Script-based Simulation Control using GeckoSCRIPT

GeckoCIRCUITS Coupling with External Software

Script-based Simulation Control using GeckoSCRIPT

GeckoCIRCUITS is a powerful circuit simulator optimized for power electronics. Its library offers the user a large number of circuit, control and thermal components for modeling power electronic systems. The user must properly set element parameters such as on-resistance, forward voltage, controller coefficients, inductance, or thermal capacity in order to model a system accurately.

There is a wide range of problems – e.g. optimizations for efficiency or controller tuning – for which the user must simulate, change some element parameters, re-simulate, and so on until a desired solution is found. As of version 1.4, GeckoCIRCUITS provides a way of automating such a process.

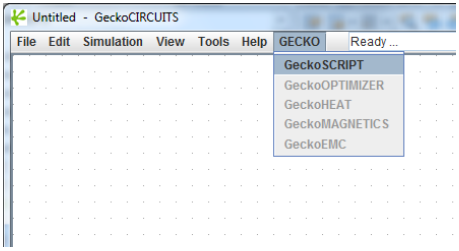
GeckoSCRIPT is a tool within GeckoCIRCUITS which allows for script-based simulation control. Via the GeckoSCRIPT tool, the user can write and execute a script in the Java programming language, using specially provided functions for manipulating the model parameters and simulation inside GeckoCIRCUITS, as well as the full Java API.

In this tutorial, we will show the functionality of the GeckoSCRIPT tool through the example of tuning a controller for a buck converter. We will also show how to use the GeckoSCRIPT functions to allow scrip-based control of GeckoCIRCUITS from MATLAB.

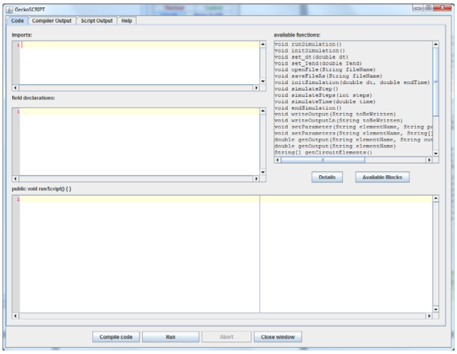
Introduction

Script-based Simulation Control using GeckoSCRIPT

To open the GeckoSCRIPT tool, start GeckoCIRCUITS, and click on the GECKO menu (furthest to the right on the menu bar). There you will find a menu item called GeckoSCRIPT.



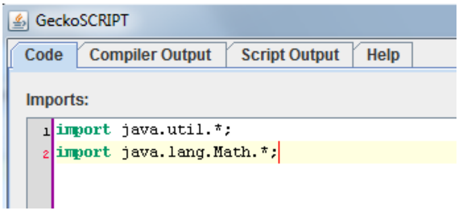
This will open the scripting environment:



```
void runSimulation()
void initSimulation()
void set_dt(double dt)
void set_Tend(double Tend)
void openFile(String fileName)
void saveFileAs(String fileName)
void initSimulation(double dt, double endTime)
void simulateStep()
void simulateSteps(int steps)
void simulateTime(double time)
void endSimulation()
void writeOutput(String toBeWritten)
void writeOutputLn(String toBeWritten)
void setParameter(String elementName, String pa
void setParameters(String elementName, String[]
double getOutput(String elementName, String out
double getOutput(String elementName)
String[] getCircuitElements()
```

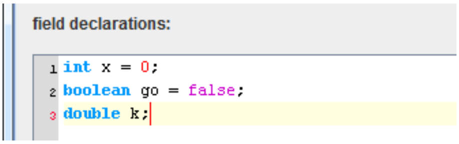
Users already familiar with Gecko-CIRCUITS will note the similarity with the Java block interface. There are four tabs at the top of the window, with the 'Code' tab opened by default. There are white editable text fields which comprise the GeckoSCRIPT code editor.

The top field, Imports, allows you to im-port any Java class or package for use within your script (e.g. java.util). Therefore the entire Java API is available for use.



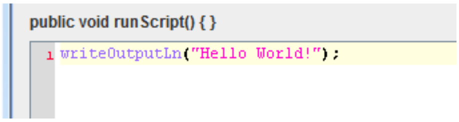
```
import java.util.*;
import java.lang.Math.*;
```

The middle field, field declarations, allows you to define and initialize variables for use in your script.



```
int x = 0;
boolean go = false;
double k;
```

The text field public void runScript() at the bottom is where you write the code for your script.

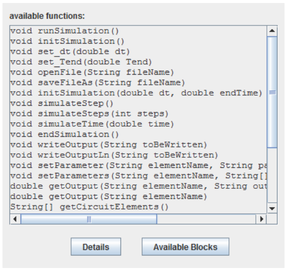


```
writeOutputLn("Hello World!");
```

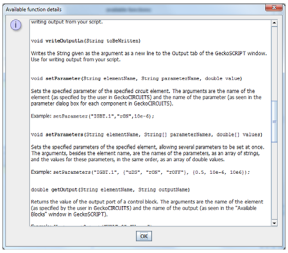
All coding is done in the Java program-ming language and correct Java syntax must be used.

Getting Started
GeckoCIRCUITS version 1.4 or later is required

The grey non-editable upper-right window lists the GeckoSCRIPT-specific functions which are available for modifying the parameters of an element of a model and for controlling the execution of the simulation of the model. A detailed description of each function with a simple example of its use is available by clicking on the Details button.


available functions:

```
void runSimulation()
void initSimulation()
void set_dt(double dt)
void set_Tend(double Tend)
void openFile(String fileName)
void saveFileAs(String fileName)
void initSimulation(double dt, double endTime)
void simulateStep()
void simulateSteps(int steps)
void simulateTime(double time)
void endSimulation()
void writeOutput(String toBeWritten)
void writeOutputLn(String toBeWritten)
void setParameter(String elementName, String pa
void setParameters(String elementName, String[]
double getOutput(String elementName, String out
double getOutput(String elementName)
String[] getCircuitElements()
```


Available function details

writing output from your script.

`void writeOutputLn(String toBeWritten)`

Writes the String given as the argument as a new line to the Output tab of the GeckoSCRIPT window. Use for writing output from your script.

`void setParameter(String elementName, String parameterName, double value)`

Sets the specified parameter of the specified circuit element. The arguments are the name of the element (as specified by the user in GeckoCIRCUITS) and the name of the parameter (as seen in the parameter dialog box for each component in GeckoCIRCUITS).

Example: `setParameter("IGBT.1","rON",10e-6);`

`void setParameters(String elementName, String[] parameterNames, double[] values)`

Sets the specified parameters of the specified element, allowing several parameters to be set at once. The arguments, besides the element name, are the names of the parameters, as an array of strings, and the values for these parameters, in the same order, as an array of double values.

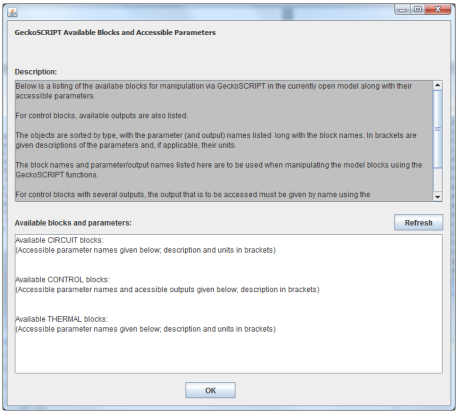
Example: `setParameters("IGBT.1", {"uDS", "rON", "rOFF"}, {0.5, 10e-6, 10e6});`

`double getOutput(String elementName, String outputName)`

Returns the value of the output port of a control block. The arguments are the name of the element (as specified by the user in GeckoCIRCUITS) and the name of the output (as seen in the "Available Blocks" window in GeckoSCRIPT).

GeckoSCRIPT functions work by operating on element and parameter names, which must be passed to the functions as Java Strings (i.e. enclosed by “”). The names of circuit and thermal elements are visible in GeckoCIRCUITS in the schematic window, and in the parameter setting dialog box that appears when elements are clicked on. Names of control elements are visible in the parameter dialog box. The names of the parameters used by GeckoSCRIPT are mostly identical to the names of the parameters used in the dialog boxes.

To simplify the writing of scripts, GeckoSCRIPT provides a listing of all blocks available in a model, their names, and for each type of block, the accessible parameters and outputs, along with their description. To show this listing, click on the Available Blocks button. As our model is currently blank, the list is empty.


GeckoSCRIPT Available Blocks and Accessible Parameters

Description:

Below is a listing of the availabe blocks for manipulation via GeckoSCRIPT in the currently open model along with their accessible parameters.

For control blocks, available outputs are also listed.

The objects are sorted by type, with the parameter (and output) names listed long with the block names. In brackets are given descriptions of the parameters and, if applicable, their units.

The block names and parameter/output names listed here are to be used when manipulating the model blocks using the GeckoSCRIPT functions.

For control blocks with several outputs, the output that is to be accessed must be given by name using the

Available blocks and parameters:

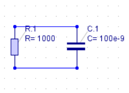
Available CIRCUIT blocks:
(Accessible parameter names given below; description and units in brackets)

Available CONTROL blocks:
(Accessible parameter names and acessible outputs given below; description in brackets)

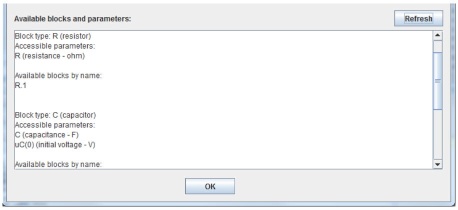
Available THERMAL blocks:
(Accessible parameter names given below; description and units in brackets)

Getting started
GeckoSCRIPT-specific functions for manipulating model blocks and controlling the simulation

Now, go back to the GeckoCIRCUITS main window, and construct a simple RC circuit schematic:



Then return to the Available Blocks window and click the Refresh button. You will now notice the two circuit elements listed, with the names of their accessible parameters and their descriptions.



Block type: R (resistor)
Accessible parameters:
R (resistance - ohm)

Available blocks by name:
R.1

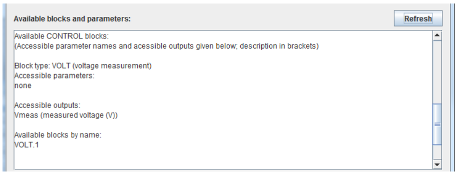
Block type: C (capacitor)
Accessible parameters:
C (capacitance - F)
uC(0) (initial voltage - V)

Available blocks by name:

When accessing and modifying the parameters of the blocks in the model using the GeckoSCRIPT functions, you must use the names listed here.

For control elements (which include the measurement blocks, e.g. voltmeters and amperemeters), the outputs of the block are also accessible. Accessing control element outputs is the means by which simulation results are extracted from the model through GeckoSCRIPT. SCOPEs are not accessible via GeckoSCRIPT.

To see a listing of available outputs, add a voltage measurement block to the model (e.g. measuring the voltage across the resistor). Click the 'Refresh' button again. You will see a list of output names and their description.



Available CONTROL blocks:
(Accessible parameter names and acessible outputs given below; description in brackets)

Block type: VOLT (voltage measurement)
Accessible parameters:
none

Accessible outputs:
Vmeas (measured voltage (V))

Available blocks by name:
VOLT.1

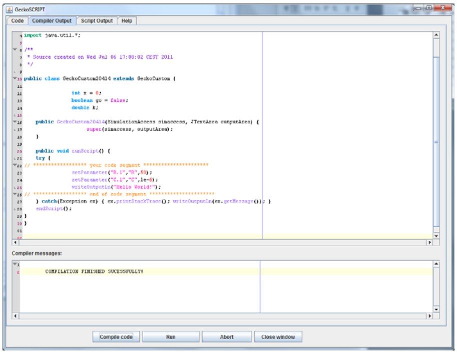
Now let's write a simple script and try to compile it. In the script code field, enter the following code:

```
setParameter("R.1","R",50);
setParameter("C.1","C",1e-6);
writeOutputLn("Hello World!");
```

The setParameter function changes the parameter of an element. From the list we looked at previously, we can see than the resistor's name is R.1, and that the parameter denoting resistance is named R. So we use these as the arguments to the function, along with the value of the resistance we wish to set. The same is done for the capacitor and its capacitance.

The writeOutputLn function is provided in order to write any sort of output you may want to the Script Output tab. You can also write any output you like to a file by using the file I/O classes provided by the Java API.

Now click on the Compile Code button. You will notice that the GeckoSCRIPT window automatically switches to the Compiler Output tab. This tab contains two text fields. On top is the full Java code that is compiled. GeckoSCRIPT creates a custom class, along with the imports and field declarations specified by the user, with a runScript() method that executes the user-specified script.

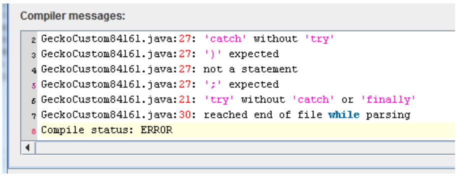


```
4  import java.util.*;
5
6  /**
7   * Source created on Wed Jul 06 17:00:02 CEST 2011
8   */
9
10 public class GeckoCustom20414 extends GeckoCustom {
11
12                 int x = 0;
13                 boolean go = false;
14                 double k;
15
16     public GeckoCustom20414(SimulationAccess simaccess, JTextArea outputArea) {
17                     super(simaccess, outputArea);
18     }
19
20     public void runScript() {
21     try {
22 // ***************** your code segment *********************
23                 setParameter("R.1","R",50);
24                 setParameter("C.1","C",1e-6);
25                 writeOutputLn("Hello World!");
26 // ***************** end of code segment *********************
27     } catch(Exception ex) { ex.printStackTrace(); writeOutputLn(ex.getMessage()); }
28     endScript();
29 }
30 }
31
```

Compiler messages:

```
1
2       COMPILATION FINISHED SUCESSFULLY!
```

The lower text field displays messages from the compiler. If your code is free of syntax errors, you will receive a message that the compilation has been successful. Otherwise, the errors from the compiler will be displayed along with line numbers where the errors are found. For the line numbers, refer to the upper text field displaying the full code.



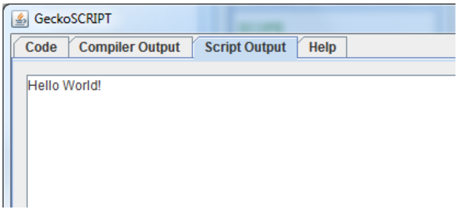
Compiler messages:

```
2 GeckoCustom84161.java:27: 'catch' without 'try'
3 GeckoCustom84161.java:27: ')' expected
4 GeckoCustom84161.java:27: not a statement
5 GeckoCustom84161.java:27: ';' expected
6 GeckoCustom84161.java:21: 'try' without 'catch' or 'finally'
7 GeckoCustom84161.java:30: reached end of file while parsing
8 Compile status: ERROR
```

Getting started
The list of block, parameter, and output names that is used to modify the model

Writing a simple script
The **setParameter(...)** function
The **writeOutputLn(...)** function
Compiler output and messages

Now let's run the simple script written previously. Click on the Run button. You will notice that the GeckoSCRIPT window automatically switches to the Script Output tab. This is where any output written using the writeOutput or the writeOutputLn functions will be displayed. The text field under this tab now displays the phrase Hello World!, as was specified in the script. Please go back to the GeckoCIRCUITS main window. You will notice that the resistor and capacitor parameter values have been changed to those specified in the script. GeckoSCRIPT has performed as instructed.

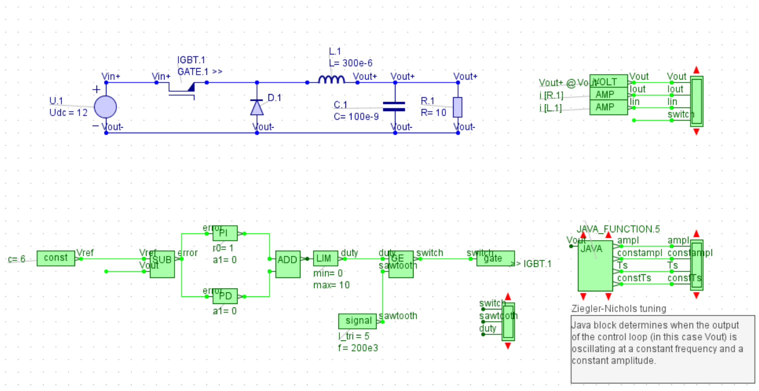


```
Hello World!
```

You will notice there is also a Help tab. It displays some basic information about GeckoSCRIPT. Now, click Close Window to close the GeckoSCRIPT window. Any code you write is saved along with the model.

Now that you have familiarized yourself with the GeckoSCRIPT environment, we will demonstrate its functionality using a simple example where the controller parameters for a buck converter are tuned to achieve a stable voltage regulation.

Running a simple script
GeckoSCRIPT output
Check parameter values have changed



Java block determines when the output of the control loop (in this case Vout) is oscillating at a constant frequency and a constant amplitude.

Included with this tutorial is a file called buck_control.ipes Open this model in GeckoCIRCUITS.
It contains a circuit with a simple Buck converter, together with a control loop modelling a PID controller using the PI and PD control blocks available in Gecko-CIRCUITS. The proportional gain is set to 1, and the integral and differential gains to zero, i.e. the loop is uncompensated. Via the constant block, the output reference is set to 6 V, and the input voltage is 12 V. If you run the simulation and open the uppermost SCOPE showing the output voltage, you will see the output voltage is 3 V. Therefore, a proper controller giving required output voltage regulation must be designed.

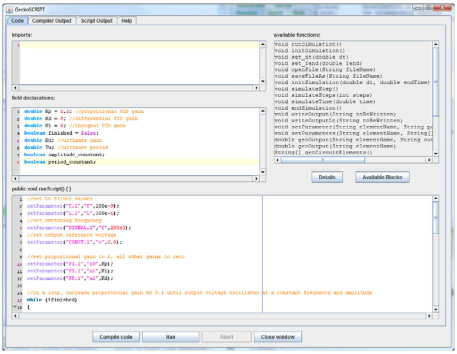
A common approach to PID controller tuning is the Ziegler-Nichols method. In this method, with the other gains set to zero, the proportional gain is increased until the output of the control loop – in this case the output voltage of the converter – oscillates at a constant frequency and amplitude. This ultimate gain, denoted ***Ku***, and the period of this oscillation, denoted ***Tu***, are then used to calculate the three PID controller coefficients – the proportional gain *Kp*, integral gain *Ki* and differential gain *Kd*.

Now this of course can be done by the user manually: simulate, increase the proportional gain in the model's PI block, simulate, and repeat until *Ku* and *Tu* are found. A much more convenient and faster way to do it is to use GeckoSCRIPT.

To facilitate this, a Java block is included in the model that monitors the output voltage, and determines whether it oscillates and constant frequency and amplitude. The outputs of the Java block are the amplitude of oscillation, a digital signal indicating (1/0 i.e. true/false) whether the amplitude is constant, the period of oscillation and a digital signal indicating whether the period is constant.

Controller tuning example
Open buck converter model
Ziegler-Nichols tuning procedure

Now open GeckoSCRIPT. A script implementing the Ziegler-Nichols algorithm has already been included with the model.



Imports:

field declarations:

```
double Kp = 2.2; //proportional PID gain
double Kd = 0; //differential PID gain
double Ki = 0; //integral PID gain
boolean finished = false;
double Ku; //ultimate gain
double Tu; //ultimate period
boolean amplitude_constant;
boolean period_constant;
```

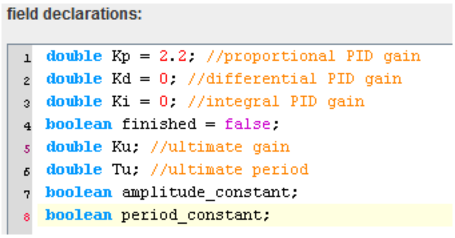
available functions:

```
void runSimulation()
void initSimulation()
void set_dt(double dt)
void set_Tend(double Tend)
void openFile(String fileName)
void saveFileAs(String fileName)
void initSimulation(double dt, double endTime)
void simulateStep()
void simulateSteps(int steps)
void simulateTime(double time)
void endSimulation()
void writeOutput(String toBeWritten)
void writeOutputLn(String toBeWritten)
void setParameter(String elementName, String pa
void setParameters(String elementName, String[]
double getOutput(String elementName, String out
double getOutput(String elementName)
String[] getCircuitElements()
```

public void runScript() { }

```
 1  //set LC filter values
 2  setParameter("C.1","C",100e-9);
 3  setParameter("L.1","L",300e-6);
 4  //set switching frequency
 5  setParameter("SIGNAL.2","f",200e3);
 6  //set output reference voltage
 7  setParameter("CONST.1","c",6.0);
 8
 9  //set proportional gain to 1, all other gains to zero
10  setParameter("PI.1","r0",Kp);
11  setParameter("PI.1","a1",Ki);
12  setParameter("PD.1","a1",Kd);
13
14  //in a loop, increase proportional gain by 0.1 until output voltage oscillates at a constant frequency and amplitude
15  while (!finished)
16  {
```

In the field declarations we have declared all the variables we need for the algorithm: the PID controller coefficients, the ultimate gain and period, Boolean flags for whether the amplitude and frequency are constant, and a flag that tells us when to stop searching.

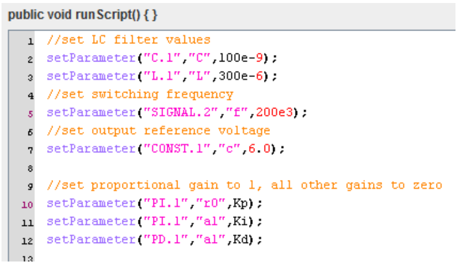


field declarations:

```
double Kp = 2.2; //proportional PID gain
double Kd = 0; //differential PID gain
double Ki = 0; //integral PID gain
boolean finished = false;
double Ku; //ultimate gain
double Tu; //ultimate period
boolean amplitude_constant;
boolean period_constant;
```

Now click the Available Blocks button. You will then see a listing of all the circuit and control blocks in the circuit by type and name, their accessible parameter names along with the parameter descriptions, and their accessible outputs (for control blocks).

The first part of the script code initializes the most important model values. Using the setParameter function, the inductance and capacitance values of the converter output LC filter are set, the output voltage reference, and the initial PID coefficents.



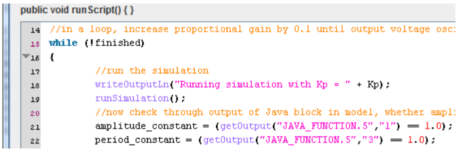
public void runScript() { }

```
 1  //set LC filter values
 2  setParameter("C.1","C",100e-9);
 3  setParameter("L.1","L",300e-6);
 4  //set switching frequency
 5  setParameter("SIGNAL.2","f",200e3);
 6  //set output reference voltage
 7  setParameter("CONST.1","c",6.0);
 8
 9  //set proportional gain to 1, all other gains to zero
10  setParameter("PI.1","r0",Kp);
11  setParameter("PI.1","a1",Ki);
12  setParameter("PD.1","a1",Kd);
13
```

On line 5 we set the switching frequency of the converter, by setting the frequency parameter of the signal generator block that creates a sawtooth waveform used to generate a pulse-width modulated signal for the converter main switch.

Controller tuning example
Script written for the Ziegler-Nichols tuning algorithm

The following code increases the proportional gain in a loop while running the simulation and checking whether *Ku* has been found.



public void runScript() { }

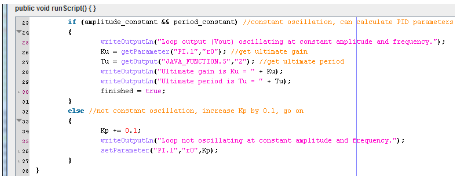
```
14 //in a loop, increase proportional gain by 0.1 until output voltage osc:
15 while (!finished)
16 {
17         //run the simulation
18         writeOutputLn("Running simulation with Kp = " + Kp);
19         runSimulation();
20         //now check through output of Java block in model, whether ampl:
21         amplitude_constant = (getOutput("JAVA_FUNCTION.5","1") == 1.0);
22         period_constant = (getOutput("JAVA_FUNCTION.5","3") == 1.0);
```

We use a standard Java while loop, and after displaying a short message, we call the runSimulation() function in line 19. Calling this function executes the simulation according to the parameters (stepwidth and end-time) specified in the simulation configuration dialog accessible via the Simulation -> Parameter menu. In this dialog, you are able to define parameters as start- and end-times of the simulation as well as other solver options. Calling the runSimulation() function within GeckoSCRIPT has the same effect as the user clicking Simulation -> Init & Start in the GeckoCIRCUITS main window.

The next step is to check the Java block outputs in the model to see whether the amplitude and period of oscillation of the output voltage is constant, which indicates whether we should stop the search or continue.

This is done via the getOutput function. As you can see, the arguments for this function are the name of the control block (JAVA_FUNCTION.5 in this case), and the name of the output. As the output of the Java block can represent different values, and the since the number of outputs can vary, the outputs are simply designated numerically from top to bottom, starting from zero. Note that the names are however still Strings, not numbers. In this case we are interested in output "1" and "3".

In the next part of the code, if the amplitude and period of oscillation are indeed constant, we retrieve the observed period of oscillation *Tu* from the simulation – in this case output "2" of the Java block – and the ultimate gain *Ku*, which is parameter "r0" of the PI control block.



public void runScript() { }

```
        if (amplitude_constant && period_constant) //constant oscillation, can calculate PID parameters
        {
                writeOutputLn("Loop output (Vout) oscillating at constant amplitude and frequency.");
                Ku = getParameter("PI.1","r0"); //get ultimate gain
                Tu = getOutput("JAVA_FUNCTION.5","2"); //get ultimate period
                writeOutputLn("Ultimate gain is Ku = " + Ku);
                writeOutputLn("Ultimate period is Tu = " + Tu);
                finished = true;
        }
        else //not constant oscillation, increase Kp by 0.1, go on
        {
                Kp += 0.1;
                writeOutputLn("Loop not oscillating at constant amplitude and frequency.");
                setParameter("PI.1","r0",Kp);
        }
}
```

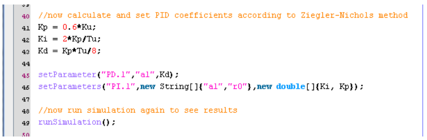
The getParameter function allows us to do this, by specifying the block and parameter name we wish to read.

Controller tuning example
runSimulation()
getOutput(...)
getParameter(...)

Then a message is displayed in the script output that the tuning process is finished, and the flag to exit the loop is set.

If the amplitude and period are not both constant, we increase the proportional gain by 0.1, set this in the model as the new proportional gain, and continue running the loop.

Once the loop finishes, the last part of the code calculates the PID coefficients according to the Ziegler-Nichols PID tuning rule, and sets these parameters in the PI and PD control blocks in the model.



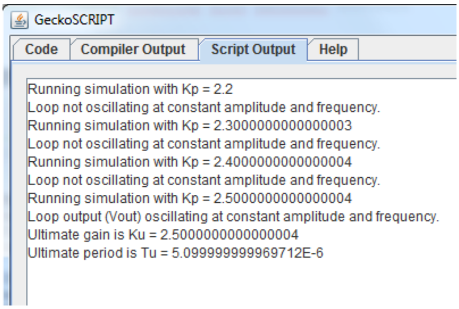
```
40 //now calculate and set PID coefficients according to Ziegler-Nichols method
41 Kp = 0.6*Ku;
42 Ki = 2*Kp/Tu;
43 Kd = Kp*Tu/8;
44 
45 setParameter("PD.1","a1",Kd);
46 setParameters("PI.1",new String[]{"a1","r0"},new double[]{Ki, Kp});
47 
48 //now run simulation again to see results
49 runSimulation();
50 
```

Please note the use of the setParameters function in line 46. This allows you to set all the parameters of a block at once, by specifying the block name, and the parameter names and values as an array of Strings and numbers respectively.

At the end of the script, the simulation is run again in order to see the final results of the tuning algorithm.

Now that you have examined and understood the script you can compile it. Compilation should finish successfully.

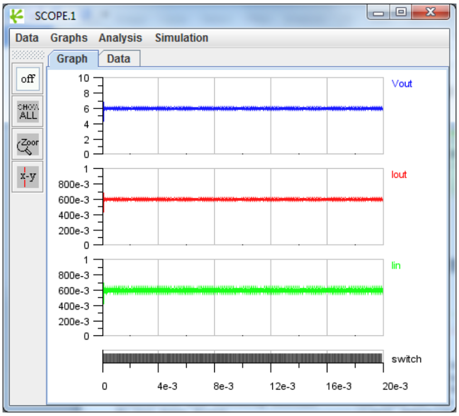
Keeping the SCOPE which displays the output voltage open, run the script. You will see the output of the script change as the loop is executed, and you will also see the model re-simulated (with different results, as the gain is changing) with each loop iteration.



```
Running simulation with Kp = 2.2
Loop not oscillating at constant amplitude and frequency.
Running simulation with Kp = 2.3000000000000003
Loop not oscillating at constant amplitude and frequency.
Running simulation with Kp = 2.4000000000000004
Loop not oscillating at constant amplitude and frequency.
Running simulation with Kp = 2.5000000000000004
Loop output (Vout) oscillating at constant amplitude and frequency.
Ultimate gain is Ku = 2.5000000000000004
Ultimate period is Tu = 5.099999999969712E-6
```

Finally, the loop will stop, and the SCOPE will sho w a converter regulating the output voltage at 6 V which is the pre-defined reference voltage. A stable set of controller coefficients have been found. These are now displayed in the main window underneath the control blocks.

Controller tuning example
setParameters(...)
Compiling the script
Running the script



The gain at which a control loop oscillates at constant period and amplitude is not necessarily unique. The initial value of the proportional gain in this script has been rather arbitrarily set to 2.2. Try using a different value, e.g. 0.5 or 1.0, and see what happens.

Also, try changing the output reference voltage, the switching frequency, or the LC filter values by modifying the appropriate lines in the script code. By doing so you change the dynamics of the system, which results in different PID coefficients. Try making such changes, recompile the script, and re-run it to see the different results.

Also, try using a function to access a parameter or output or block that does not exist in the model. Such errors are not caught at compile-time since they depend on the model itself, but run-time errors will be displayed in the script output tab if you attempt to access something which does not exist.

Controller tuning example
Modifying the script
Run-time errors

So far we have shown how to write simulation control scripts using the scripting environment provided within GeckoCIRCUITS. However, sometimes coupling a GeckoCIRCUITS simulation with another simulation or computation program is desirable.

All of functions for model manipulation and simulation control available in GeckoSCRIPT within GeckoCIRCUITS, some of which were described previously, are also available for control of GeckoCIRCUITS through and within MATLAB®.

This allows a user to load GeckoCIRCUITS into the MATLAB environment, control a simulation from the MATLAB environment (from the command line or an m-file), to use MATLAB functions to calculate parameters for the simulation, and to plot simulation results – or the results of several simulations – within MATLAB.

In the following we will show how a script for PID tuning like that presented in the previous sections can be implemented using GeckoCIRCUITS and MATLAB.

This tutorial was implemented and tested on MATLAB release 2010b. Java 1.6 has to be the actual Java version in the MATLAB environment. Before proceeding make sure that Java 1.6 is the actual Java version in the MATLAB environment:

```
>> version -java

ans =
Java 1.6.0 with Sun
Microsystems Inc. Java
HotSpot(TM) Client VM mixed
mode
```

In the Help Directory of MATLAB you will find a description of how to install a newer Java version if necessary.

Controlling GeckoCIRCUITS in MATLAB using GeckoSCRIPT functions
Check the Java version in MATLAB

In order to use GeckoCIRCUITS in MATLAB, you must first tell MATLAB of its location. If you are using a Windows machine and have installed GeckoCIRCUITS into a directory called C:/GeckoCIRCUITS/, then the path to the GeckoCIRCUITS simulator is

```
C:/GeckoCIRCUITS/GeckoCIRCUITS.jar
```

To add this to MATLAB's class path, append the above line to MATLAB's classpath.txt file. You can do this within MATLAB by typing

```
>> edit classpath.txt
```

which will open that file in the editor.

Please note that by default, the file permission is set to read-only. Therefore you may need administrator rights to change the file permission.

If you don't have the permission to change the file classpath.txt, you can also define the path locally for your MATLAB session to a dynamic class path. Therefore, execute the following command:

```
>> javaaddpath 'C:/GeckoCIRCUITS
/GeckoCIRCUITS.jar'
```

You can view and undo your changes to the dynamic class path employing the commands

```
>> javaclasspath
>> javarmpath
```

When you have set the static class path by editing classpath.txt, then you must restart MATLAB to apply the changes. The dynamic class path setting, however, will only persist during your current MATLAB session.

Please save your simulation model and close GeckoCIRCUITS now.

Controlling GeckoCIRCUITS in MATLAB using GeckoSCRIPT functions
Add path to GeckoCIRCUITS

The functions for controlling a GeckoCIRCUITS simulation and model from MATLAB are identical to those listed in the GeckoSCRIPT main window and function in the same way. The functions for MATLAB are located in a Java package called gecko.GeckoRemote.*.

To avoid calling the functions always with the full package name, import this package into MATLAB with the command

```
>> import('gecko.GeckoRemote.*')
```

As of version 1.6, communication between MATLAB and GeckoCIRCUITS is done using network ports on the local machine. This allows GeckoCIRCUITS to run outside of MATLAB, in your operating system's Java Virtual Machine (JVM), rather than MATLAB's, while still allowing you to control simulations and access their results from MATLAB.

As a result, besides the list of functions you have already seen in GeckoSCRIPT, for control from MATLAB there are several more functions necessary. One of them is startGui(int port). This function starts a new instance of GeckoCIRCUITS, enabled for remote access, from MATLAB. You must specify a free network port number. If you do not, GeckoCIRCUITS will attempt to use the default port, 43035. Therefore, before you proceed, type into the MATLAB command line

```
>> startGui()
```

This will open the GeckoCIRCUITS main window. You will also see this message displayed in the MATLAB command line:

```
Starting GeckoCIRCUITS in OS JVM
(outside MATLAB)...
GeckoCIRCUITS should now be running
outside of MATLAB at port 43035
You are now connected to the
GeckoCIRCUITS instance at port
43035
```

If an error message is displayed instead, the port is already being used or is blocked. Try then another port number. On most machines, the default port 43035 should be free.

It is also possible that any firewall program you have installed on your computer will complain about unauthorized access from MATLAB or the Java Runtime Environment. On Windows machines for example, if prompted by a message from the firewall, click "Allow Access" in either case. Even if you do not have administrative rights to do this, ignoring such messages (i.e. clicking "Cancel" in this example) should not be a problem, since both MATLAB and GeckoCIRCUITS are running on the local machine.

NOTE: If you are already familiar with GeckoCIRCUITS and MATLAB, and have previously used the old package GeckoExternal, see Appendix for instructions on migrating to GeckoRemote.

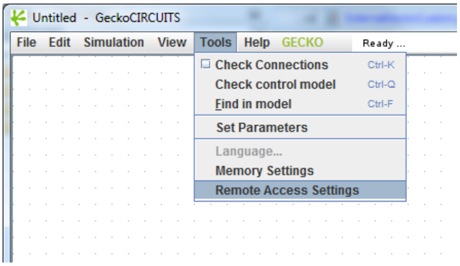
Controlling GeckoCIRCUITS in MATLAB using GeckoSCRIPT
Import GeckoRemote package
startGui()

If problems with the firewall persist, please contact your System Administrator and ask him to allow MATLAB and Java access to the network ports. Ask for advice on which network ports to use.

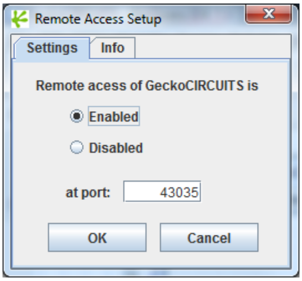
Now, it is not necessary to start GeckoCIRCUITS from inside MATLAB using the startGui() function. You can connect from MATLAB to GeckoCIRCUITS if it is already running, provided it is enabled for remote access. To try this out, let us first disconnect from the GeckoCIRCUITS instance we started from MATLAB. To do this, type

```
>> disconnectFromGecko()
```

A message in the MATLAB command window will inform you that the connection has severed. However GeckoCIRCUITS is still running. Go to the GeckoCIRCUITS window. Under the Tools menu click the Remote Access Settings option.



A dialog window will open, allowing you to enable or disable remote access and control of GeckoCIRCUITS, and at which port.



Remote acess of GeckoCIRCUITS is

(● Enabled) ○ Disabled

at port: 43035

Since everything was set up as we want it already when we called startGui(), click "OK". Now go back to the MATLAB window. Connect again to GeckoCIRCUITS – this time let's specify the port explicitly – by typing into the command line

```
>> connectToGecko(43035)
```

Once more there will be a notification message that you are now connected:

```
You are now connected to the
GeckoCIRCUITS instance at port
43035
```

Controlling GeckoCIRCUITS in MATLAB using GeckoSCRIPT
Connecting to an already-running instance of GeckoCIRCUITS

Now, let's go back to our simulation task.

Open the model you previously modified and saved, buck_control.ipes. You can do this from MATLAB by typing

```
>> openFile('buck_control.ipes')
```

Assuming the file is in your MATLAB working directory (otherwise, specify the correct path).

We will repeat the tuning procedure from before, now from within MATLAB. Since we saved the model last time with the new PID coefficients, we need to reset the control blocks to their original, un-tuned values. Do this from the MATLAB command line by typing

```
>> setParameter('PD.1', 'a1',0)
>> setParameter('PI.1','a1',0)
>> setParameter('PD.1', 'r0',1)
```

You will see the parameters change in GeckoCIRCUITS. Note the different use of quotation marks for strings (single rather than double as in Java code). This is because in MATLAB you must respect MATLAB syntax, even though the functions are imported from a Java package.

You can now verify that the converter is again not regulating the output voltage properly by typing into MATLAB

```
>> runSimulation()
```

which will simulate the model as shown earlier from within GeckoSCRIPT.

Controlling GeckoCIRCUITS in MATLAB using GeckoSCRIPT
Using GeckoSCRIPT functions from the MATLAB command line

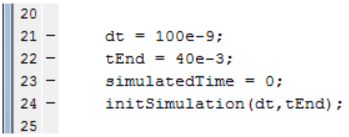
Included with this tutorial is a file called buck_control_tuning.m, which you can open from MATLAB. It is an m-file that performs the Ziegler-Nichols tuning, just like the script we previously analysed and ran from within GeckoCIRCUITS.

There are however, some differences. If you look at the simulation we performed during the tuning process earlier, you will notice that the simulation was performed each time for 0.02 seconds. However, we can tell whether we have reached *Ku* much earlier than that. We only need to see a few cycles. Wouldn't it be better if we could run a part of the simulation, check the results, then change the parameters if necessary, and continue, and so on?

There are certain GeckoSCRIPT functions which allow for just that.

You can use GeckoSCRIPT functions to simulate one single simulation step at a time, so simulate several steps at a time, and to simulate a specified certain time (e.g. 10 microseconds).

Now let's take a look at the m-file. Up to line 21, the script does the same as previously, only with MATLAB syntax rather than Java. Now look at lines 21 to 24. In order to control a simulation step-by-step, we first must initialize it and for this we use the function initSimulation.



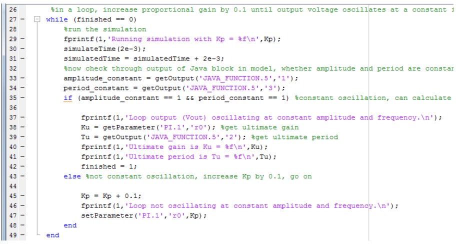
```
20
21 -      dt = 100e-9;
22 -      tEnd = 40e-3;
23 -      simulatedTime = 0;
24 -      initSimulation(dt,tEnd);
25
```

We define a step-width dt and simulation time tEnd and pass these as arguments to the function. We also define a variable to keep track of how much of the simulation time we've covered.

You can also call initSimulation() without arguments. In that case the simulation is initialized with parameters specified from within GeckoCIRCUITS, just like when you call runSimulation().

Controlling GeckoCIRCUITS in MATLAB using GeckoSCRIPT
m-file with tuning algorithm
Doing a partial simulation

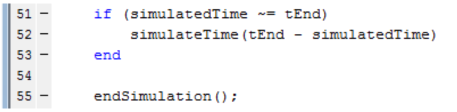
Now let's take a look at the loop which performs the tuning. Apart from the MAT-LAB / Java syntax differences, you will notice that instead of using the runSimulation() function as before, which runs the entire simulation up to tEnd, we are using simulateTime.



```
26         %in a loop, increase proportional gain by 0.1 until output voltage oscillates at a constant f
27 -  ⊟ while (finished == 0)
28          %run the simulation
29 -        fprintf(1,'Running simulation with Kp = %f\n',Kp);
30 -        simulateTime(2e-3);
31 -        simulatedTime = simulatedTime + 2e-3;
32          %now check through output of Java block in model, whether amplitude and period are constar
33 -        amplitude_constant = getOutput('JAVA_FUNCTION.5','1');
34 -        period_constant = getOutput('JAVA_FUNCTION.5','3');
35 -        if (amplitude_constant == 1 && period_constant == 1) %constant oscillation, can calculate
36
37 -            fprintf(1,'Loop output (Vout) oscillating at constant amplitude and frequency.\n');
38 -            Ku = getParameter('PI.1','r0'); %get ultimate gain
39 -            Tu = getOutput('JAVA_FUNCTION.5','2'); %get ultimate period
40 -            fprintf(1,'Ultimate gain is Ku = %f\n',Ku);
41 -            fprintf(1,'Ultimate period is Tu = %f\n',Tu);
42 -            finished = 1;
43 -        else %not constant oscillation, increase Kp by 0.1, go on
44
45 -            Kp = Kp + 0.1;
46 -            fprintf(1,'Loop not oscillating at constant amplitude and frequency.\n');
47 -            setParameter('PI.1','r0',Kp);
48 -        end
49 -  └ end
```

The argument is the length of time we wish to simulate. This function continues from where the simulation last left off (or starts from zero, if it has not started) and simulates for the specified time. In this case (line 30) we chose 2 milliseconds, giving the system plenty of time to settle before analysing it. We also keep track (line 31) of how much we've simulated so far. The rest of the loop is the same as in the previous example.

Now take a look at what happens after the loop (lines 51-55). First, we must check whether we have simulated up to the specified end time, and if not, simulate the remainder (lines 51-53).



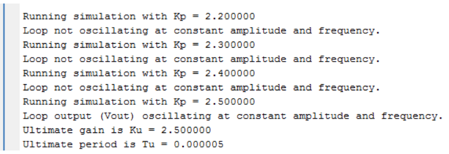
```
51 -     if (simulatedTime ~= tEnd)
52 -         simulateTime(tEnd - simulatedTime)
53 -     end
54
55 -     endSimulation();
```

Then we must tell GeckoCIRCUITS that the simulation is finished. We do this in line 55 by calling endSimulation().

The rest of the script is the same as before within the Java version.

Note that instead of using simulateTime, you can use simulateSteps, and specify the exact number of steps to simulate rather than the time to simulate.

Now execute the m-file. You should see the same output as before, but now in the MATLAB console. Check again the simulation results, and you will see that the coefficients have been found at that the converter regulates at 6 V.



```
Running simulation with Kp = 2.200000
Loop not oscillating at constant amplitude and frequency.
Running simulation with Kp = 2.300000
Loop not oscillating at constant amplitude and frequency.
Running simulation with Kp = 2.400000
Loop not oscillating at constant amplitude and frequency.
Running simulation with Kp = 2.500000
Loop output (Vout) oscillating at constant amplitude and frequency.
Ultimate gain is Ku = 2.500000
Ultimate period is Tu = 0.000005
```

Controlling GeckoCIRCUITS in MATLAB using the GeckoSCRIPT interface
simulateTime()

In this tutorial, a step-by-step introduction was given on how to control a GeckoCIRCUITS model and simulation via a custom-made script using special functions provided by GeckoCIRCUITS for that purpose, both using the GeckoSCRIPT tool within GeckoCIRCUITS and through MATLAB. The tutorial demonstrated how to tune the PID coefficients for a buck converter's controller using a simple script written by the user.

This tutorial example gives a good starting point to the reader who wants to implement sophisticated scripts for simulation control for the purpose of performing parameter sweeps, optimizations, and so forth.

Further examples demonstrating more advanced and practical uses of GeckoSCRIPT functions will be periodically made available via the Gecko Newsletter.

Contact Information / Feedback

Andreas Müsing

Gecko-Research GmbH
Riedtlistrasse 72
CH-8006 Zürich, Switzerland

Phone +41-44-632 4265
Fax +41-44-632 1212

www.gecko-research.com
andreas.muesing@gecko-research.com
Document version: July 2012

APPENDIX

Switching from GeckoExternal to GeckoRemote for MATLAB Use

Prior to GeckoCIRCUITS version 1.6, the package gecko.GeckoExternal was used for controlling GeckoCIRCUITS from MATLAB.

As of version 1.6, this API is deprecated – please do not use it anymore. Switch to using gecko.GeckoRemote instead.

GeckoExternal remains usable for purposes of backwards compatibility. using it, it will display a message that it is deprecated and recommend moving to GeckoRemote, but otherwise it will work as before. Therefore your old MATLAB scripts, written before version 1.6, are still usable as they are.

However, GeckoExternal is no longer maintained or supported. Therefore you are strongly urged to switch to using GeckoRemote in your MATLAB scripts.

The main reason for the switch is that GeckoRemote enables GeckoCIRCUITS to run in your operating system's Java Virtual Machine (JVM) rather than MATLAB's JVM.

This approach makes it easier for you to work with GeckoCIRCUITS from MATLAB.

The main advantages are that

1) You can now close GeckoCIRCUITS, even if when started from MATLAB, without having to close all of MATLAB;

2) You don't have to worry about memory allocation for Java inside MATLAB;

3) You will not have problems with compiling your simulation models' Java blocks when running GeckoCIRCUITS from MATLAB. This was a serious problem experienced by many users.

From a user's point of view, the changes are minor and easy to make. In your old scripts, simply change the line
import('gecko.GeckoExternal.*')
to
import('gecko.GeckoRemote.*')
and everything else should work without modifications. Please note that now you must be mindful of port numbers in some cases – please read the section of this tutorial dealing with setting up the GeckoCIRCUITS-MATLAB link.