this sample time.

Variable   The variable time setting is selected by entering -2 into the sample time dialog. With the discrete-variable time setting, the next time step is determined dynamically by the C-Script block itself by setting the `NextSampleHit` built-in macro. The `NextSampleHit` must be initialized at the beginning of the simulation to a value greater than or equal to the `CurrentTime` macro. See below for more information on macros in the C-Script block.

3.2   Other parameters

The other parameters are described completely in the C-Script block's documentation. However, it is worth noting the following at this stage. When creating a C-Script block that contains static variables,

you can add discrete states to create global static variables. The discrete states are accessed using a macro command `DiscState`.

3.3 List of commonly-used macros

The C-Script block contains a number of built-in macro functions that can be used to interact with the model or solver. Some of the commonly used macros are:

`InputSignal(j, i)`	Reference the ith signal of the jth C-Script block input.
`OutputSignal(j, i)`	Reference the ith signal of the jth C-Script block output.
`DiscState(i)`	Reference a discrete state with index i.
`NextSampleHit`	Set the next call time for the C-Script block. This variable is used when the variable sample-time setting is active.
`CurrentTime`	Retrieve the current simulation time.
`SetErrorMessage("msg")`	Abort the simulation with an error message.

4 Exercise: Implement a Mathematical Function

In this exercise, you will use the C-Script block to implement the sine function with an offset value.

Your Task:

1 Create a new simulation model, and place a C-Script component and two Constant source blocks into it. Label the first Constant block "Offset" and set its value to 0.5. Label the second Constant block "Frequency" and set its value to $2\pi \cdot 50$. Use a Signal Multiplexer block to route the two constant values into the C-Script block. Your simulation model should look like that shown in Fig. 3.
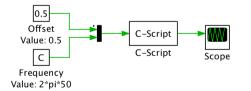
Figure 3: Implementing the function $y = 0.5 + \sin(2\pi 50 \cdot t)$ **with the C-Script block.**

2 You will then need to configure the C-Script block and write the code. To configure the C-Script block, open the block by double-clicking and in the **Setup** tab set the **Number of inputs** to 2 and the **Number of outputs** to 1. Set the **Sample time** setting to 0 to select a continuous, or inherited sample time. To write the sine function you will need to use the cmath library (`math.h` header). In the **Code declarations** window of the **Code** tab, enter the following code:

```
#include <math.h>
#define offset InputSignal(0,0)
#define freq InputSignal(0,1)
```

In the **Output function code** window, enter the following code to create the sine function:

```
OutputSignal(0,0) = sin(freq*CurrentTime) + offset;
```

3 Run the simulation: Set the simulation parameters of the PLECS solver to the following:

- Simulation stop time: $20\mathrm{e}{-}3\,\mathrm{ms}$.
- Maximum step size: $1\mathrm{e}{-}4\,\mathrm{s}$

When you run the simulation, you should see a sine wave with a period of $20\,\mathrm{ms}$ and a vertical offset of $0.5\,\mathrm{V}$.

🏁   At this stage, your model should be the same as the reference model, `sine_wave.plecs`.

5   Exercise: Implement a Digital PI Controller

In this exercise, you will replace a continuous proportional-integral (PI) voltage controller for a buck converter with a digital PI controller. The continuous PI voltage controller for the buck converter is depicted in Fig. 4. The continuous PI control law is described by the function:

$$y(t) = k_p e(t) + k_i \int_0^t e(\tau)\, d\tau \tag{1}$$

In order to implement a digital PI controller using the C-Script block, you will need to use a discrete form of the PI control law. The simplest way to discretize the PI controller is to use the backwards rectangular rule to approximate the integral term with:

$$i_k = i_{k-1} + T_s e_k \tag{2}$$

where i_k is the value at sample number k and T_s is the sample time. Thus the digital PI control law becomes:
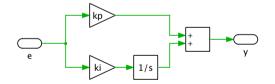
$$y_k = k_p e_k + k_i i_k \tag{3}$$

Figure 4: Continuous PI voltage controller.

5.1   Configure the C-Script block

📋   **Your Task:** Open the buck converter model `buck_converter.plecs` and look at the implementation of the continuous PI voltage controller. Save a copy of the buck converter model before you proceed with the following steps:

1   Look under the mask of the PI controller by right-clicking on the component and selecting **Look under mask** (or use **Ctrl+U**) and delete all components except for the input and output ports. Place a C-Script block directly between the input and output ports. Ensure the **Number of inputs** and **Number of outputs** in the C-Script block settings are both set to 1.

2 Add a parameter, **Sample frequency** (f_s), to the PI controller mask and set its value to $25e3\,\text{Hz}$. To add a parameter to the PI controller mask, right-click on the mask and select **Edit Mask...** (or use **Ctrl+M**).

3 In the C-Script parameters, set the **Sample time** setting to $1/f_s$. This will cause the C-Script block to execute at a discrete-periodic, or fixed sample rate of $1/f_s$.

4 The C-script code requires access to the parameters k_p, k_i and T_s. To pass these directly to the C-Script block, enter them in the **Parameters** box that is displayed in the **Setup** tab. Enter the variables k_p, k_i, $1/fs$ into the **Parameters** box.

5 Switch to the **Code** tab and in the **Code declarations** function define the following variables

```
static double kp, ki, Ts;
```

6 In the **Start** function assign the input parameters to the defined variables:

```
kp = ParamRealData(0,0);
ki = ParamRealData(1,0);
Ts = ParamRealData(2,0);
```

7 In the **Code declarations** function, also map the error input signal to the variable e_k:

```
#define ek InputSignal(0,0)
```

5.2 Implement the digital control law using a discrete state

Your Task: The control code should be written in the **Update** function, since this is only called once per sample period. On the other hand, the **Output** function is typically called several times per sample period. Therefore, any discrete states such as integrator values that are calculated in the **Output** function will be incorrect.

1 In the C-Script settings, set the **Number of disc. states** to 1. This creates a static internal variable named DiscState(0) and causes the solver to invoke the **Update** function once every sample period.

2 In the **Code declarations** function, define a global variable to represent the controller output, and map the discrete state to a variable that represents the previous integrator value, i_{k-1}.

```
double yk;
#define ik_1 DiscState(0)
```

Initialize i_{k-1} to 0 in the **Start** function. Note that y_k needs to be a global variable since it is accessed in both the **Update** and **Output** functions.

3 In the **Update** function, define the variable double ik, which is used to store intermediate results. Then implement the control law defined in Eq. (2) and (3). Don't forget to add ik_1 = ik; after calculating ik.

4 In the **Output** function, assign the result of the control law calculation, to the output OutputSignal(0,0) = yk. Note that the output can only be written to in the **Output** function.

When you run the simulation the output voltage should be the similar to the model with the continuous PI controller.

At this stage, your model should be the same as the reference model, `cscript_controller_1.plecs`.

Note: The output of the digital PI controller is delayed by one cycle because the **Update** function is called after the **Output** function, as shown in Fig. 1. The **Output** function therefore outputs the result calculated in the previous time step. The exact sequence of function calls for this simulation is depicted in Fig. 5. The number of calls to the **Output** function per time step is determined internally by the solver. For this particular model, the **Output** function is called twice during each major time step. However, for other models, the **Output** function may be called more often.

Figure 5: Timing of function calls for `cscript_controller_1.plecs`.

Eliminate the one cycle delay

Your Task: The one cycle delay can result in instability if the sample frequency is too low. To observe this effect, change the sample frequency to $10e3\,\mathrm{Hz}$ and rerun the simulation. To eliminate the delay, ensure the control result is output in the same time step it is calculated.

1 Create a global variable, `double ik`, in the **Code declarations** function.

2 Remove the control code from the **Update** function except for the line updating the discrete state, `ik_1 = ik;`

3 Shift the control code to the **Output** function:

```
ik = ik_1 + Ts*ek;
OutputSignal(0,0) = kp*ek + ki*ik;
```

In other words, the integral action is calculated in the **Output** function, but the running total, recorded by the discrete state, is not updated until the **Update** function.

At this stage, your model should be the same as the reference model, `cscript_controller_2.plecs`.

5.3   Implement the continuous control law

Although the primary function of the C-Script block is for implementing complex functions and discrete controllers, it allows continuous states and differential equations to be defined for solving ordinary differential equations of the form $\dot{x} = f(x)$. The simulation engine solves the differential equation using numerical integration. Since the integrator in Eq. (1) can be described by an ordinary differential equation:

$$\frac{di}{dt} = e(t) \tag{4}$$

the integral action can be modeled in the C-Script block by defining a continuous state, $i(t)$, and the differential equation Eq. (4). At each time step the solver will calculate $i(t)$ numerically.

For each continuous state that you define, the following macros are created: `ContState(i)` and `ContDeriv(i)`. These macros are the hooks that allow the simulation solver to solve the differential equation. All you need to do is describe the equation in the **Derivative** function.

✓ Your Task:

1 Create a copy of the model `cscript_controller_2.plecs` and reconfigure the C-Script settings. Set the **Sample time** setting to 0 and remove the parameter $1/f_s$. Set the **Number of disc. states** to 0 and the **Number of cont. states** to 1. The continuous state will be used to represent the integral term in Eq. (1).

2 In the **Code declarations** function, change the `InputSignal(0,0)` mapping to e and map the continuous state and derivative to variable names with the following:

```
#define e InputSignal(0,0)
#define I ContState(0)
#define I_deriv ContDeriv(0)
```

3 In the **Derivative** function, you need to enter the differential equation that describes the integrator. This is $I = \int e(t)\,dt$ or $dI/dt = e(t)$, therefore enter `I_deriv = e;`. The solver will then solve the differential equation to yield the integrator value, I.

4 The appropriate initial value for the integrator value $I = 0$ is set in the **Start** function.

5 Remove all code from **Update** functions, and in the **Output** function, remove all code except for the following:

```
OutputSignal(0,0) = kp*e+ki*I;
```

When you run the simulation, you should see the same output voltage as with the original continuous PI controller.

🏁 At this stage, your model should be the same as the reference model, `cscript_controller_4.plecs`.

When to use a continuous state

In this example, implementing an integrator by creating a continuous state and defining a differential equation was more work than using the integrator component itself. However, working with continuous states inside the C-Script block allows you to add advanced functionality to differential equations or state-space systems. For example, you can implement an integrator that resets itself when its output reaches a certain value. This is not possible using a standard integrator component with a comparator, since feeding back the comparator output to the integrator reset port creates an algebraic loop.

6 Advanced Exercise: Implement a Digital PI Controller with Calculation Delay

In Section 5.2 you implemented a PI controller without a calculation delay. In a practical system, a finite delay exists due to the time needed for the controller to read the input(s), perform the control

calculation and write to the output(s). This delay can degrade the stability for certain systems. To simulate this calculation delay, a delay time is introduced before the control result, y_k, is written to `OutputSignal(0,0)`.

☑ Your Task:

1 Save a copy of the model `cscript_controller_2.plecs` and add an additional parameter to the voltage controller mask labeled **Calculation delay**. Assign this to a variable named t_d and set its value to 0.1. This will be used to set the calculation delay time to $0.1T_s$.

2 In the C-Script block settings, add the argument t_d/f_s to the list of **Parameters** in the **Setup** tab and define a variable T_d in the **Code declarations** function:

```
static double Td;
```

and assign the value `Td = ParamRealData(3,0);` in the **Start** function.

3 To implement the calculation delay, you will first need to implement a hybrid discrete-variable, sample time setting. The fixed-step setting will provide a sample hit at the beginning of each period and the variable time step will provide a hit after the calculation delay. Hybrid time settings must be entered as a matrix format, where the first entry in a row is the sample time and the second entry is the offset time. Enter the following **Sample time** setting: `[1/fs, 0; -2, 0]`

4 To ensure the first hit time is generated by the fixed time step setting, you should initialize the `NextSampleHit` macro, which defines the variable step hit time, to a large number in the **Start** function: `NextSampleHit = NEVER;`

5 Note that you will need to define `NEVER` as a very large number in the **Code declarations** function. If you include the file <float.h> you can define `NEVER` as `DBL_MAX`, the largest machine representable float number.

6 At the beginning of the switching cycle you will need to carry out the control calculations for i_k and y_k. The calculated control action, y_k, is not output until the next call to the **Output** function, which will occur at the time `CurrentTime + Td`. Add the following lines in the **Update** function:

```
if (NextSampleHit == NEVER) //beginning of switching cycle
{
    //Control calculations for ik, ik_1, yk here
    NextSampleHit = CurrentTime + Td;
}
else
    NextSampleHit = NEVER;
```

7 In the **Output** function, assign y_k to the output port in order to output the control action that was calculated at the beginning of the switching cycle.

🏁 At this stage, your model should be the same as the reference model, `cscript_controller_3.plecs`. To observe the influence of the calculation delay, set f_s to $10e3$ Hz and run the simulation for a calculation delay of 0.1 and 0.9. Note that this implementation only allows values $t_d \in {]}0, 1{[}$, the treatment of the special cases 0 and 1 is left for the user as an additional exercise.

7   Conclusion

In this exercise you learned how to use the PLECS C-Script block to implement a custom digital PI controller with several adaptations. This involved having an understanding of the function calls that are predefined in the block and are used in order to interface with the simulation engine. Another important aspect of the C-Script block is understanding the different time settings that are available for configuration in the block, which are crucial for ensuring certain desired behavior such as dynamic step size calls for timing functions or mimicking a fixed-step controller. The PLECS C-Script block is highly versatile and can be used to model elaborate controllers and components.

Revision History:

Tutorial Version 1.0   First release

How to Contact Plexim:

PLECS Tutorial