NL5 DLL User's Manual

Ver.3.12

VERSION

NL5 DLL User's Manual version 3.12.19, 04/24/2024
The latest version of User's Manual can be found at sidelinesoft.com/nl5.

LIMITED LIABILITY

NL5 DLL, together will all accompanying materials, is provided on a "as is" basis, without warranty of any kind. The author makes no warranty, either expressed, implied, or stationary, including but not limited to any implied warranties of merchantability or fitness for any purpose. In no event will the author be liable to anyone for direct, incidental or consequential damages or losses arising from use or inability to use NL5 DLL.

COPYRIGHTS

sidelinesoft

Table of Contents

I. Introduction

What is NL5 DLL

NL5 DLL is a 64-bit dynamic-linked library available for **Windows**, **Linux**, and **macOS**. It is included in the NL5 Circuit Simulator package. NL5 DLL performs transient and AC simulation of circuits created by NL5 Circuit Simulator, provides raw simulation data, allows modification of circuit parameters, adding data traces, and some other operations through DLL API functions. It can be used as an analog simulator which is started and controlled from other applications and tools (MATLAB, Python, custom C/C++ code), and as an analog co-simulation tool working with digital simulation tools (for example SystemVerilog simulators through DPI interface).

NL5 DLL users are supposed to be familiar with NL5 Circuit Simulator principle and operation. Please refer to NL5 Manual and NL5 Reference for information.

Please use public resources or specific documentation for general information about dynamic-linked libraries, SystemVerilog, and digital simulation tools.

Version

Current released Version and Revision of NL5 DLL is always the same as Version and Revision of NL5 Circuit Simulator. This guarantees full compatibility in terms of components, models, features, and performance. However, there is nothing wrong in using different Versions/Revisions of DLL and NL5.

Current build of DLL can be different from NL5, due to possible DLL and NL5 specific fixes and modifications.

NL5 DLL is distributed as part of NL5 complete package, which can be found at [sidelinesoft.com/nl5.](sidelinesoft.com/nl5)

NL5 DLL Ver.3 can open and simulate schematics created by NL5 Ver.2. When saved back into the file, a schematic will be automatically converted to Ver.3 format and cannot be opened by NL5 Ver.2 anymore.

Files

The following files are distributed to customers:

- `nl5_dll.h`
- `nl5_dll.lib`
- `nl5_dll.dll` – Windows
- `nl5_dll.so` – Linux
- `nl5_dll.dylib` – macOS (x64 and arm64)
- `MATLAB/` – demo files for MATLAB
- `Python/` – demo files for Python
- `SystemVerilog/` - supporting files for SystemVerilog
- `SystemVerilog/Vivado/` - supporting files for co-simulation with Vivado

License

Without a license, NL5 DLL operates as a **Demo version**. Demo version has all full function features available, however the total number of components in the schematic is limited to **20**. For unlimited number of components, NL5 DLL should use **NL5 License**.

II. Using DLL

Functions

Function parameters

The following parameter types are used in DLL functions:

`int` - 32-bit integer
`double` - 8-byte floating point
`char*` - pointer to null-terminated ASCII (1-byte) character string (character array)

Some functions return `double` values through pointers to `double` variable (`double*`) provided as a parameter of the function.

Function result

Most of DLL functions return integer value: function result. If function result is negative, it is an error code. Only error code -1 is currently used, however more error codes may be added in the future. It is not recommended to continue DLL execution if error code was received, since it may result in DLL crash.

If error code is returned, text description of the error can be obtained by `NL5_GetError` function:

```
if(NL5_GetValue(ncir, "R1.R", &value) < 0)
{
    printf("%s", NL5_GetError());
}
```

In case of successful execution, some functions return 0, and some functions return non-negative integer value, with the meaning depending on the function. For example, `NL5_Open` returns integer value: circuit handle, `NL5_GetText` returns number of characters placed into the character array, etc.:

```
int ncir = NL5_Open("rc.nl5");
if(ncir < 0)
{
    printf("%s", NL5_GetError());
}
```

Functions `NL5_GetInfo` and `NL5_GetError` return pointer to null-terminated ASCII character string:

```
char* str = NL5_GetInfo();
printf("%s", str);
```

The content of that string is valid only until execution of the next DLL function: then it will be changed. If the text requested by calling those functions is needed for the future use, it is user's responsibility to copy it to safe location.

Handles

Handle is an index of the object in the internal DLL objects list. Handle is non-negative integer value. Some functions return handle as a function result. The handle referring to a specific object can be used as a parameter for other functions, related to that object. Handles are used for circuits, component parameters, inputs/outputs, and traces.

For example, function result of function `NL5_Open` is circuit handle. Once received, the handle can be used as an `ncir` parameter for many other functions, such as `NL5_Simulate`, `NL5_GetValue`, `NL5_GetParam`, `NL5_GetTrace`, etc.:

```
int ncir = NL5_Open("rc.nl5");
if(ncir < 0)
{
    printf("%s", NL5_GetError());
}

double r;
if(NL5_GetValue(ncir, "R1.R", &r) < 0)
{
    printf("%s", NL5_GetError());
}
```

Using DLL

Error message

A general function which may be called after calling practically any other function is `NL5_GetError`. It returns text description of the error which might occur while executing previous function, or `"OK"` if execution was successful:

```
if(NL5_GetValue(ncir, "R1.R", &value) < 0)
{
    printf("%s", NL5_GetError());
}
```

DLL information

A function you might want to call at DLL startup is `NL5_GetInfo`. It returns information about DLL: version and date:

```
char* str = NL5_GetInfo();
printf("%s", str);
```

This information is useful for troubleshooting, so please provide it when submitting bug reports or other requests.

License

If you have NL5 License with DLL option, call `NL5_GetLicense` function before performing simulation. Specify path of the license nl5.nll file as a parameter of the function. Call `NL5_GetError` right after that to obtain License ID or text description of the error:

```
int err = NL5_GetLicense("C://Projects/nl5/nl5.nll");
printf("%d, %s", err, NL5_GetError());
```

Error code and text description of the error are useful for troubleshooting, so please provide it when submitting bug reports or other requests.

Another way to use the license is placing the license file into the same folder as schematic file to be simulated. If `NL5_GetLicense` function was not called, then `NL5_Open` function will automatically try to find and check the license.

Schematic

To perform simulation, a schematic should be loaded into the DLL from a schematic "`*.nl5`" file. Once loaded, the schematic is stored in the DLL memory, and can be used for simulation. During simulation, the circuit component parameters can be modified by DLL, and simulated data will be saved as a traces. A modified schematic with simulation data can be saved back into the schematic file.

To load schematic into DLL use `NL5_Open` function. If file name does not have a path, DLL will look for a file in the directory where DLL is located. The function returns non-negative circuit handle `ncir`, which will be used in other DLL functions to identify the circuit:

```
int ncir = NL5_Open("rc.nl5");
if(ncir < 0)
{
    printf("%s", NL5_GetError());
}
```

If schematic file could not be loaded for any reason, a negative error code is returned. Also, an error occurs if requested file consists of too many components (currently 10) and is not DLL-enabled. Call `NL5_GetError` function to get text description of the error.

You can load several circuits at once by calling `NL5_Open`: a unique circuit handle will be returned for each circuit. If circuit is not needed anymore, it can be closed by `NL5_Close` function, however closing the circuit is not required.

The circuit can be saved back to the same schematic file by calling `NL5_Save`, or to a new file by calling `NL5_SaveAs` functions:

```
int ncir = NL5_Open("rc.nl5");
NL5_SetValue(ncir, "R1.R", 123.456);
NL5_SaveAs(ncir, "rc_new.nl5");
NL5_Close(ncir);
```

Use these functions to save schematic back to the file if any modification of component parameters were made by DLL, IC (Initial Conditions) were saved, or if you want to save schematic with obtained simulation and post-processing data.

To save schematic with transient and/or AC data, load schematic file in NL5, go to Schematic/Settings/Save options, and enable **Save with transient data** and/or **Save with AC data** option.

Parameters

DLL functions can access and modify component parameters. Parameters can be modified before simulation is started, as well as between DLL simulation calls. This is similar to pausing NL5 simulation, changing the parameter, and continuing the simulation.

Please be aware that changing the parameter between DLL simulation calls will result in recalculating the system matrix and switching to a new linear range of simulation. If parameters are being changed often, it may affect simulation speed. To change the value of voltage or current source in a "continuous manner", use DLL **input functions** instead. Those functions will modify the value of the sources keeping the simulation in the same linear range, which results in much more efficient and fast simulation. Please note that source values defined and changed as an input will not be saved into schematic file by `NL5_Save` and `NL5_SaveAs` finctions.

To specify parameter name in the function, use component parameter name in the format <component>.<parameter> (`"R1.R"`, `"V1.V"`). See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names).

There are two methods to access component parameters:

1. Direct.
2. Through parameter handle.

Direct method is an easiest one, however not optimal in terms of performance. To get component parameter value, use `NL5_GetValue` function. It returns value into the variable of `double` type. The pointer to that variable is passed to the function as a parameter:

```
double value;
NL5_GetValue(ncir, "R1.R", &value);
```

See Reference for explanation on working with different parameter types.

To set parameter value, use function `NL5_SetValue`:

```
NL5_SetValue(ncir, "R1.R", 123.456);
```

To get/set parameter value represented as a text, use `NL5_GetText` and `NL5_SetText` functions. These functions are applicable to practically all parameter types, including numerical. If numerical parameter is defined as a formula, those functions will get/set text of the formula:

```
char str[100];

NL5_SetText(ncir, "V1.Slope", "Linear");

NL5_GetText(ncir, "V1.Slope", str, 100);
// returns str = "Linear"

NL5_GetText(ncir, "R1.R", str, 100);
// returns str = "1.23e-3"

NL5_SetText(ncir, "R2.R", "=R1.R*2");

NL5_GetText(ncir, "R2.R", str, 100);
```

```
// returns str = "=R1.R*2"
```

These function can also be used to access and modify component model by using <component>.model format:

```
NL5_GetText(ncir, "V1.model", str, 100);
// returns str = "Pulse"

NL5_SetText(ncir, "V1.model", "Sin");
```

Accessing parameters **through parameter handle** would be a better option if parameter is being accessed at least several times. Using that method improves performance by parsing parameter name and searching for required component and parameter only once while obtaining parameter handle.

Use `NL5_GetParam` function to obtain the parameter handle first:

```
int nparam = NL5_GetParam(ncir, "R1.R");
if(nparam < 0))
{
    printf("%s", NL5_GetError());
}
```

Then use the parameter handle in functions `NL5_GetParamValue`, `NL5_SetParamValue`, `NL5_GetParamText`, and `NL5_SetParamText`:

```
NL5_SetParamValue(ncir, nparam, 1.0);
. . .
double r;
NL5_GetParamValue(ncir, nparam, &r);
```

Traces

DLL will store simulation data for all traces specified in the schematic file. The data can be accessed through the trace handle, obtained by `NL5_GetTrace` function for transient trace, or `NL5_GetACTrace` function for AC trace:

```
int ntrace = NL5_GetTrace(ncir, "V(R1)");
if(ntrace < 0))
{
    printf("%s", NL5_GetError());
}
```

A new trace for transient simulation can be added using functions `NL5_AddVTrace`, `NL5_AddITrace`, `NL5_AddPTrace`, `NL5_AddVarTrace`, and `NL5_AddFuncTrace`. These functions return trace handle. In the following example, a trace with voltage across resistor R1 is added:

```
int ntrace = NL5_AddVTrace(ncir, "R1");
if(ntrace < 0))
{
    printf("%s", NL5_GetError());
}
```

To minimize memory consumption, or accelerate simulation, any trace can be deleted by `NL5_DeleteTrace` function:

```
NL5_DeleteTrace(ncir, ntrace);
```

Please note that DLL **does not calculate** traces of **Math** type. Those traces are calculated only when using GUI version of NL5.

A special trace of **Data** type can be used for post-processing (see **Data post-processing** section for details),

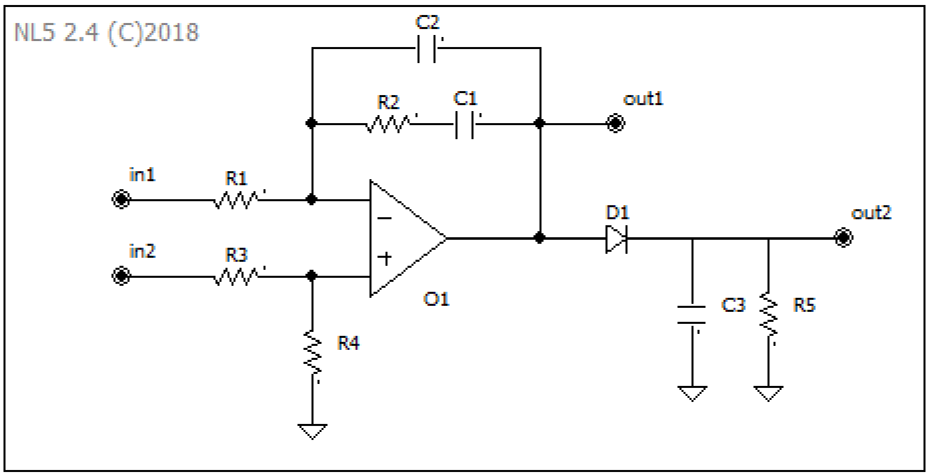
Co-simulation

NL5 DLL can be used for transient co-simulation with other tools, such as system-modeling, behavioral modeling tools, or digital simulators. DLL will provide fast and reliable simulation of analog part of the system. To provide better performance of co-simulation, the following system structure is suggested.

The analog circuit has constant voltage or current sources (Label, Voltage source, or Current source components) specified as **inputs**. The voltage or current value of those inputs are modified by the other tool before calling DLL simulation.

Also, the analog circuit has voltage or current meters (Label, Voltmeter, or Amperemeter) specified as **outputs**. When DLL simulation is completed for requested interval, the voltages/currents at specified outputs are transferred to the other tool as a result of analog simulation.

Here is an example of an analog part of the system, with two inputs (Labels `"in1", "in2"`) and two outputs (Labels `"out1", "out2"`):



Please note that input signals are modified in a "continuous" manner, keeping the simulation in the same linear range, thus providing fast simulation. However, any component parameters can be modified using parameter-based functions (for example `NL5_SetValue`) as well: this will result in recalculating the system matrix and switching to a new linear range of simulation.

If state of switch component needs to be modified, use voltage-controlled switch controlled by the input voltage source.

Please note that DLL will not store all simulated data at specified outputs: only last simulated data at the output is being stored until the next simulation call. However, DLL will still store data of all traces, specified in the circuit file, or added by calling DLL function. When the circuit is saved back into schematic file, the simulated data of those traces will be saved too, if "Save with transient data" option is set in the schematic file. To set the option, open schematic file in NL5, go to File/Properties/Save, select "Save with transient data" checkbox, and save schematic into the file.

Use **inputs/outputs** DLL functions to specify inputs and outputs for co-simulation.

Inputs/Outputs

Inputs/outputs can be accessed through the input/output handle.

Inputs. Call `NL5_GetInput` function to specify the input. 3 types of components can serve as an input:

- Label component;
- Voltage source component (V);
- Current source component (I).
-

Provide the label/component name as a parameter of the function. The function returns non-negative integer value: input handle:

```
int nin = NL5_GetInput(ncir, "in1");
if(nin < 0))
{
    printf("%s", NL5_GetError());
}
```

Use the handle and a desired source value to set input voltage/current by `NL5_SetInputValue` function:

```
int nin = NL5_GetInput(ncir, "in1");
. . .
NL5_SetInputValue(ncir, nin, 10.0);
```

Outputs. Call `NL5_GetOutput` function to specify the output. 3 types of components can serve as an output:

- Label component;
- Voltmeter (V);
- Amperemeter (A).

Provide the label/component name as a parameter of the function. The function returns non-negative integer value: output handle:

```
int nout = NL5_GetOutput(ncir, "out1");
if(nout < 0))
{
    printf("%s", NL5_GetError());
}
```

Use the handle and a pointer to the double variable to obtain output voltage by `NL5_GetOutputValue` function:

```
int nout = NL5_GetOutput(ncir, "out1");
. . .
double v;
NL5_GetOutputValue(ncir, nout, &v);
```

Typically, `NL5_SetInputValue` functions should be called for each specified input before calling DLL simulation function, and `NL5_GetOutputValue` functions should be called for each specified output after simulation function returns. However, those functions can be called any time. Input functions can be called only when input value, changed, and output functions can be called only when output value is needed.

Transient simulation

Transient simulation is performed with **simulation step** defined in the schematic file (see NL5 transient settings: Transient/Settings/Calculation step). If needed, the step can be modified any time by `NL5_SetStep` function:

```
double step = 1.0e-6;
NL5_SetStep(ncir, step);
```

To prevent DLL from being "stuck" due to erroneous code of C-code component (infinite while/do/for loop), or inability to resolve states of piece-wise linear components, a **simulation time-out** can be set up using function `NL5_SetTimeout`:

```
int time_out = 3;
NL5_GetTimeout (ncir, time_out);
```

If simulation time of one transient step exceeds the time-out value (in seconds), the simulation will stop with error message. Time-out equal to zero disables time-out detection.

DLL keeps track of current **simulation time** in the internal `simulation_time` variable. When simulation function is called, simulation is continued for requested interval starting from current `simulation_time`. Current `simulation_time` value can be obtained by `NL5_GetSimuationTime` function:

```
double current_time;
NL5_GetSimulationTime(ncir, &current_time);
```

To **start simulation**, call `NL5_Start` function. It resets `simulation_time` to 0, initializes circuit components, erases existing simulation data, and calculates initial state of the circuit according to specified Initial Conditions. This function should be called first to start simulation from t=0, prior to calling any simulation functions. When `NL5_Start` returns, the simulation data consists of circuit state at t=0. The simulation data at t=0 can be obtained by data-related functions described later.

However, calling `NL5_Start` is not required. It will be executed automatically if any of simulation functions is called, while simulation has not been started yet.

After simulation is started, there are three methods of performing simulation:

1. Simulate;
2. Simulate interval;
3. Simulate step.

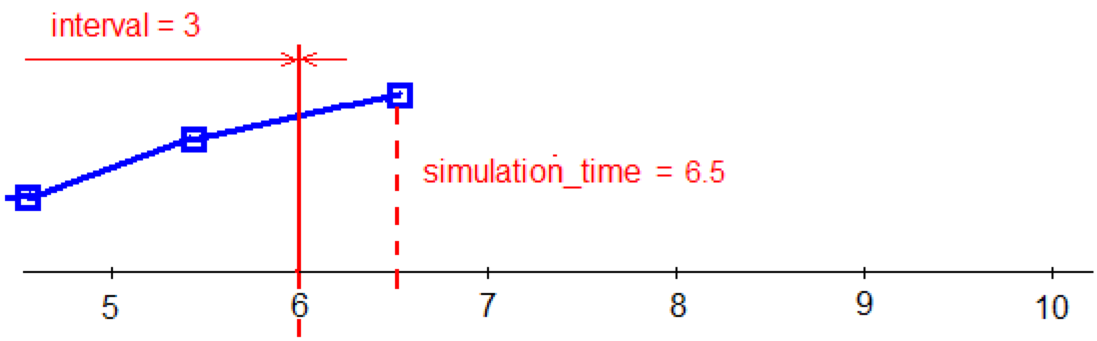
You can use just one method during all simulation, or any combination in any order.

Simulate method is performed by `NL5_Simulate` function, and it runs simulation for requested interval. The function does not change simulation step in order to stop exactly at the end of requested time, so the time of the last calculated data may exceed requested end time. When next simulation function is called, simulation will be continued with simulation step equal to the last simulation step.

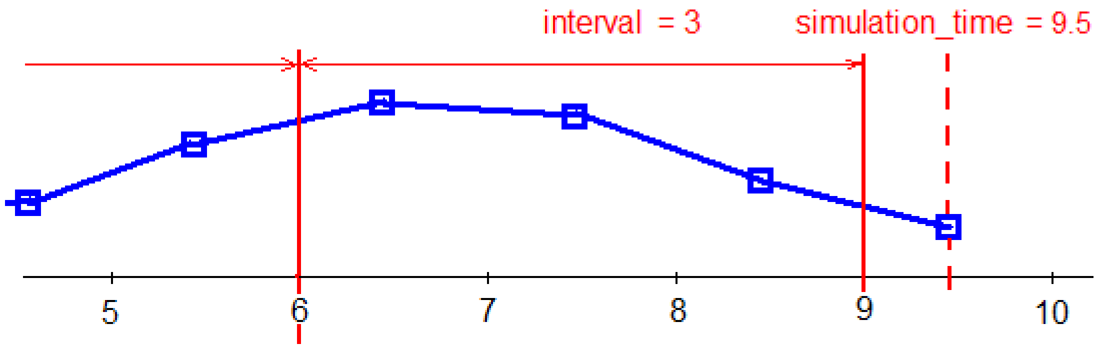
Here is an example of two consecutive calls of `NL5_Simulate` function. The first call was made at t = 3s (not shown on the graph), with interval = 3s:

```
NL5_Simulate(ncir, 3.0);
```

Due to selected simulation step = 1s, simulation stopped when the time of the last data point was 6.5s, which exceeded requested end time = 6s. At that moment, reported `simulation_time = 6.5s`:



When `NL5_Simulate` function with the same 3s interval is called again, simulation continues with the same simulation step = 1s, and stops at end time = 9s, with reported `simulation_time = 9.5s`:



Using `NL5_Simulate` function provides the best simulation performance. It won't decrease simulation step at the end of current linear range, so that there is no need to restore the step back as simulation continues. Thus, the simulation will be performed in a fastest manner, regardless of simulation interruptions.

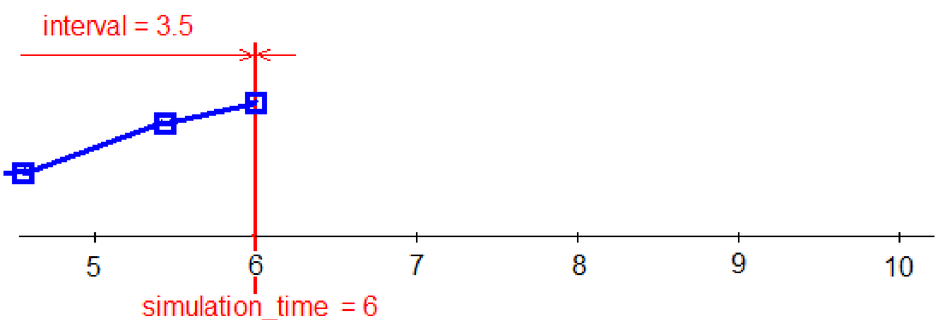
Simulate interval method is performed by `NL5_SimulateInterval` function, and it runs simulation **exactly** for requested interval. Unlike `NL5_Simulate`, it will adjust (decrease) simulation step if needed to stop exactly at the end of the requested interval. When next simulation function is called, simulation step will be restored, and a new linear range will be started.

Please note that if requested interval is smaller than simulation step, NL5 may not be able to decrease simulation step exactly as needed, and actual simulated interval might be longer than requested. To avoid that, it is recommended to use simulation step at least not greater than desired intervals.

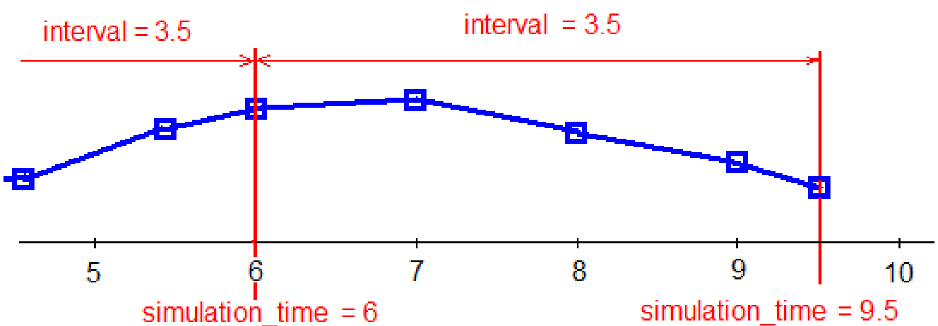
Here is an example of two consecutive calls of `NL5_SimulateInterval` function. The first call was made at t = 2.5s (not shown on the graph), with interval = 3.5s:

```
NL5_SimulateInterval(ncir, 3.5);
```

Simulation was performed with constant simulation step = 1s. Simulation stopped exactly at 6s, as requested. In order to do that, the last simulation step was decreased from 1s down to 0.5s:



When `NL5_SimulateInterval` is called again with requested interval = 3.5s, simulation step is restored back to 1s, and simulation continues:



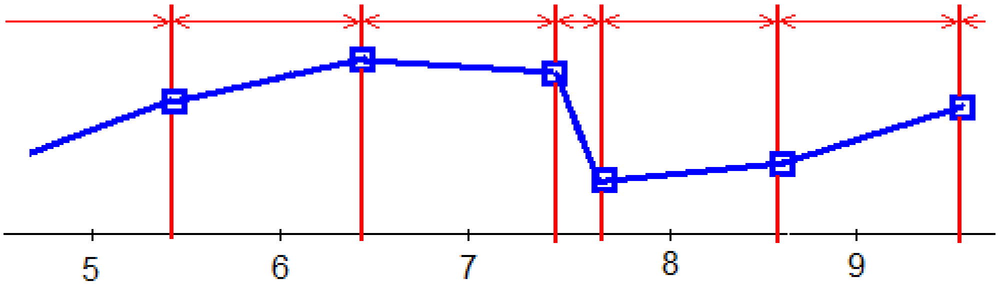
In this call, simulation step was also decreased at the end of the interval from 1s down to 0.5s, in order to stop exactly at 9.5s.

Due to possible change of simulation step even within the linear range, using of `NL5_SimulateInterval` may result in extremely slow simulation (especially if requested interval is small, and comparable with simulation step). Use this function only if it is really needed for your task.

Simulate step method is performed by `NL5_SimulateStep` function, and it executes just **one simulation step**. At the end, `simulation_time` is incremented by that simulation step, so that `simulation_time` is always equal to the time of last calculated data point.

Here is an example of simulation using `NL5_SimulateStep` function:

```
NL5_SimulateStep(ncir);
```



Please note that simulation step can be reduced by simulation algorithm if needed.

`NL5_SimulateStep` function can be used if DLL performs co-simulation with another simulation tool when it should continuously provide state of analog circuit with minimal possible time interval.

One more function related to simulation is `NL5_SaveIC`. Calling this function is similar to executing command Transient/Save IC in the NL5 Circuit Simulator. Current Initial Conditions are saved into components in the DLL memory. Use `NL5_Save` or `NL5_SaveAs` to save components with new Initial Conditions into the schematic file.

Simulation data

NL5 DLL saves all simulated data points into DLL memory. To obtain data of a specific trace, first obtain trace handle by calling `NL5_GetTrace` function:

```
int ntrace = NL5_GetTrace(ncir, "V(R1)");
if(ntrace < 0))
{
    printf("%s", NL5_GetError());
}
```

There are three ways to retrieve the data of the trace:

1. Read interpolated data;
2. Read data of a specific data point;
3. Read last data.

To read **interpolated data** at specific time, use `NL5_GetData` function with the time as a parameter, and pointer to `double` for amplitude of the data point:

```
double data;
NL5_GetData(ncir, ntrace, 1.234, &data);
```

Please be aware that interpolated data are calculated using linear interpolation, and may not accurately represent actual signals of the circuit between calculated data points.

To read the data of a **specific data point**, use `NL5_GetDataAt` function with index of the data point. Provide pointers to `double` variables for time and amplitude of the data point:

```
double t, data;
int index = 123;
NL5_GetDataAt(ncir, ntrace, index, &t, &data);
```

Data point index is zero-based: index of the first data point is 0, index of the last data point is equal to number of data points minus 1. Use `NL5_GetDataSize` function to obtain number of data points available for the trace:

```
int ndata = NL5_GetDataSize(ncir, ntrace);
if(ntrace < 0))
{
    printf("%s", NL5_GetError());
}
```

Please note that the number of data points can be different for different traces due to data compression.

To read **last data**, use `NL5_GetLastData` function with pointers to `double` variables for time and amplitude of the data point:

```
double t, data;
NL5_GetLastData(ncir, ntrace, &t, &data);
```

This function returns the data of last calculated data point.

As mentioned before, `NL5_Start` function erases all existing simulation data. Then, during simulation, all data points are being stored into DLL memory. There is a special algorithm in place to reduce the memory required for the data which are not changing (constant voltage/current supplies, output of digital components, etc.). However, if simulation is performed with small simulation step, the total available memory of the DLL can be easily exceeded.

If large amount of simulated data is expected, it is recommended to upload simulated data to your application or save into the file from time to time and delete that data from DLL memory by calling `NL5_DeleteOldData` function:

```
NL5_DeleteOldData(ncir);
```

This function does not erase all the data: it always leaves the very last calculated data point, or two data points, in order to be able to obtain interpolated data in the new interval.

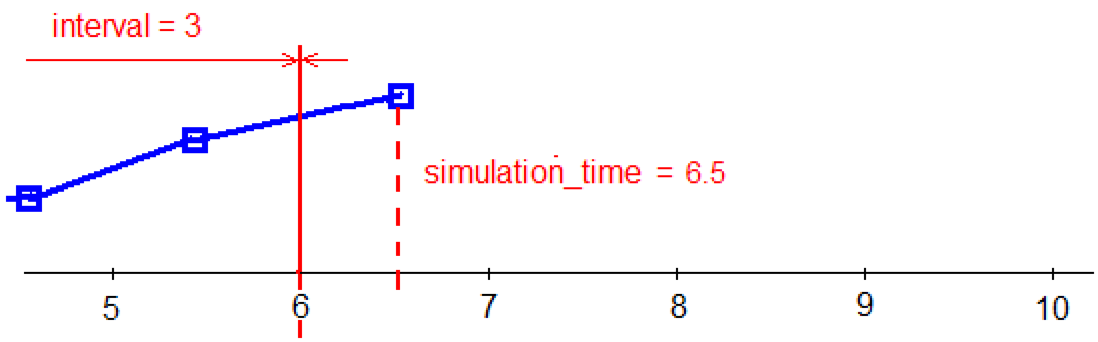
Simulation data can be saved into the file in the NL5 data format:

```
NL5_SaveData(ncir, "rc_data.nlt");
```

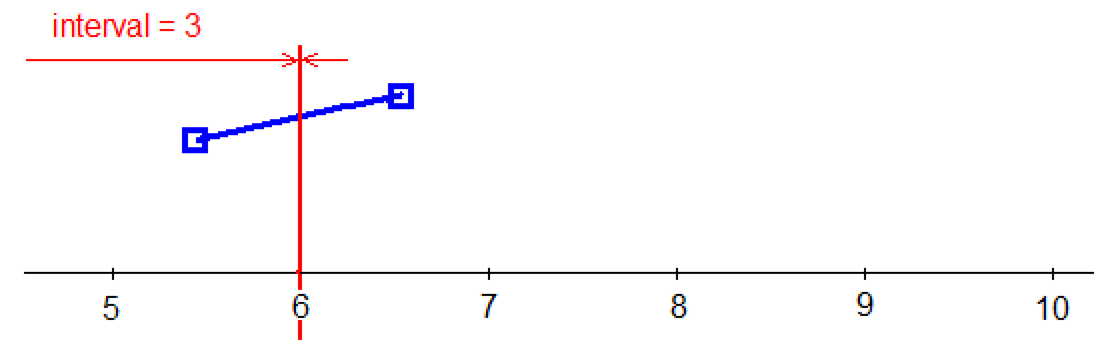
The data can be loaded into NL5 and shown on the transient graph.

Also, transient data will be saved in the schematic file if **Save with transient data** option of the schematic is enabled (Schematic/Settings/Save options in NL5).

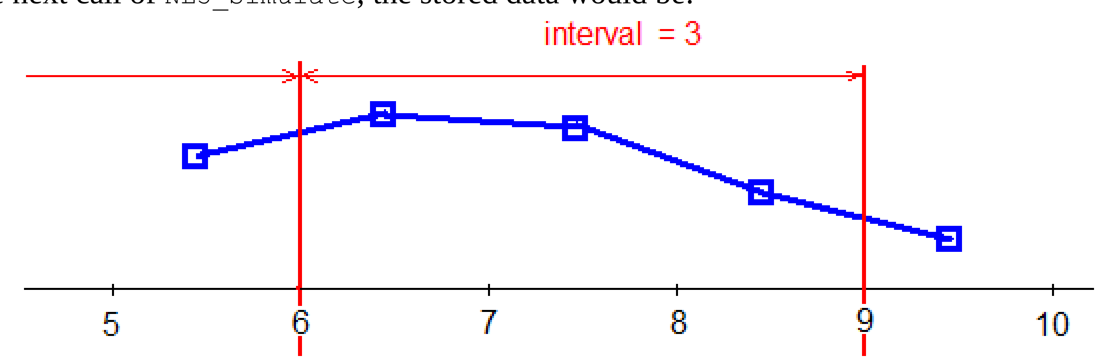
In the following example, simulation stopped after simulating two 3 second intervals using `NL5_Simulate` function, and final `simulation_time = 6.5s`:



When `NL5_DeleteOldData` function is called, it will erase old data, except last two points:



After the next call of `NL5_Simulate`, the stored data would be:



Now, all the data of the new calculated interval between t = 6s and t = 9s can be retrieved using interpolation.

Data post-processing

A special trace of **Data** type can be used for post-processing. Use `NL5_AddDataTrace` function to create the trace:

```
int ndata = NL5_AddDataTrace(ncir, "trace_name");
```

To add data to the trace, use `NL5_AddData` function. In this example, a new calculated trace is equal to squared `V(R1)` trace:

```
int nsource = NL5_GetTrace(ncir, "V(R1)");
int size = NL5_GetDataSize(ncir, nsource);

for(int i=0; i<size; ++i)
{
    double t, v;
    NL5_GetDataAt(ncir, nsource, i, &t, &v);
    NL5_AddData(ncir, ndata, t, v*v);
}
```

To delete current trace data (for example, before new simulation run) use `NL5_DeleteData` function:

```
NL5_DeleteData (ncir, ntrace);
```

The trace can be saved either to NL5 transient data file by `NL5_SaveData` function, or in the schematic file, if **Save with transient data** option of the schematic is enabled (Schematic/Settings/Save options in the NL5).

AC simulation

NL5 DLL performs AC simulation with simulation parameters specified in the schematic file, or defined by `NL5_SetAC` function:

```
NL5_SetAC(ncir, from, to, points, log_scale);
```

To set or change AC source, call `NL5_SetACSource` function with AC source component name as a parameter:

```
NL5_SetACSource(ncir, "V1");
```

Only "Linearize schematic" methos is currently supported. Call `NL5_CalcAC` function to run simulation:

```
NL5_CalcAC(ncir);
```

To obtain calculated AC data, first obtain trace handle:

```
int ntrace = NL5_GetACTrace(ncir, "V(C1)");
if(ntrace < 0))
{
    printf("%s", NL5_GetError());
}
```

Please note that AC traces cannot be added through NL5 DLL functions: they should be specified in the schematic file.

To read a specific data point, use `NL5_GetACDataAt` function with index of the data point. Provide pointers to `double` variables for frequency, magnitude, and phase of the data point:

```
double f, mag, phase;
int index = 123;
NL5_GetACDataAt(ncir, ntrace, index, &f, &mag, &phase);
```

Data point index is zero-based: index of the first data point is 0, index of the last data point is equal to number of data points minus 1. Use `NL5_GetACDataSize` function to obtain number of data points available for the trace:

```
int ndata = NL5_GetACDataSize(ncir, ntrace);
if(ntrace < 0))
{
    printf("%s", NL5_GetError());
}
```

Typically, all traces should have the same number of data points, however this may change in the future DLL versions.

AC data can be saved into the file in the NL5 data format:

```
NL5_SaveACData(ncir, "rc_data.nlf");
```

The data can be loaded into NL5 and shown on the AC graph.

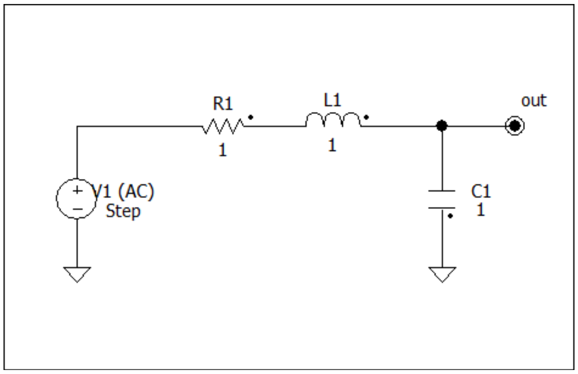
Also, AC data will be saved in the schematic file if **Save with AC data** option of the schematic is enabled (Schematic/Settings/Save options in NL5).

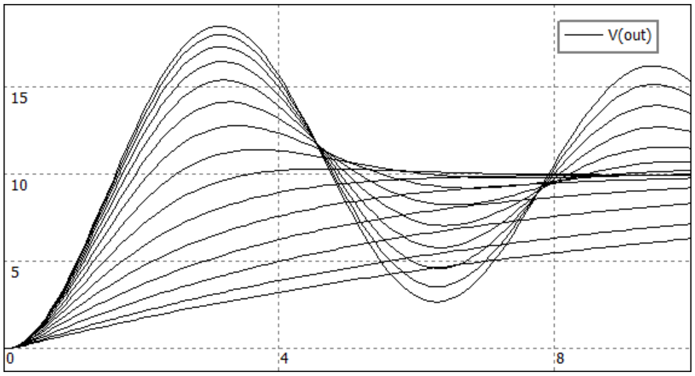
Using DLL with MATLAB

When using NL5 DLL with MATLAB, please use header file `nl5_dll.h` located in the MATLAB folder of NL5 DLL download package. Due to the way MATLAB is handling Windows DLLs, all `extern "C"` declarations must be removed from the header file.

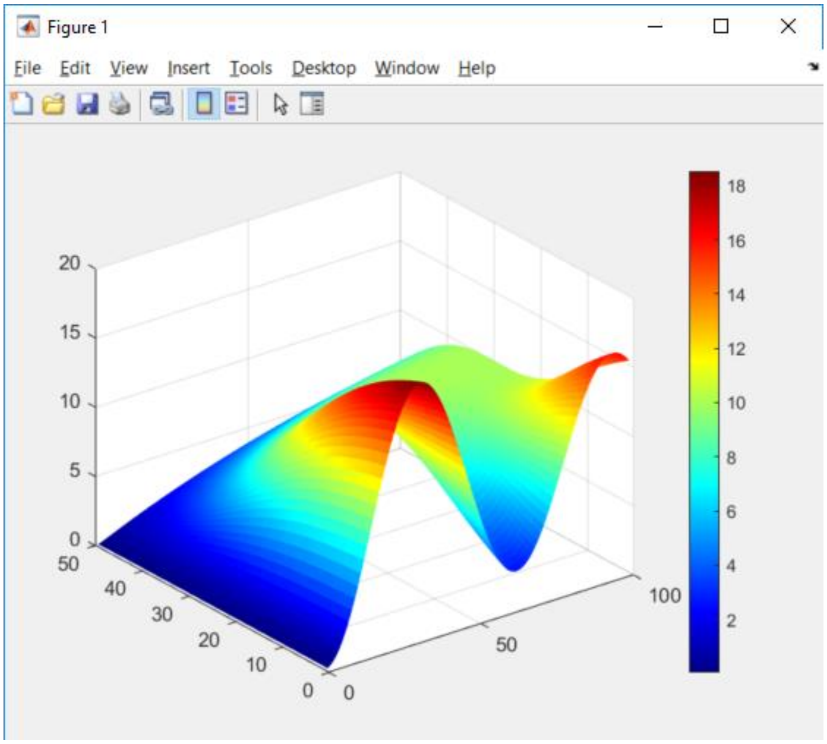
Simple examples of the MATLAB code `dll_example.m` can be found in the `MATLAB` folder of NL5 DLL download package. The first one, `dll_example.m`, opens schematic file `dll_example.nl5`, changes value of R1 in specified range, runs transient for each R1 value, reads transient data of trace `V(out)`, and displays results as a 3-D surface.

Here is schematic and results of transient simulation in NL5:





Here is a 3-D surface obtained in similar MATLAB simulation performed with NL5 DLL:



Here is MATLAB code:

```
clear;
clc;
close all;
R=logspace(-1,1,50);

% load library
loadlibrary('nl5_dll.dll', 'nl5_dll.h');

% open schematic
is = calllib('nl5_dll', 'NL5_Open', 'dll_example.nl5');
calllib('nl5_dll', 'NL5_GetError');

% get trace handle
it = calllib('nl5_dll', 'NL5_GetTrace', is, 'V(out)');

% create pointers to data
pd = libpointer('doublePtr', 0.0);

for k=1:50

    % set R1 value
      calllib('nl5_dll', 'NL5_SetValue', is, 'R1', R(k));

    % simulate for 10 s
    calllib('nl5_dll', 'NL5_Start', is);
    calllib('nl5_dll', 'NL5_Simulate', is, 10.0);

    % read data
      for i=1:100
             t = i*0.1;
             calllib('nl5_dll', 'NL5_GetData', is, it, t, pd);
             Z(k,i)=pd.value;
      end

end

% close document
calllib('nl5_dll', 'NL5_Close', is);
calllib('nl5_dll', 'NL5_GetError');

% unload library
unloadlibrary 'nl5_dll';

[X,Y] = meshgrid(1:100,1:50);
surf(X,Y,Z);
shading flat;
colormap jet;
colorbar;
ylim([0 50]);
```

Please note that you may need to change path of DLL and header file in function `loadlibrary`:

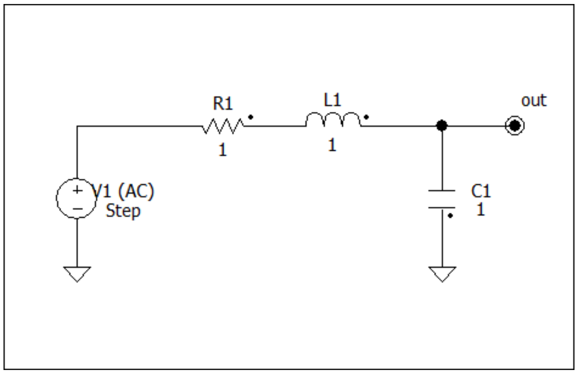
```
loadlibrary('Your_Path\\nl5_dll.dll', 'Your_Path\\nl5_dll.h');
```

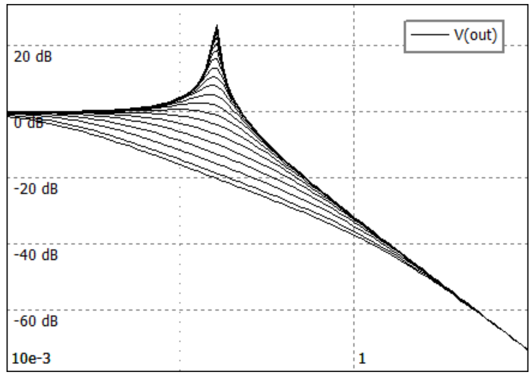
and path of the schematic file in the function `calllib` which calls DLL function `NL5_Open`:

```
is = calllib('nl5_dll', 'NL5_Open', 'Your_Path\\dll_example.nl5');
```

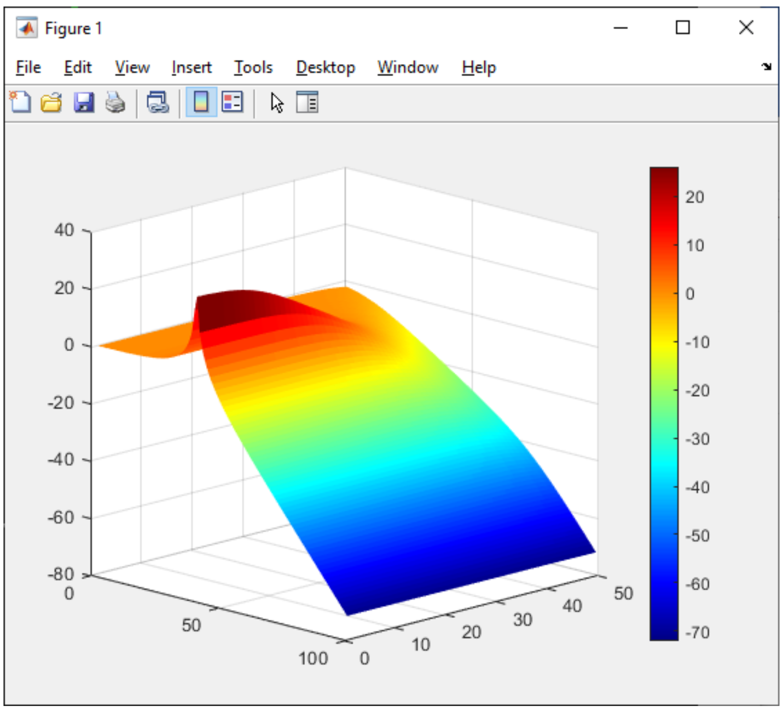
Another example code `dll_ac_example.m` performs AC analysis of the same circuit. It changes value of R1 in specified range, reads AC data of trace `V(out)`, and displays magnitude (in dB) as a 3-D surface.

Here is schematic, and results of AC simulation in NL5:





Here is a 3-D surface obtained in similar MATLAB simulation performed with NL5 DLL:



Here is the code:

```
clear;
clc;
close all;
R=logspace(-2,1,50);

% load library
loadlibrary('nl5_dll.dll', 'nl5_dll.h');

% open schematic
is = calllib('nl5_dll', 'NL5_Open', 'dll_example.nl5');
calllib('nl5_dll', 'NL5_GetError');

% get trace handle
it = calllib('nl5_dll', 'NL5_GetACTrace', is, 'V(out)');

% create pointers to data
freq = libpointer('doublePtr', 0.0);
mag = libpointer('doublePtr', 0.0);
phase = libpointer('doublePtr', 0.0);

for k=1:50

    % set R1 value
       calllib('nl5_dll', 'NL5_SetValue', is, 'R1', R(k));

    % simulate for 10 s
    calllib('nl5_dll', 'NL5_CalcAC', is);

    % read data
       for i=1:100
              calllib('nl5_dll', 'NL5_GetACDataAt', is, it, i, freq, mag, phase);
              Z(k,i)=20.0*log10(mag.value);
       end

end

% close document
calllib('nl5_dll', 'NL5_Close', is);
calllib('nl5_dll', 'NL5_GetError');

% unload library
unloadlibrary 'nl5_dll';

[X,Y] = meshgrid(1:100,1:50);
surf(X,Y,Z);
shading flat;
colormap jet;
colorbar;
ylim([0 50]);
view(45,15)
```

Using DLL with Python

When using NL5 DLL with Python, please use the Python package `nl5py` located in the `Python` folder of NL5 DLL download package. This package makes use of `ctypes`, which is a foreign function library for Python. The `ctypes` library provides C compatible data types, and it allows calling functions in dynamic linked libraries or shared libraries, such as provided with NL5 DLL.

Setup. The `nl5py` package includes a required initialization file, `__init__.py`. Prior to using `nl5py`, you will need to edit the `__init__.py` file to include the path to the appropriate library file for your system.

Windows library file is `nl5_dll.dll`. Edit the path variable as appropriate to point to the library:

```
path = Path(r'C:\path\to\your\library\nl5_dll.dll')
```

Note that for Windows, the lower case 'r' is necessary to ensure that the backslash ('\') is correctly interpreted. It is optional in the case of Linux or macOS.

Linux library file is `nl5_dll.so`. Edit the path variable as appropriate to point to the library:

```
path = Path(r'/path/to/your/library/nl5_dll.so')
```

macOS library file is `nl5_dll.dylib`. Edit the path variable as appropriate to point to the library:

```
path = Path(r'/path/to/your/library/nl5_dll.dylib')
```

Note that there are two different library files: for Intel processor (**x64**), and Silver processor (**arm64**).

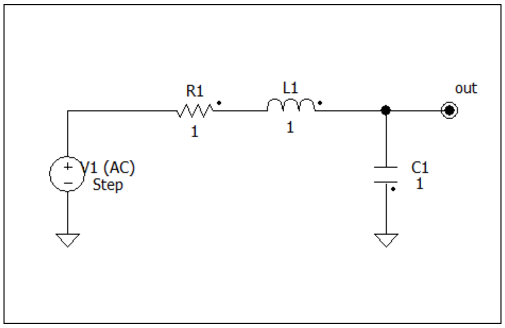
The `nl5py` package may be placed in any location in the file system pointed to by the environment variable `PYTHONPATH`. `PYTHONPATH` is used by Python to specify directories from which modules can be imported. Please consult online resources if you are unsure of how to set `PYTHONPATH`.

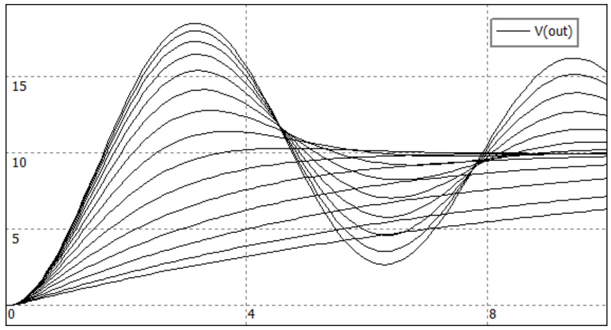
The example Python scripts assume that the schematic file `dll_example.nl5` is located in the working directory. If you place it somewhere else in the file system, be sure to specify the path correctly when you call `NL5_Open`.

Finally, the demo makes use of Python packages `numpy` and `matplotlib`. Please make sure that your Python distribution has these packages installed.

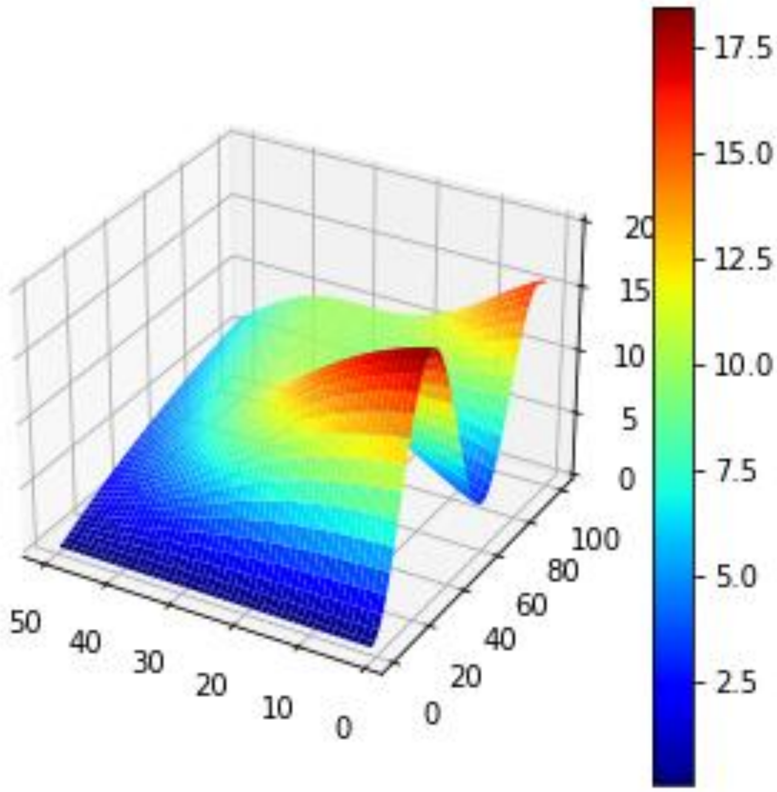
Demo. Simple examples of the Python code `dll_example.py` can be found in the `Python` folder of NL5 DLL download package. The first one, `dll_example.py`, opens schematic file `dll_example.nl5`, changes value of R1 in specified range, runs transient for each R1 value, reads transient data of trace `V(out)`, and displays results as a 3-D surface.

Here is schematic and results of transient simulation in NL5:





Here is a 3-D surface obtained in similar Python simulation performed with NL5 DLL:

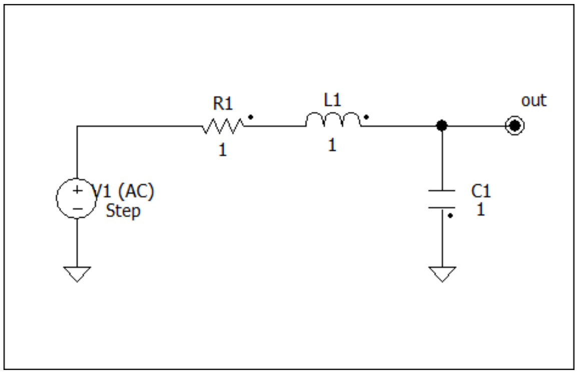


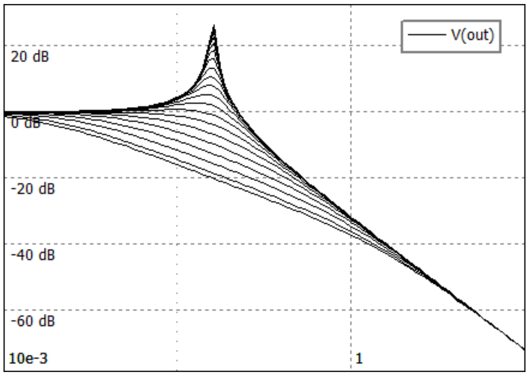
Here is the code:

```
# import required modules
import nl5py as nl5
import ctypes as ct
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# open schematic
ncir = nl5.NL5_Open(b'dll_example.nl5')

# create trace handle
ntrace = nl5.NL5_GetTrace(ncir, b'V(out)')

# create pointer to data
pd = ct.c_double()

# initialize
R = np.logspace(-1, 1, 50)
Z = np.zeros((50, 100))

for k in range(50):
    # set R1 value
    nl5.NL5_SetValue(ncir, b'R1.R', R[k])

    # simulate for 10s
    nl5.NL5_Start(ncir)
    nl5.NL5_Simulate(ncir, 10)

    # read data
    for i in range(100):
        t = i * 0.1
        nl5.NL5_GetData(ncir, ntrace, t, pd)
        Z[k, i] = pd.value

# close document
nl5.NL5_Close(ncir)
print(nl5.NL5_GetError())

# plot a 3D Surface
X = np.linspace(1, 100, 100)
Y = np.linspace(1, 50, 50)
Y, X = np.meshgrid(X, Y)

# formatting the figure
fig = plt.figure(figsize=(5, 5))
ax = fig.add_subplot(111, projection='3d')
ax.set_zlim(0, 20)
mycmap = plt.get_cmap('jet')
plt.gca().invert_xaxis()

# plotting the surface
surf = ax.plot_surface(X, Y, Z, cmap=mycmap)

# adding the colorbar
cb = plt.colorbar(surf)

plt.show()
```

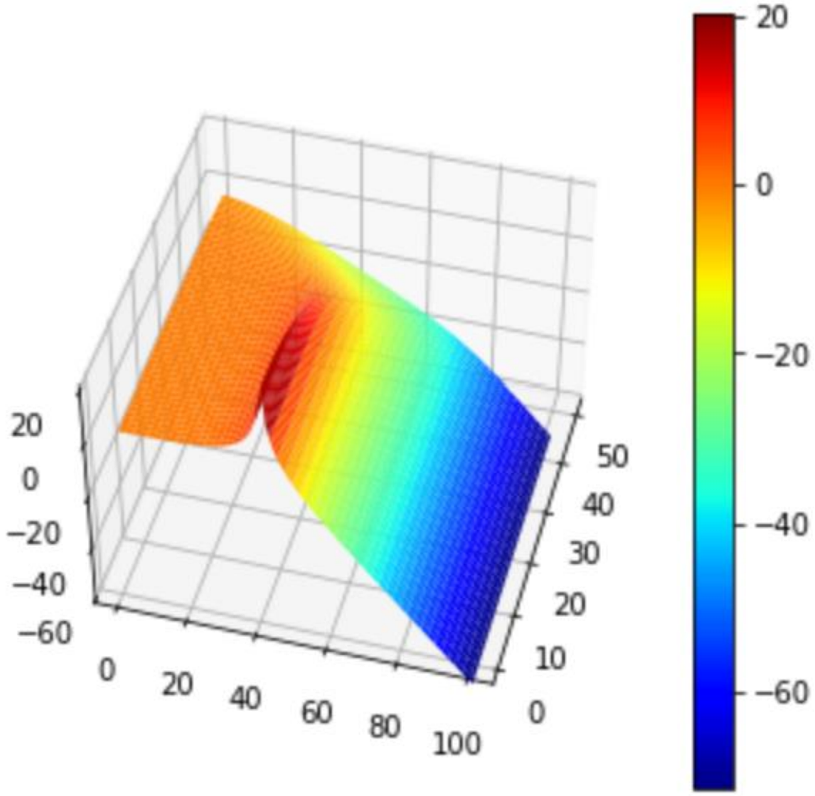
Another example code `dll_ac_example.py` performs AC analysis of the same circuit. It changes the value of R1 in specified range, reads AC data of trace `V(out)`, and displays magnitude (in dB) as a 3-D surface.

Here is schematic, and results of AC simulation in NL5:





Here is a 3-D surface obtained in similar Python simulation performed with NL5 DLL:



Here is the code:

```
# import required modules
import nl5py as nl5
import ctypes as ct
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# open schematic
ncir = nl5.NL5_Open(b'dll_example.nl5')

# create trace handle
ntrace = nl5.NL5_GetACTrace(ncir, b'V(out)')
ndata = nl5.NL5_GetACDataSize(ncir, ntrace)

# create pointer to data
freq = ct.c_double()
mag = ct.c_double()
phase = ct.c_double()

# initialize
R = np.logspace(-2, 1, 50)
Z = np.zeros((50, 100))

for k in range(50):
    # set R1 value
    nl5.NL5_SetValue(ncir, b'R1.R', R[k])

    # simulate AC
    nl5.NL5_CalcAC(ncir)

    # read data
    for t in range(100):
        nl5.NL5_GetACDataAt(ncir, ntrace, t, freq, mag, phase)
        Z[k, t] = 20.0*np.log10(mag.value)

# close document
nl5.NL5_Close(ncir)
print(nl5.NL5_GetError())

# plot a 3D Surface
X = np.linspace(1, 100, 100)
Y = np.linspace(1, 50, 50)
Y, X = np.meshgrid(X, Y)

# formatting the figure
fig = plt.figure(figsize=(5, 5))
ax = fig.add_subplot(111, projection='3d')
ax.set_zlim(-60, 20)
mycmap = plt.get_cmap('jet')
plt.gca().invert_xaxis()

# plotting the surface
surf = ax.plot_surface(X, Y, Z, cmap=mycmap)

# adding the colorbar
cb = plt.colorbar(surf)

ax.view_init(45, 15)

plt.show()
```

Using DLL with SystemVerilog

NL5 DLL can be used for co-simulation with SystemVerilog digital simulators, where DLL functions are being called through DPI – Direct Programming Interface.

Files

The following files can be used for interfacing DLL with SystemVerilog DPI:

- `nl5_dll.dll` (Windows)
- `nl5_dll.lib` (Windows)
- `nl5_dll.so` (Linux)
- `nl5_sv.svh` – header file for SystemVerilog code
- `nl5_sv.c` – "wrapper" C-code
- `svdpi.h` – header file for "wrapper" C-code

Using DLL

To use DLL with SystemVerilog code, link the project with appropriate DLL library file, and place appropriate NL5 DLL file into the directory where it can be accessed. Also, include `nl5_sv.svh` header file into Verilog code. This file contains prototypes of DLL functions.

Refer to the documentation of your SystemVerilog simulation tool for details on creating the project and using DPI.

Using DLL with C-code "wrapper"

If DLL library file cannot be linked to the SystemVerilog project for any reason, NL5 DLL can be accessed using provided "wrapper" C-code `nl5_sv.c`. Compile and link that code to the SystemVerilog project. Please note that different tools may require their own specific header file `svdpi.h`. Refer to the documentation of your SystemVerilog simulation tool for details on creating the project and using DPI.

Include `nl5_sv.svh` header file into SystemVerilog code: this file contains prototypes of DLL functions.

Place DLL file into the directory where it can be accessed. Before calling any DLL functions first time, DLL should be loaded into memory by calling `NL5_OpenDLL` function with appropriate dll file name as a parameter. The function returns 0 if successful, or negative error code if failed. The following error codes are currently used:

```
int result = NL5_OpenDLL("nl5_dll.dll");
if(result == -1)
{
    // DLL not found. Handle the error here
    . . .
}
else if(result == -2)
{
    // Some DLL functions not found. Handle the error here
    . . .
}
else if(result == -3)
{
    // DLL already loaded. Handle the error here
    . . .
}
else
{
    // OK
}
```

Once DLL is successfully loaded, all DLL functions can be called.

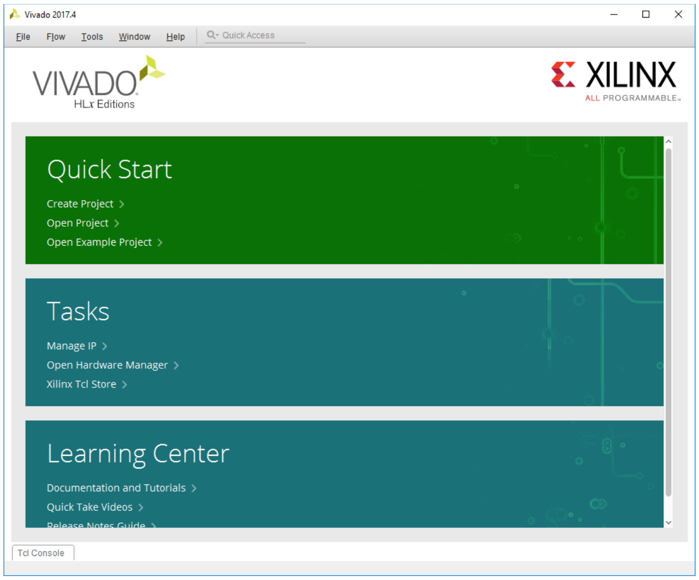
Running co-simulation demo with Xilinx Vivado

Creating demo project

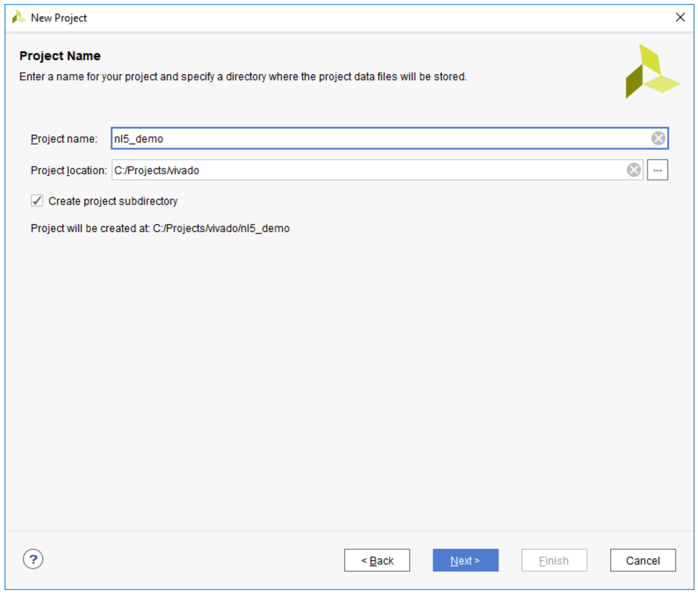
There are many ways of creating and configuring Vivado project. Please refer to Vivado Manual, or use public on-line tutorials on Vivado for more information.

For this instructions, Vivado HLx Edition, v2017.4 (64-bit) was used.

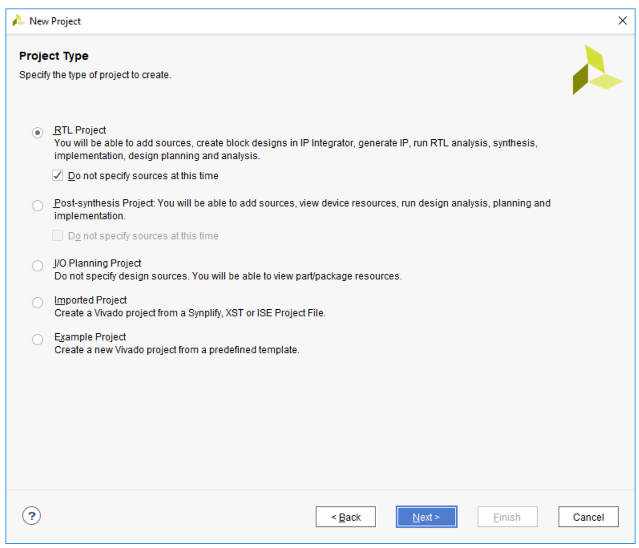
To create a new project, open Vivado:



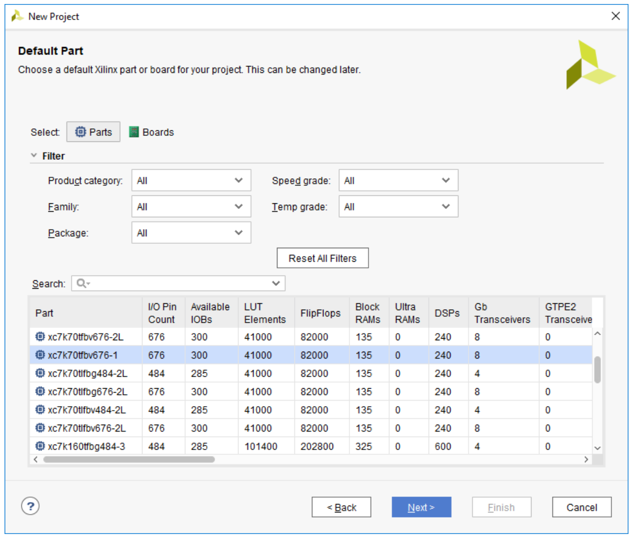
Select "Quick Start" / "Create Project", Click "Next"



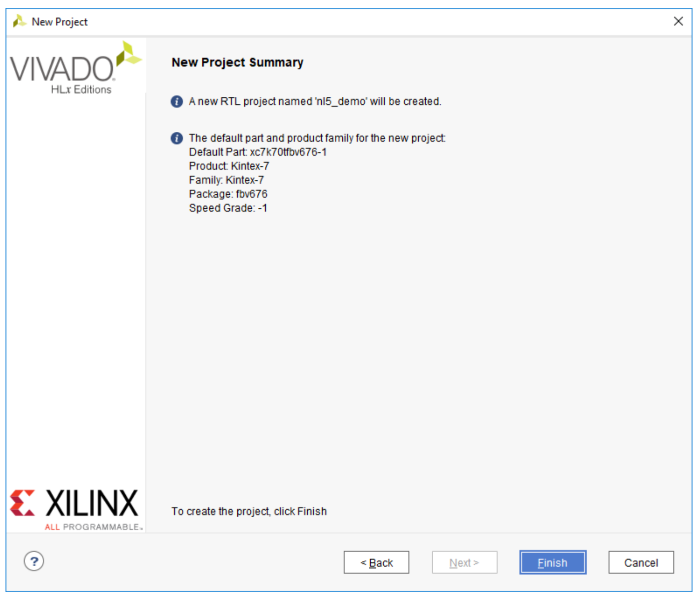
Project name: enter project name ("nl5_demo"), click "Next":



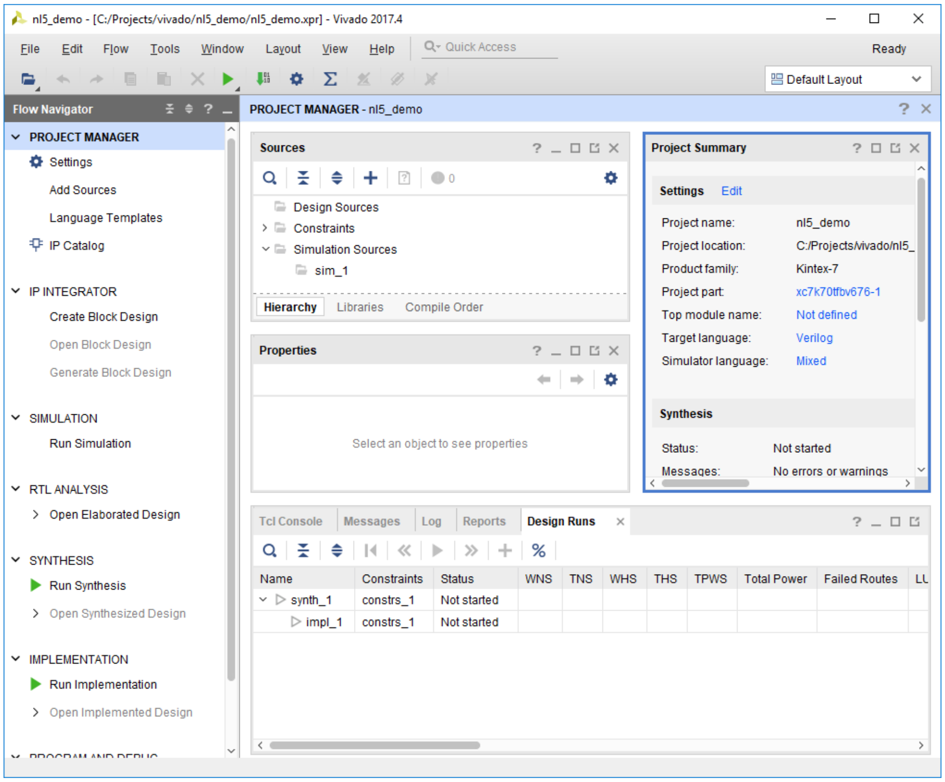
Project type: click "Next":



Default part: please note that list of parts will depend on your installation. Select Xilinx part or board, click "Next":



Click "Finish"



The project has been created; project directory is:

```
C:\Projects\vivado\nl5_demo
```

Creating library file

To create library file `dpi.a`, copy the following files from `SystemVerilog` directory of the NL5 DLL installation package to Vivado temporary directory
`C:\Users\<UserName>\AppData\Roaming\Xilinx\Vivado`

```
nl5_sv.c
svdpi.h
```

In the Vivado Tcl Console command line, type:

```
xsc nl5_sv.c
```

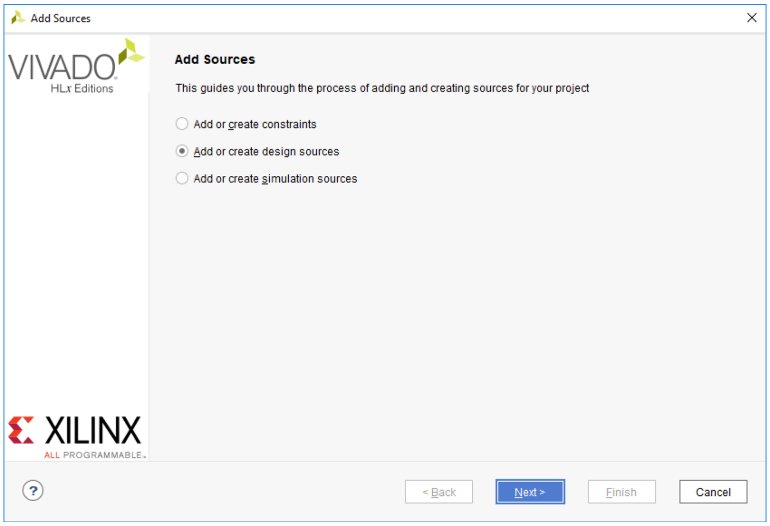
For running NL5 DLL demo, copy new `dpi.a` file from
`C:\Users\<UserName>\AppData\Roaming\Xilinx\Vivado`
to `C:\Projects\vivado\nl5_demo\nl5_demo.sim\sim_1\behav\xsim`
as described in the next section.

Configuring and running demo

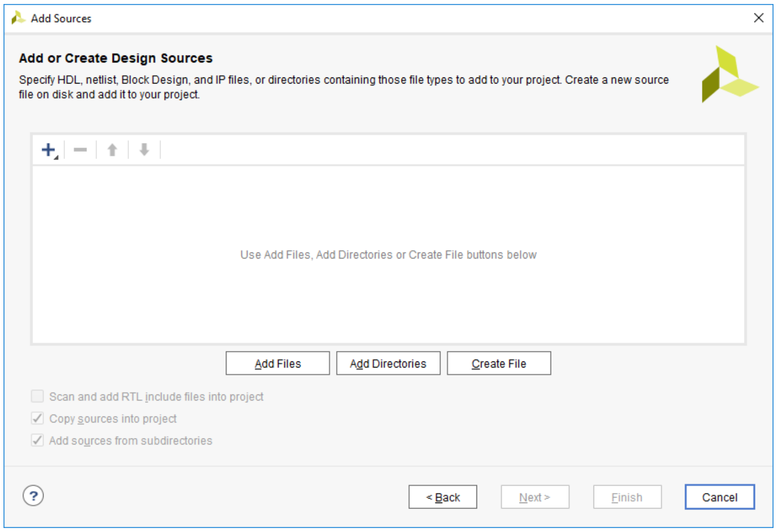
In the NL5 DLL installation package, go to `SystemVerilog\Vivado\src` directory, and copy the following files into project directory `C:\Projects\vivado\nl5_demo`

```
nl5_demo.sv
nl5_sv.svh
```

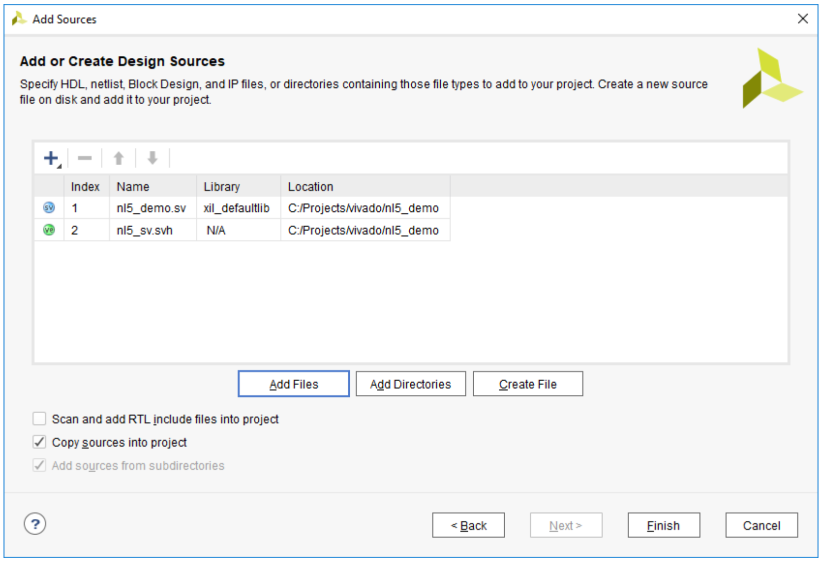
Select "Project manager" / "Add Sources":



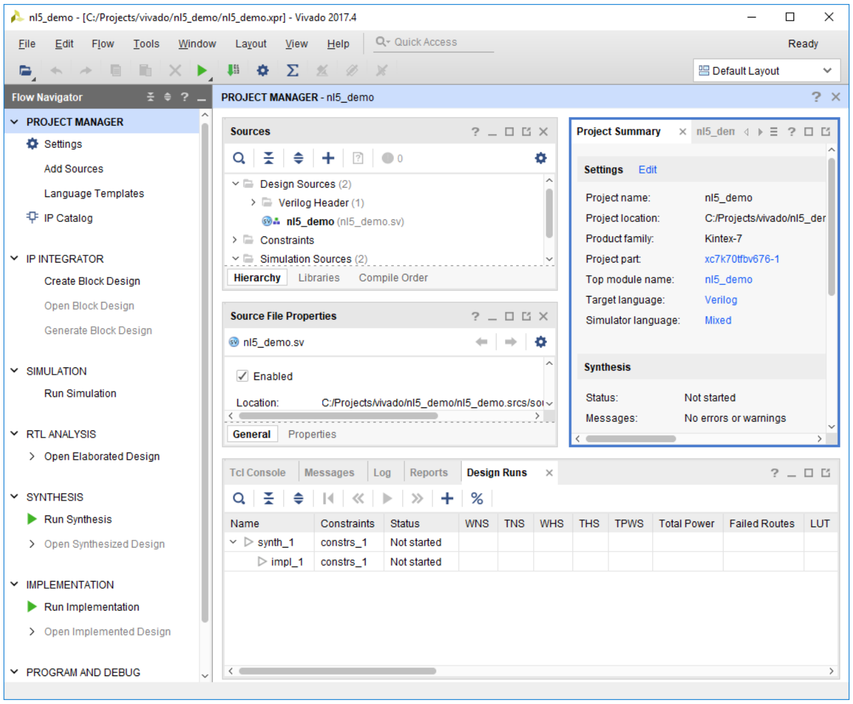
Add Sources: select "Add or create design source", click "Next":



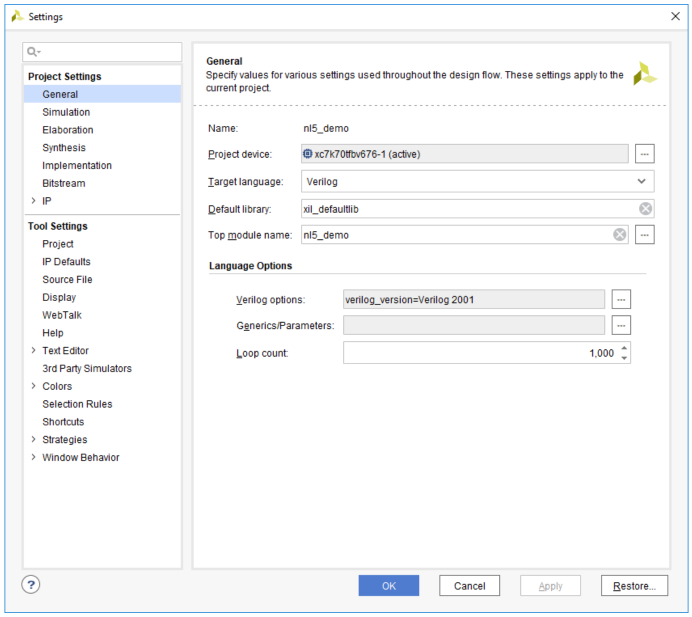
Add or Create Design Sources: click Add Files, select `C:\Projects\vivado\nl5_demo` directory, select `nl5_demo.sv` and `nl5_sv.svh` files (using Ctrl key), click "OK":



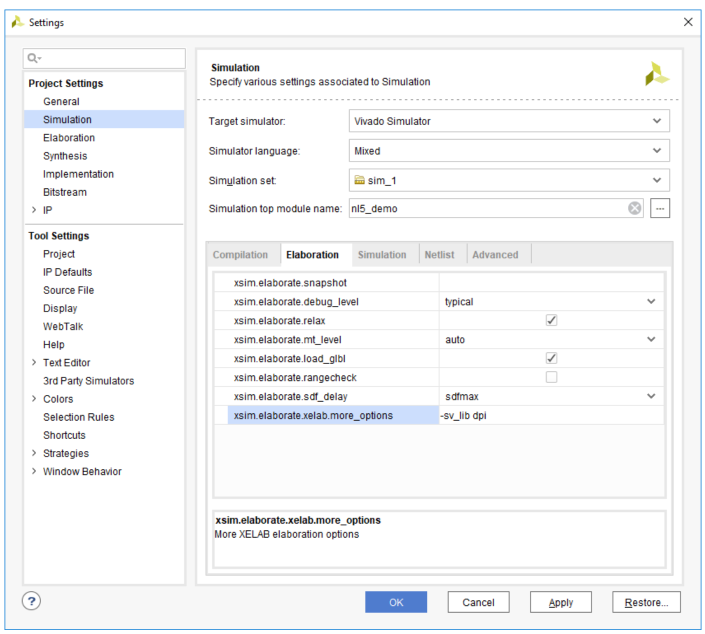
Click "Finish":



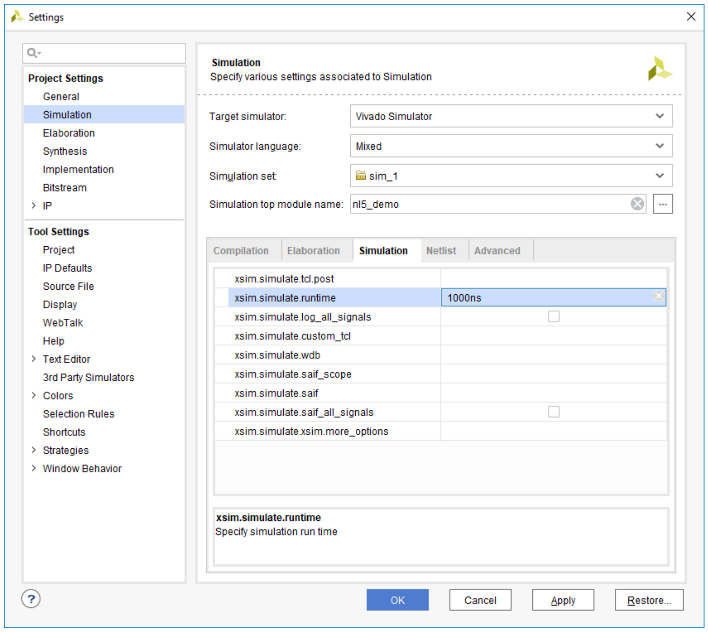
Select "Project manager" / "Settings":



Select "Project Settings" / "Simulation", "Elaboration" tab, enter:
xsim.elaborate.xelab.more_options = -sv_lib dpi

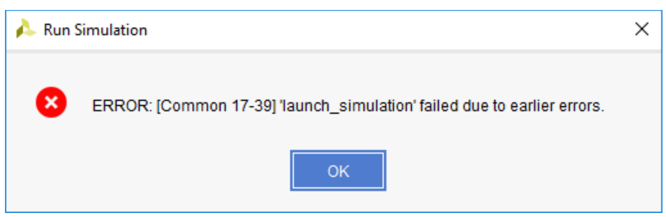


Select “Simulation” tab, enter:
xsim.simulation.runtime = 1000ns



Click “OK”

Select “Project Manager” / “Simulation” / “Run Simulation” / Run Behavioral Simulation”. An error message will pop up:



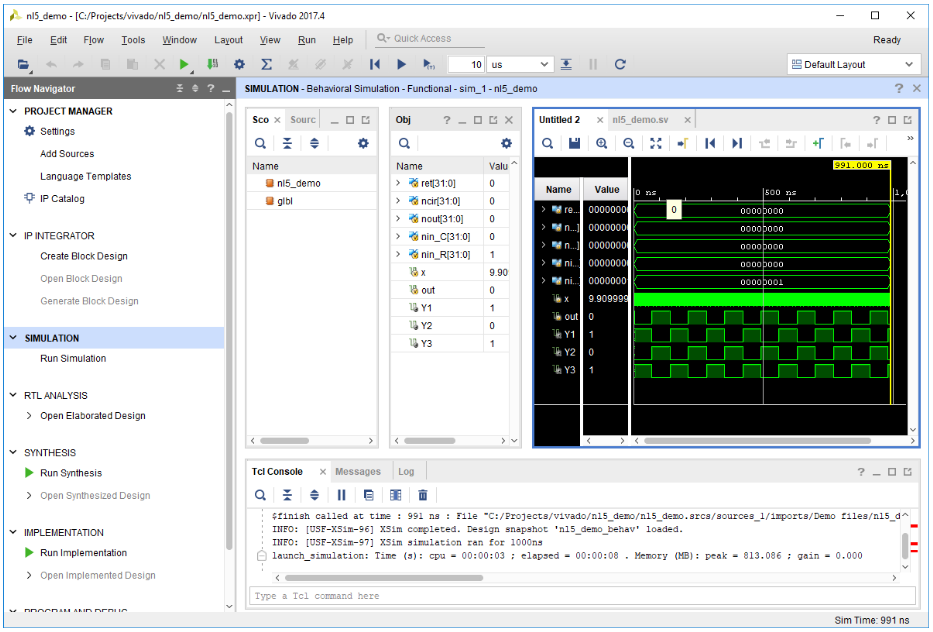
Click “OK” two times. This step is required in order to force Vivado to create simulation directory, and then copy required nl5 demo files into that directory.

In the NL5 DLL installation package, go to `SystemVerilog\Vivado\sim` directory, and copy the following files into simulation directory
`C:\Projects\vivado\nl5_demo\nl5_demo.sim\sim_1\behave\xsim`

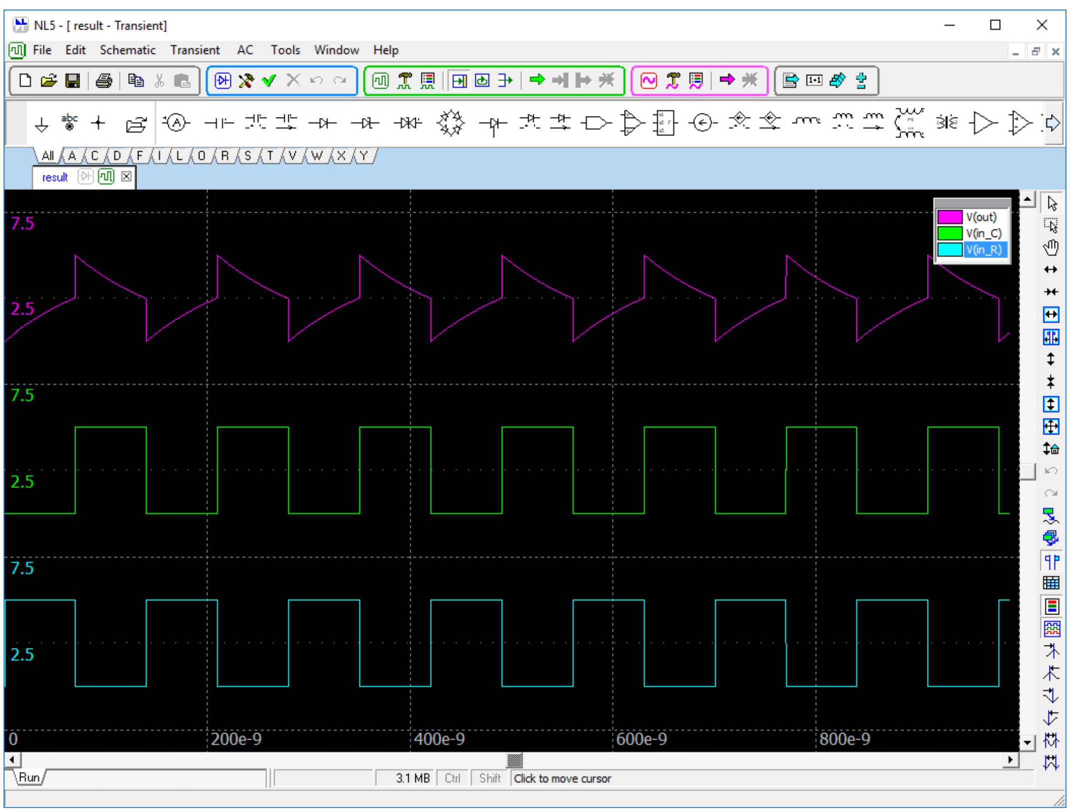
```
nl5_dll.dll
rc.nl5
```

Also, copy library file `dpi.a,` as described in ”Creating library file” section.

Select "Project Manager" / "Simulation" / "Run Simulation" / Run Behavioral Simulation".
After successful simulation, the results will be shown in the Waveform Window:

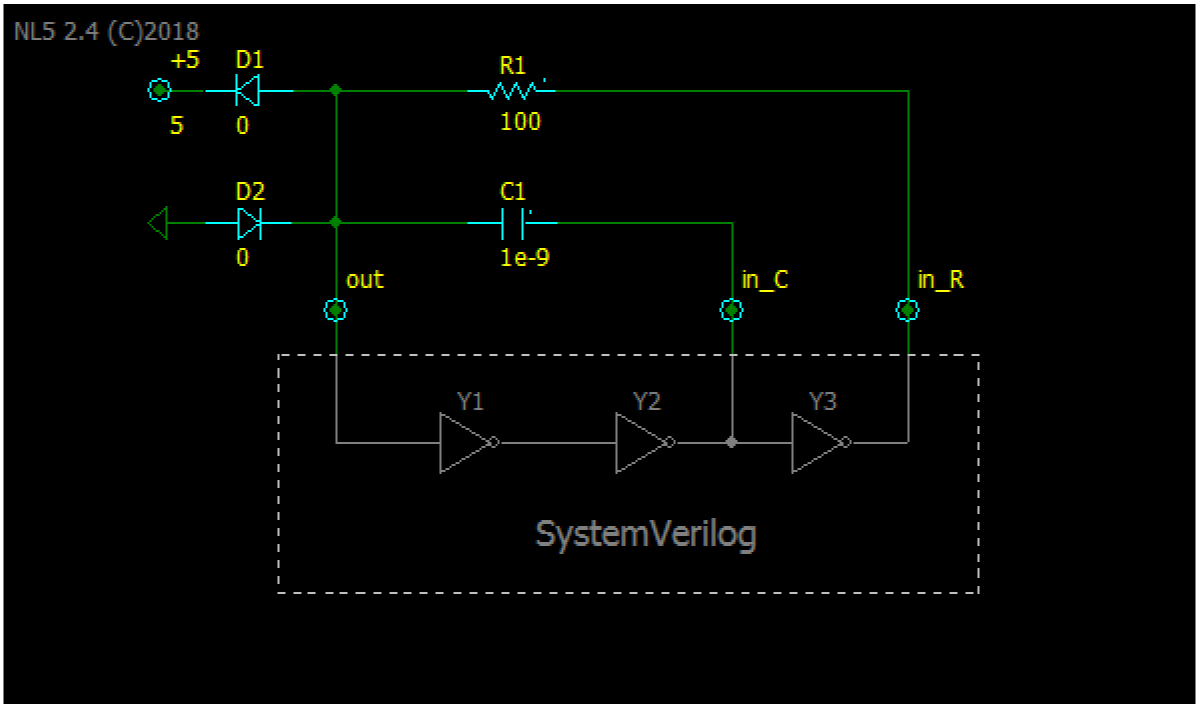


To see analog waveforms of the simulation, start NL5 Circuit Simulator, open nl5 file with simulation results
`C:\Projects\vivado\nl5_demo\nl5_demo.sim\sim_1\behav\xsim\result.nl5`
and open transient window:



Demo circuit

A simple oscillator circuit with 3 inverters is used as a demo:



Digital part (Y1, Y2, Y3) of the circuit is disabled, since it will be simulated by SystemVerilog. Labels "out", "in_C", and "in_R" are used for passing signals between analog and digital parts.

When SystemVerilog simulation is completed, the schematic is saved into the file `result.nl5` along with transient results. Start NL5 Circuit Simulator, and open `result.nl5` to see analog waveforms in details.

To run simulation with NL5 Circuit Simulator, enable digital part of the schematic, and run transient. To enable/disable schematic, select part of the schematic, right-click on the selection, select "Enable" or "Disable" from context menu.

III. DLL Functions

NL5_GetError

Prototype:

```
char* NL5_GetError()
```

Parameters:

No parameters

Returns:

Pointer to null-terminated ASCII character string

Description:

Returns text description of last execution error. If no error, returns "OK".

The content of the string is valid only until execution of the next DLL function. If the text is needed for the future use, it is user's responsibility to copy it to safe location.

NL5_GetInfo

Prototype:

```
char* NL5_GetInfo()
```

Parameters:

No parameters

Returns:

Pointer to null-terminated ASCII character string

Description

Returns information about DLL, such as version and date.

The content of the string is valid only until execution of the next DLL function. If the text is needed for the future use, it is user's responsibility to copy it to safe location.

NL5_GetLicense

Prototype:

```
int NL5_GetLicense(char* name)
```

Parameters:

`char* name` – pointer to null-terminated ASCII character string with NL5 license file name

Returns:

0 : valid license file with DLL license option found
<0 : error, or license does not have DLL option

Description

The function loads NL5 license file and checks if DLL license option is enabled. Call `NL5_GetError()` after calling `NL5_GetLicense()` to get License ID, or error message.

NL5_Open

Prototype:

```
int NL5_Open(char* name)
```

Parameters:

`char* name` – pointer to null-terminated ASCII character string with NL5 schematic file name

Returns:

>=0 : circuit handle
-1 : error

Description

Opens NL5 schematic file "`name`".

Returns non-negative circuit handle, or -1 if file not found, cannot be open for any reason, or file and is not DLL-enabled and contains too many components.

Circuit handle can be used as input parameter `ncir` for other DLL functions.

If file name does not have path specified, DLL will search for the file in the same directory where NL5 DLL is located.

NL5_Close

Prototype:

```
int NL5_Close(int ncir)
```

Parameters:

`int ncir` – circuit handle

Returns:

0 : OK
-1 : error

Description

Close schematic with handle `ncir`. Schematic information will be removed from DLL, handle `ncir` cannot be used anymore.

NL5_Save

Prototype:

```
int NL5_Save(int ncir)
```

Parameters:

int ncir – circuit handle

Returns:

0 : OK
-1 : error

Description

Save schematic with handle `ncir` into the same file.

Use this function to save schematic back to NL5 schematic file. You might want to save the schematic if any modification of component parameters were made, IC (Initial Conditions) were saved, or if you want to save schematic with transient data (simulation data traces).

To save schematic with transient data, make sure the "Save with transient data" option is set in the schematic file. To set the option, open schematic file in NL5, go to File/Properties/Save, select "Save with transient data" checkbox, and save schematic into the file.

NL5_SaveAs

Prototype:

```
int NL5_SaveAs(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with NL5 schematic file name

Returns:

0 : OK
-1 : error

Description

Save schematic with handle `ncir` into a new schematic file.

Use this function to save schematic into a new NL5 schematic file. You might want to save the schematic if any modification of component parameters were made, IC (Initial Conditions) were saved, or if you want to save schematic with transient data (simulation data traces).

To save schematic with transient data, make sure the "Save with transient data" option is set in the schematic file. To set the option, open schematic file in NL5, go to File/Properties/Save, select "Save with transient data" checkbox, and save schematic into the file.

NL5_GetValue

Prototype:

```
int NL5_GetValue(int ncir, char* name, double* v)
```

Parameters:

int ncir	–	circuit handle
*char** name	–	pointer to null-terminated ASCII character string with parameter name
*double** v	–	pointer to value variable

Returns:

0 : OK
-1 : error

Description

Returns `double` value of component parameter.

`name` is component parameter name in the format <component >.<parameter> (`“R1.R”, “V1.V”`). See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names).

Returns -1 if parameter not found, or parameter type is not supported.

Depending on parameter type, the following value is returned:

- formula : number in `double` format
- Initial Condition : number in `double` format if not blank, not supported if blank
- `“On/Off”` : 1 for `“On”`, 0 for `“Off”`
- `“High/Low”` : 1 for `“High”`, 0 for `“Low”`
- `“Yes/No”` : 1 for `“Yes”`, 0 for `“No”`
- text list : parameter number in the list (zero based)

Other parameter types are not supported.

NL5_SetValue

Prototype:

```
int NL5_SetValue(int ncir, char* name, double v)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with parameter name
double	v	– parameter value

Returns:

0 : OK
-1 : error

Description

Sets value of parameter to `v`.

`name` is component parameter name in the format <component >.<parameter> (`"R1.R", "V1.V"`).
See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names).

Returns -1 if parameter not found, or parameter type is not supported.

Depending on parameter type, number `v` is interpreted as follows:

- formula : number in `double` format
- Initial Condition : number in `double` format
- `"On/Off"` : 1 for `"On"`, 0 for `"Off"`
- `"High/Low"` : 1 for `"High"`, 0 for `"Low"`
- `"Yes/No"` : 1 for `"Yes"`, 0 for `"No"`
- text list : parameter number in the list (zero based)

Other parameter types are not supported.

NL5_GetText

Prototype:

```
int NL5_GetText(int ncir, char* name, char* text, int length)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with parameter name
*char**	text	– pointer to null-terminated ASCII character string with parameter text
int	length	– max number of characters allowed to return into `text`, including trailing null

Returns:

>=0 : number of characters returned into `text`, including trailing null.
-1 : error

Description

Returns text (parameter value in text format) of component parameter or model into character string `text`.

`name` is component parameter name in the format <component >.<parameter> (“`R1.R`”, “`V1.V`”). For component model, use <component >.model format (“`V1.model`”). See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names).

Size of character string `text` should be not less than `length.`

Returns -1 if parameter not found, or parameter type is not supported.

Practically all parameter types are supported. The text returned is the same as displayed in the components window of NL5 Circuit Simulator.

If parameter is defined as a formula, text of the formula will be returned.

NL5_SetText

Prototype:

```
int NL5_SetText(int ncir, char* name, char* text)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with parameter name
*char**	text	– pointer to null-terminated ASCII character string with parameter `text`

Returns:

0 : OK
-1 : error

Description

Sets text of component parameter `name` or model to `text.`

`name` is component parameter name in the format <component >.<parameter> ("`R1.R`", "`V1.V`"). For component model, use <component >.model format ("`V1.model`"). See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names).

Returns -1 if parameter not found, or parameter type is not supported.

Practically all parameter types are supported. The text provided is expected to be the same as displayed in the components window of NL5 Circuit Simulator.

To enter a formula for parameter of "formula" type, provide text of the formula started with equal sign '`=`'.

NL5_GetParam

Prototype:

```
int NL5_GetParam(int ncir, char* name)
```

Parameters:

int ncir – circuit handle
*char** name – pointer to null-terminated ASCII character string with parameter name

Returns:

>=0 : parameter handle
-1 : error

Description

`name` is component parameter name in the format <component >.<parameter> (“R1.R”, “V1.V”). See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names).

Returns non-negative handle of component parameter, or -1 if parameter not found.

NL5_GetParamValue

Prototype:

```
int NL5_GetParamValue(int ncir, int npar, double* v)
```

Parameters:

int	ncir	–	circuit handle
int	npar	–	parameter handle
*double**	v	–	pointer to the variable

Returns:

0 : OK
-1 : error

Description

Returns `double` value of parameter with handle `npar` into variable `v`. Parameter handle `npar` should be obtained by function `NL5_GetParam`.

Returns -1 if parameter handle `npar` is not valid, or parameter type is not supported.

Depending on parameter type, the following value is returned:

- formula : number in `double` format
- Initial Condition : number in `double` format if not blank, not supported if blank
- `"On/Off"` : 1 for `"On"`, 0 for `"Off"`
- `"High/Low"` : 1 for `"High"`, 0 for `"Low"`
- `"Yes/No"` : 1 for `"Yes"`, 0 for `"No"`
- text list : parameter number in the list (zero based)

Other parameter types are not supported.

NL5_SetParamValue

Prototype:

```
int NL5_SetParamValue(int ncir, int npar, double v)
```

Parameters:

int	ncir	– circuit handle
int	npar	– parameter handle
double	v	– parameter value

Returns:

0 : OK
-1 : error

Description

Sets value of parameter with handle `npar` to `v`. Parameter handle `npar` should be obtained by function `NL5_GetParam`.

Returns -1 if parameter handle `npar` is not valid, or parameter type is not supported.

Depending on parameter type, number `v` is interpreted as follows:

- formula : number in `double` format
- Initial Condition : number in `double` format
- "`On/Off`" : 1 for "`On`", 0 for "`Off`"
- "`High/Low`" : 1 for "`High`", 0 for "`Low`"
- "`Yes/No`" : 1 for "`Yes`", 0 for "`No`"
- text list : parameter number in the list (zero based)

Other parameter types are not supported.

NL5_GetParamText

Prototype:

```
int NL5_GetParamText(int ncir, int npar, char* text, int length)
```

Parameters:

int	ncir	– circuit handle
int	npar	– parameter handle
*char**	text	– pointer to null-terminated ASCII character string with parameter `text`
int	length	– max number of characters allowed to return into `text`, including trailing null

Returns:

>=0 : number of characters returned into `text`, including trailing null.
-1 : error

Description

Copies text (parameter value in text format) of component parameter with handle `npar` into character string `text`.

Parameter handle `npar` should be obtained by function `NL5_GetParam`.

Size of character string `text` should be not less than `length`.

Returns -1 if parameter handle `npar` is not valid, or parameter type is not supported.

Practically all parameter types are supported. The text returned is the same as displayed in the components window of NL5 Circuit Simulator.

If parameter is defined as a formula, text of the formula will be returned.

NL5_SetParamText

Prototype:

```
int NL5_SetParamText(int ncir, int npar, char* text)
```

Parameters:

int	ncir	– circuit handle
int	npar	– parameter handle
*char**	text	– pointer to null-terminated ASCII character string with parameter `text`

Returns:

0 : OK
-1 : error

Description

Sets text of component parameter with handle `npar` to `text`. Parameter handle `npar` should be obtained by function `NL5_GetParam`.

Returns -1 if parameter handle `npar` is not valid, or parameter type is not supported.

Practically all parameter types are supported. The text provided is expected to be the same as displayed in the components window of NL5 Circuit Simulator.

To enter a formula for parameter of "formula" type, provide text of the formula started with equal sign '='.

NL5_GetTrace

Prototype:

```
int NL5_GetTrace(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with trace name

Returns:

>=0 : trace handle
-1 : error

Description

`name` is the trace name in the format used by NL5 Circuit Simulator. See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names/Trace).

Returns non-negative trace handle, or -1 if trace `name` not found.

NL5_AddVTrace

Prototype:

```
int NL5_AddVTrace(int ncir, char* name)
```

Parameters:

int	`ncir`	– circuit handle
*char**	`name`	– pointer to null-terminated ASCII character string with component `name`

Returns:

>=0	:	trace handle
-1	:	error

Description

Creates voltage trace for component `name`.

Returns non-negative trace handle, or -1 if component `name` not found, or voltage trace is not supported by the component.

NL5_AddITrace

Prototype:

```
int NL5_AddITrace(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with component `name`

Returns:

>=0 :	trace handle
-1 :	error

Description

Creates current trace for component `name`.

Returns non-negative trace handle, or -1 if component `name` not found, or current trace is not supported by the component.

NL5_AddPTrace

Prototype:

```
int NL5_AddPTrace(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with component `name`

Returns:

>=0	: trace handle
-1	: error

Description

Creates power trace for component `name`.

Returns non-negative trace handle, or -1 if component `name` not found, or power trace is not supported by the component.

NL5_AddVarTrace

Prototype:

```
int NL5_AddVarTrace(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with schematic variable `name`

Returns:

>=0 : trace handle
-1 : error

Description

Creates trace for schematic variable `name`.

Returns non-negative trace handle, or -1 if variable `name` not found.

NL5_AddFuncTrace

Prototype:

```
int NL5_AddFuncTrace(int ncir, char* text)
```

Parameters:

int	ncir	– circuit handle
*char**	text	– pointer to null-terminated ASCII character string with function text

Returns:

>=0 : trace handle
-1 : error

Description

Creates trace of function `text`. See NL5 Circuit Simulator Manual for details on function trace (Transient Analysis/Transient Data/Traces/Function trace).

Returns non-negative trace handle, or -1 if error occurred.

NL5_AddDataTrace

Prototype:

```
int NL5_AddDataTrace(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with trace name

Returns:

>=0 : trace handle
-1 : error

Description

Creates trace of **Data** type for post-processing data.

Returns non-negative trace handle, or -1 if error occurred.

NL5_DeleteTrace

Prototype:

```
int NL5_DeleteTrace(int ncir, int ntrace)
```

Parameters:

int `ncir` – circuit handle
int `ntrace` – trace handle

Returns:

0 : OK
-1 : error

Description

Deletes traces with trace handle `ntrace`.

NL5_GetInput

Prototype:

```
int NL5_GetInput(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with component name

Returns:

>=0 : input handle
-1 : error

Description

`name` is component name.
The following component types are supported:

- Label
- Voltage source
- Current source

Returns non-negative input handle or -1 if component not found, or is not supported as an input. The model of the component will be automatically changed to 'V" (constant voltage source) or "I" (constant current source).

NL5_SetInputValue

Prototype:

```
int NL5_SetInputValue(int ncir, int nin, double v)
```

Parameters:

int	ncir	– circuit handle
int	nin	– input handle
double	v	– parameter value

Returns:

0 : OK
-1 : error

Description

Sets voltage or current of the input with handle `npar` to `v`. Input handle `nin` should be obtained by function `NL5_GetInput`.

Returns -1 if input handle `nin` is not valid.

NL5_SetInputLogicalValue

Prototype:

```
int NL5_SetInputLogicalValue(int ncir, int nin, int i)
```

Parameters:

int	ncir	– circuit handle
int	nin	– input handle
int	i	– parameter value

Returns:

0 : OK
-1 : error

Description

Sets voltage or current of the input with handle `npar` to:

- low logical level value, if `i == 0`
- high logical level value, if `i != 0`

Logical levels are set up in the NL5 Transient Settings, Advanced settings, Transient tab.

Returns -1 if input handle `nin` is not valid.

NL5_GetOutput

Prototype:

```
int NL5_GetOutput(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with component name

Returns:

>=0 : input handle
-1 : error

Description

`name` is label or component name

Returns non-negative output handle or -1 if component not found, or is not supported as an output.

NL5_GetOutputValue

Prototype:

```
int NL5_GetOutputValue(int ncir, int nout, double* v)
```

Parameters:

int	ncir	– circuit handle
int	nout	– output handle
*double**	v	– pointer to the variable

Returns:

0 : OK
-1 : error

Description

Sets `double` value of voltage of output with handle `nout` into variable `v`.

Returns -1 if output handle `nout` is not valid.

NL5_GetOutputLogicalValue

Prototype:

```
int NL5_GetOutputValue(int ncir, int nout, int* i)
```

Parameters:

int	ncir	– circuit handle
int	nout	– output handle
*int**	i	– pointer to the variable

Returns:

0 : OK
-1 : error

Description

Sets `int` value of logical level of output with handle `nout` into variable i:

- 0, if output voltage is below logical threshold
- 1, if output voltage is equal or above logical threshold

Logical threshold is set up in the NL5 Transient Settings, Advanced settings, Transient tab.

Returns -1 if output handle `nout` is not valid.

NL5_SetStep

Prototype:

```
int NL5_SetStep(int ncir, double step)
```

Parameters:

int ncir – circuit handle
double step – calculation step

Returns:

0 : OK
-1 : error

Description

Sets maximum calculation step size. If this function was not called, an original calculation step from schematic file will be used (Transient/Settings/"Calculation step").

NL5_SetTimeout

Prototype:

```
int NL5_SetTimeout(int ncir, int t)
```

Parameters:

int	ncir	– circuit handle
int	t	– time-out, seconds

Returns:

0 : OK
-1 : error

Description

Sets maximum time allowed for calculating one simulation step. If this function was not called, a default time-out value is used (0). If time-out is equal to zero, time-out detection is disabled.
If time-out occurred due to unresolved switching iterations, the error message will indicate a component which started switching process. Time-out may also occur due to infinite while/do/for loops of C-code.

NL5_GetSimulationTime

Prototype:

```
int NL5_GetSimulationTime(int ncir, double* t)
```

Parameters:

int	ncir	– circuit handle
*double**	t	– pointer to time variable

Returns:

0 : OK
-1 : error

Description

Sets `t` to the current value of internal `simulation_time` variable.

NL5_Start

Prototype:

`int NL5_Start(int ncir)`

Parameters:

`int ncir` – circuit handle

Returns:

0 : OK
-1 : error

Description

Start simulation.

The function resets internal `simulation_time` variable to 0, initializes circuit components, erases existing simulation data, and calculates initial state of the circuit according to specified Initial Conditions. When function returns, the simulation data consists of circuit state at `t=0`.

The function should be called first to start simulation from `t=0`, prior to calling any simulation functions. However, calling `NL5_Start` is not required. It will be executed automatically if any of simulation functions is called, and simulation has not been performed yet.

The function may return error code if not-DLL enabled schematic contains too many components after loading subcircuits.

NL5_Simulate

Prototype:

```
int NL5_Simulate(int ncir, double interval)
```

Parameters:

int `ncir` – circuit handle
double `interval` – time interval to simulate, in seconds

Returns:

0 : OK
-1 : error

Description

Performs transient simulation at least for requested `interval`.

The function does not change simulation step in order to stop exactly at the end of requested `interval`, so the time of the last calculated data may exceed requested end time. When next simulation function is called, simulation will be continued with simulation step equal to the last simulation step.

The function may return error code if not-DLL enabled schematic contains too many components after loading subcircuits.

`NL5_SimulateInterval`

Prototype:

```
int NL5_SimulateInterval(int ncir, double interval)
```

Parameters:

int	ncir	– circuit handle
double	interval	– time interval to simulate, in seconds

Returns:

0 : OK
-1 : error

Description

Performs transient simulation exactly for requested `interval`.

The function may adjust (decrease) simulation step in order to stop exactly at the end of requested `interval`. When next simulation function is called, simulation step will be restored, and a new linear range will be started.

Please note that if requested interval is less than simulation step, NL5 may not be able to decrease simulation step exactly as needed, and actual simulated interval will be longer than requested. To avoid that, it is recommended to use simulation step at least not greater than desired intervals.

The function may return error code if not-DLL enabled schematic contains too many components after loading subcircuits.

NL5_SimulateStep

Prototype:

```
int NL5_SimulateStep(int ncir)
```

Parameters:

`int ncir` – circuit handle

Returns:

0 : OK
-1 : error

Description

Performs one step of transient simulation.

When the function returns, `simulation_time` variable is set to the time of last calculated data.

The function may return error code if not-DLL enabled schematic contains too many components after loading subcircuits.

NL5_SaveIC

Prototype:

```
int NL5_SaveIC(int ncir)
```

Parameters:

`int ncir` – circuit handle

Returns:

0 : OK
-1 : error

Description

Saves current component states into components' Initial Conditions.

The function does not save schematic into schematic file.

NL5_GetDataSize

Prototype:

```
int NL5_GetDataSize(int ncir, int ntrace)
```

Parameters:

int ncir – circuit handle
int ntrace – trace handle

Returns:

>=0 : data size (number of data points)
-1 : error

Description

Returns non-negative number of data points of the trace with trace handle `ntrace` or -1 if error occurred.

NL5_GetDataAt

Prototype:

```
int NL5_GetDataAt(int ncir, int ntrace, int n, double* t, double* data)
```

Parameters:

int	ncir	–	circuit handle
int	ntrace	–	trace handle
int	n	–	data point index
*double**	t	–	pointer to time variable
*double**	data	–	pointer to value variable

Returns:

0 : OK
-1 : error

Description

Returns time and data of data point with index `n`. Data index is zero-based.

Returns -1 if index is less than zero, or greater or equal to data size.

NL5_GetLastData

Prototype:

```
int NL5_GetLastData(int ncir, int ntrace, double* t, double* data)
```

Parameters:

int	ncir	–	circuit handle
int	ntrace	–	trace handle
*double**	t	–	pointer to time variable
*double**	data	–	pointer to data variable

Returns:

0 : OK
-1 : error

Description

Sets `t` and `data` to the time and data value of the last data point.

Returns -1 if there is no data.

NL5_GetData

Prototype:

```
int NL5_GetData(int ncir, int ntrace, double t, double* data)
```

Parameters:

int	ncir	– circuit handle
int	ntrace	– trace handle
double	t	– time
*double**	data	– pointer to data variable

Returns:

0 : OK
-1 : error

Description

Sets `data` to the data value at time `t`. The data is calculated as linear interpolation between two data points, with time below and above requested time.

Returns -1 if `t` is less than time of first data point, or greater than the time of last data point.

NL5_AddData

Prototype:

```
int NL5_AddData(int ncir, int ntrace, double t, double data)
```

Parameters:

int	ncir	– circuit handle
int	ntrace	– trace handle
double	t	– time
double	data	– data

Returns:

0 : OK
-1 : error

Description

Add data value `data` at time `t` to specified trace.

NL5_DeleteData

Prototype:

```
int NL5_DeleteData(int ncir, int ntrace)
```

Parameters:

int `ncir` – circuit handle
int `ntrace` – trace handle

Returns:

0 : OK
-1 : error

Description

Delete all data of specified trace.

NL5_SaveData

Prototype:

```
int NL5_SaveData(int ncir, char* name)
```

Parameters:

```
int   ncir  – circuit handle
char* name  – pointer to null-terminated ASCII character string with NL5 data file name
```

Returns:

0 : OK
-1 : error

Description

Save transient data of the schematic with handle `ncir` into the data file.

Use this function to save transient data into the file in NL5 data format. Default file extension is "nlt". The data can be loaded into NL5 and shown on the transient graph.

NL5_SetAC

Prototype:

```
int NL5_SetAC(int ncir, double from, double to, int points, int scale)
```

Parameters:

int	ncir	– circuit handle
double	from	– start frequency
double	to	– end frequency
int	points	– number of frequency points
int	scale	– frequency scale: 0 – logarithmic, 1 - linear

Returns:

0 : OK
-1 : error

Description

Set AC simulation parameters.

NL5_SetACSource

Prototype:

```
int NL5_SetAC(int ncir, char* name)
```

Parameters:

int ncir – circuit handle
*char** name – pointer to null-terminated ASCII character string with component name

Returns:

0 : OK
-1 : error

Description

Set component `name` as a source for AC simulation.

NL5_CalcAC

Prototype:

```
int NL5_CalcAC(int ncir)
```

Parameters:

int ncir – circuit handle

Returns:

0 : OK
-1 : error

Description

Perform AC simulation with simulation parameters specified in the schematic file. Only "Linearize schematic" method is supported.

NL5_GetACTrace

Prototype:

```
int NL5_GetACTrace(int ncir, char* name)
```

Parameters:

int	ncir	– circuit handle
*char**	name	– pointer to null-terminated ASCII character string with trace name

Returns:

>=0 : trace handle
-1 : error

Description

`name` is AC trace name in the format used by NL5 Circuit Simulator. See NL5 Circuit Simulator Manual for details (User Interface/Data format/Names/Trace).

Returns non-negative trace handle, or -1 if trace `name` not found.

NL5_GetACDataSize

Prototype:

```
int NL5_GetACDataSize(int ncir, int ntrace)
```

Parameters:

int `ncir` – circuit handle
int `ntrace` – trace handle

Returns:

>=0 : data size (number of AC data points)
-1 : error

Description

Returns non-negative number of AC data points of the trace with trace handle `ntrace` or -1 if error occurred.

NL5_GetACDataAt

Prototype:

```
    int NL5_GetACDataAt(int ncir, int ntrace, int n, double* f, double* mag,
double* phase)
```

Parameters:

int	ncir	– circuit handle
int	ntrace	– trace handle
int	n	– data point index
*double**	f	– pointer to frequency variable
*double**	mag	– pointer to magnitude variable
*double**	phase	– pointer to phase variable

Returns:

0 : OK
-1 : error

Description

Returns frequency (Hz), magnitude, and phase (radians) values of data point with index `n`. Data index is zero-based.

Returns -1 if index is less than zero, or greater or equal to data size.

NL5_SaveACData

Prototype:

```
int NL5_SaveACData(int ncir, char* name)
```

Parameters:

`int ncir` – circuit handle
`char* name` – pointer to null-terminated ASCII character string with NL5 data file name

Returns:

0 : OK
-1 : error

Description

Save AC data of the schematic with handle `ncir` into the data file.

Use this function to save transient data into the file in NL5 data format. Default file extension is "nlf". The data can be loaded into NL5 and shown on the AC graph.

IV. Attachments

END USER LICENSE AGREEMENT

This End-User License Agreement ("EULA", "Agreement") is a legal agreement between you ("you", either an individual or a single entity) and Sidelinesoft, LLC ("Sidelinesoft") for the NL5 Circuit Simulator and NL5 DLL software ("the Software", "the Software Product"), NL5 License ("the Software License"), and accompanying documentation.

Ownership
The Software, any accompanying documentation, and all intellectual property rights therein are owned by Sidelinesoft. The Software is licensed, not sold. The Software is protected by copyright laws and treaties, as well as laws and treaties related to other forms of intellectual property. The Licensee's license to download, use, copy, or change the Software Product is subject to these rights and to all the terms and conditions of this Agreement.

Acceptance
YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT BY DOWNLOADING THE SOFTWARE PRODUCT OR BY INSTALLING, USING, OR COPYING THE SOFTWARE PRODUCT. YOU MUST AGREE TO ALL OF THE TERMS OF THIS AGREEMENT BEFORE YOU WILL BE ALLOWED TO DOWNLOAD THE SOFTWARE PRODUCT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, YOU MUST NOT INSTALL, USE, OR COPY THE SOFTWARE PRODUCT.

License Grant
Sidelinesoft grants you a right to download, install, and use unlimited copies of the Software Product. Without a Software License, the Software operates as a Demo version, with limited number of components in the schematic, and possibly some functional and performance limitations. Several types of Full-Function Software Licenses can be obtained at Software Product website (nl5.sidelinesoft.com). Terms and conditions of each type of Full-Function Software License are available at the website and are subject to change without notice.

Restrictions on Reverse Engineering, Decompilation, and Disassembly.
You may not decompile, reverse-engineer, disassemble, or otherwise attempt to derive the source code for the Software Product.

Restrictions on Alteration
You may not modify the Software Product or create any derivative work of the Software Product or its accompanying documentation without obtaining permission of Sidelinesoft. Derivative works include but are not limited to translations. You may not alter any files or libraries in any portion of the Software Product.

Consent to Use of Data
Sidelinesoft may ask for your permission to collect and use technical information gathered as part of the product support services provided to you, if any, related to the Software. Sidelinesoft may use this information solely to improve the Software or to provide customized services to you and will not disclose this information in a form that personally identifies you.

Disclaimer of Warranties and Limitation of Liability
UNLESS OTHERWISE EXPLICITLY AGREED TO IN WRITING BY SIDELINESOFT, SIDELINESOFT MAKES NO OTHER WARRANTIES, EXPRESS OR IMPLIED, IN FACT OR IN LAW, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF

MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OTHER THAN AS SET FORTH IN THIS AGREEMENT.
Sidelinesoft makes no warranty that the Software Product will meet your requirements or operate under your specific conditions of use. Sidelinesoft makes no warranty that operation of the Software Product will be secure, error free, or free from interruption. YOU MUST DETERMINE WHETHER THE SOFTWARE PRODUCT SUFFICIENTLY MEETS YOUR REQUIREMENTS FOR SECURITY AND UNINTERRUPTABILITY. YOU BEAR SOLE RESPONSIBILITY AND ALL LIABILITY FOR ANY LOSS INCURRED DUE TO FAILURE OF THE SOFTWARE PRODUCT TO MEET YOUR REQUIREMENTS. UNDER NO CIRCUMSTANCES SHALL SIDELINESOFT BE LIABLE TO YOU OR ANY OTHER PARTY FOR INDIRECT, CONSEQUENTIAL, SPECIAL, INCIDENTAL, PUNITIVE, OR EXEMPLARY DAMAGES OF ANY KIND (INCLUDING LOST REVENUES OR PROFITS OR LOSS OF BUSINESS) RESULTING FROM THIS AGREEMENT, OR FROM THE PERFORMANCE, INSTALLATION, USE OR INABILITY TO USE THE SOFTWARE PRODUCT, WHETHER DUE TO A BREACH OF CONTRACT, BREACH OF WARRANTY, OR THE NEGLIGENCE OF SIDELINESOFT OR ANY OTHER PARTY, EVEN IF SIDELINESOFT IS ADVISED BEFOREHAND OF THE POSSIBILITY OF SUCH DAMAGES. TO THE EXTENT THAT THE APPLICABLE JURISDICTION LIMITS SIDELINESOFT'S ABILITY TO DISCLAIM ANY IMPLIED WARRANTIES, THIS DISCLAIMER SHALL BE EFFECTIVE TO THE MAXIMUM EXTENT PERMITTED.

Limitation of Remedies and Damages
Your remedy for a breach of this Agreement or of any warranty included in this Agreement is the correction or replacement of the Software Product. Selection of whether to correct or replace shall be solely at the discretion of Sidelinesoft. Any claim must be made within the applicable warranty period. All warranties cover only defects arising under normal use and do not include malfunctions or failure resulting from misuse, abuse, neglect, alteration, improper installation, or a virus. All limited warranties on the Software Product are granted only to you and are non-transferable. You agree to indemnify and hold Sidelinesoft harmless from all claims, judgments, liabilities, expenses, or costs arising from your breach of this Agreement and/or acts or omissions.

Severability
If any provision of this Agreement shall be held to be invalid or unenforceable, the remainder of this Agreement shall remain in full force and effect. To the extent any express or implied restrictions are not permitted by applicable laws, these express or implied restrictions shall remain in force and effect to the maximum extent permitted by such applicable laws.

Termination
This Agreement is effective until terminated. Without prejudice to any other rights, Sidelinesoft may terminate this Agreement if you fail to comply with the terms and conditions of this Agreement. In such event, you must destroy all copies of the Software License.

Governing Law, Dispute Resolution
This Agreement is governed by the laws of the State of Colorado, U.S.A., without regard to its choice of law principles to the contrary.

Contact Information.
Any inquiries regarding this Agreement or the Software may be addressed to Sidelinesoft at the Software Product website (nl5.sidelinesoft.com).

The end