Calling Python from TRNSYS with CFFI

User manual

Nicolas Bernier, MASc student
Bruno Marcotte, Research Associate
Michaël Kummert, Professor

Revision: 2022-05-06 (version 0.6.0)

Department of Mechanical Engineering
Polytechnique Montréal

michael.kummert@polymtl.ca

POLYTECHNIQUE
MONTRÉAL

Short instructions (TL; DR)

If you want to try the software quickly, you can follow the instructions below. They assume you have a recent 64-bit TRNSYS version (TRNSYS 18.4) and you have installed (or are willing to install) Python 3.10 x-64 so that it is found on Windows path, which requires admin privileges. Running **conda environments will not work** out of the box, you will need to read the detailed instructions in section 8.

- If you have Python 3.10 x-64 installed with numpy, check that it is found on the path. For that, you can open a command prompt (Search for "cmd") and type `where python310.dll`. If Windows cannot find it, you have to add the path to that file manually to the search path, or follow the instructions below.
- Installing and configuring Python (Note: it is highly recommended to uninstall other Python versions)
  - Download the latest x-64 Python 3.10 release from https://www.python.org
  - Launch the installer with admin privileges (right-click, run as admin)
  - Make sure you check the "Add Python to PATH" box, and click on "Customize"
  - In the second screen, click "Next"
  - In the third screen, check "Install for all users" and click on "Install"
  - Close the final screen
- Installing the required TRNSYS files
  - All required files are in the ZIP archive provided with this manual
  - Extract the archive to your TRNSYS installation directory (by default, C:\TRNSYS18)
    - After extraction, you should be able to find this PDF document in `%TRNSYS18%\TRNLib\CallingPython-Cffi\Documentation` (where %TRNSYS18% is your TRNSYS installation directory)
  - Open the Simulation Studio, and run the examples located in: `%TRNSYS18%\TRNLib\CallingPython-Cffi\Examples` You can start from these examples to develop your own Python scripts implementing TRNSYS components
- If something does not work, you will have to read some more details in the next pages…

Citation

To cite this document and the associated software, please use:

Bernier, Nicolas, Bruno Marcotte, and Michaël Kummert. 2022. Calling Python from TRNSYS with CFFI. Polytechnique Montréal. DOI: 10.5281/zenodo.6523078

License

Acknowledgements

The idea to use CFFI to call Python from TRNSYS came from a blog post by Noah Brenowitz's (https://www.noahbrenowitz.com/post/calling-fortran-from-python/). The initial code was adapted from his repository (https://github.com/nbren12/call_py_fort).

The developed TRNSYS Type uses a workaround to avoid triggering floating point exceptions when some libraries are imported (such as numpy), as documented here: https://github.com/numpy/numpy/issues/20504.

The CFFI package was created by Armin Rigo & Maciej Fijalkowski (2022a).

Table of contents

1. What does this do? How does it work?

The "Calling Python (CFFI)" component calls a Python module implemented in a .py file located in the same directory as the TRNSYS input file (the *deck* file) at runtime. That script can use any package or library installed in your Python environment. The communication between the (Fortran) TRNSYS DLL and the Python environment takes place thanks to a Foreign Function Interface defined using the CFFI Python package (Rigo & Fijalkowski, 2022a).

In short, the CFFI package allows to define an interface to some predefined Python functions that can be called from any program using C calling conventions. The package produces a DLL (Windows dynamic library) which can then be linked to that calling program – in this case, the calling program is TRNSYS, which is programmed in Fortran but can use C calling conventions. At runtime, the TRNSYS DLL (or more exactly the DLL with the "Calling Python" type) communicates with the CFFI DLL, which communicates with the main Python DLL. Although it sounds complicated, using the TRNSYS Type is as simple as this: configure how many inputs and outputs you need, give the name of the main Python module implementing your model, and code that model respecting a template which deals with the different type of calls during a simulation and predefined data exchange variables (such as parameters, inputs, and outputs of the type, TRNSYS time, etc.).

The Python interface created by CFFI uses predefined function names, and defining new functions requires recoding and rebuilding the interface. But the predefined names in our implementation are generic functions as "assign a Python variable", "get a variable value from Python", and "call a Python function" (any function!). So creating new TRNSYS component models does not require regenerating a new Python interface. It just requires to drag the proforma for the TRNSYS type and create a Python file implementing the model.

The Type itself is implemented in Fortran. It exchanges data with Python through a dictionary (*dict*). So, for example, the inputs to the TRNSYS component at the current time step will be sent to Python as a variable in the dictionary, and the outputs of the TRNSYS component will be read by the Fortran code from the same dictionary.

In order to allow using multiple "Calling Python" components in the same simulation, the data exchange uses a nested dictionary. So if the dictionary is called TRNData, and the main module for a given TRNSYS unit (i.e. an instance of the Type) is called "MyModel" (implemented in MyModel.py), the inputs to that components will be placed by the Fortran code in `TRNData["MyModel"]["inputs"]`, and the first element in that array (i.e. the first input) will be accessed by `TRNData["MyModel"]["inputs"][0]` with Python's 0-based indexing. The Python code will calculate the outputs and place them in `TRNData["MyModel"]["outputs"]`.

The Python module should always define 5 functions with one single variable (the data exchange dictionary). Those functions, described in #TMP#, implement the tasks that must be performed at different moments during the simulation: at start time, during the iterations, and at the end of the simulation (with some subtleties described in #TMP#).

The configuration of the TRNSYS component in the Simulation Studio is very simple: you should define how many inputs and outputs the Python module will use, and specify the name of the main Python file (which is a Python module name, so no spaces or special characters!).

The following sections describe in detail the necessary steps to configure your system and (if required) rebuild the Python interface for a different Python version. If your system is already configured with the correct software versions, you can also check the short instructions (page i) and learn how to use the type by looking at the examples.

2. Installing and configuring Python

A Python version compatible with the interface created by CFFI must be installed on your system with all the packages and libraries that your component models will require. At the minimum, you need to install numpy, and if you need to rebuild the interface you will need to install cffi.

You can install Python from any distribution, but the instructions below refer to the Windows x-64 installer that can be obtained from www.python.org. If you use a different installer, you need to make sure that the main Python DLL is on the Windows search path and that the 2 packages listed above are installed (see below for how to check that). This section does not cover running conda environments – see section 8 for instructions.

2.1. Installation

The steps to install Python are listed below and shown in Figure 1. Note that it is highly recommended to uninstall other Python versions before you proceed.

- Download the latest x-64 Python release from https://www.python.org
- Launch the installer with admin privileges (right-click, choose "run as admin")
- Make sure you check the "Add Python to PATH" box, and click on "Customize"
- In the second screen, click "Next"
- In the third screen, check "Install for all users" and click on "Install"
- Close the final screen

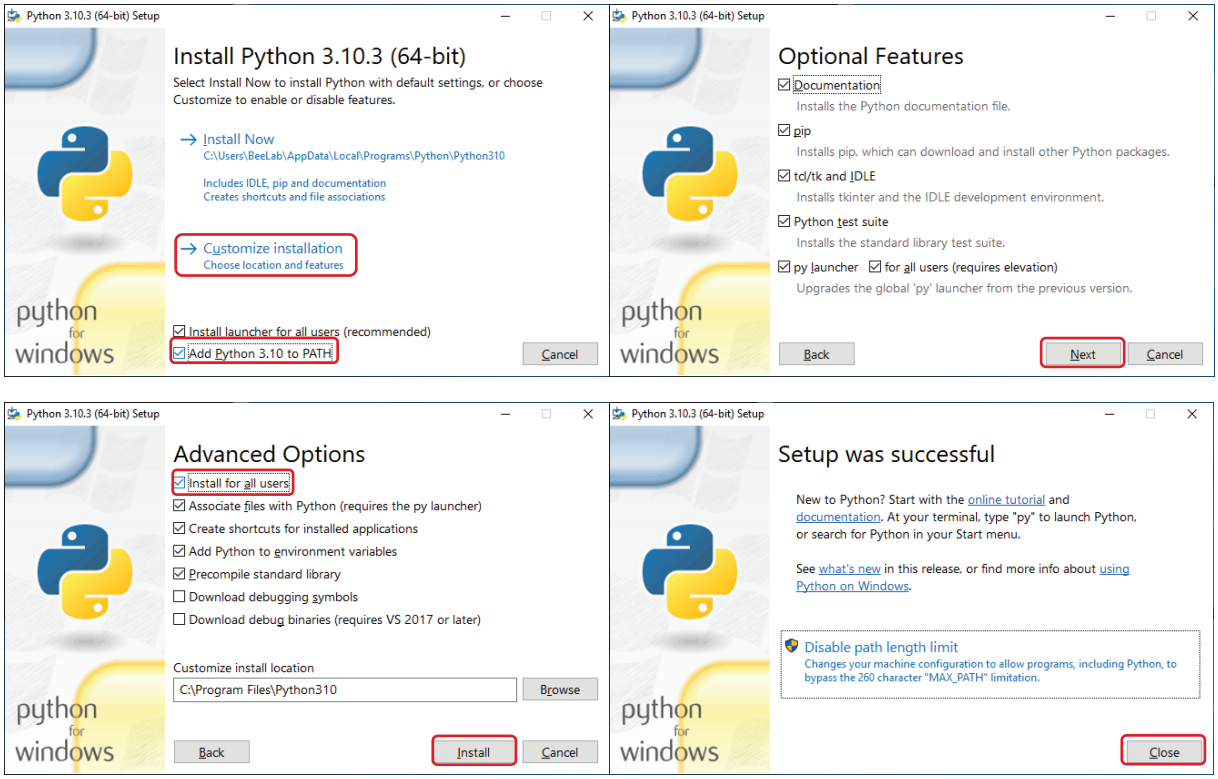

Figure 1: Installing Python in 4 easy steps (but with administrative privileges)

2.2. Configuration and checks

To check that Python is correctly installed, launch a command prompt with administrative privileges (Use Windows search, type "cmd" and you will see the shortcut, right-click on that shortcut and select "run as admin"). Type the following commands:

- `python --version`
  This should return the installed version
- `where python310.dll`
  [replace 310 by the actual version number you want to test, e.g., for version 3.9, type `where python39.dll`]
  This should return the path to the DLL. If Windows cannot find it, you should search for that file and add its path to Windows search path.

Install the 2 required packages (or check that they are installed). In the same command prompt, type:

- `pip install numpy`
- `pip install cffi`

If those packages are correctly installed, you should get messages such as the ones in Figure 2.

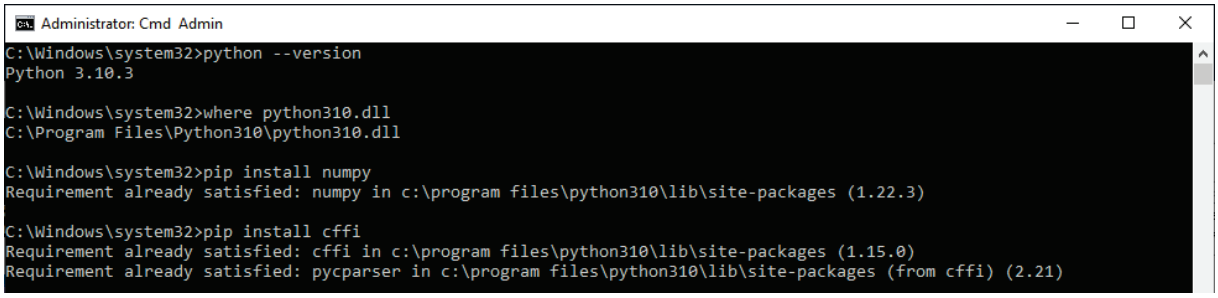

Figure 2: Administrative command prompt with configuration checks (on a system where numpy and cffi were already installed)

3. Installing the required TRNSYS files

The required files are distributed in a ZIP archive which should be extracted in the TRNSYS installation directory (C:\TRNSYS18 by default). After extraction, the following files and directories should be present (.\ represents the TRNSYS installation directory, e.g. C:\TRNSYS18\):

```
.\Exe\PythonInterface.dll
      Precompiled Python interface for Python 3.10, which you will need to replace to
      work with different versions

.\Studio\Proformas\TRNLib\Calling Python (CFFI)\
      Directory with the proforma and icon for the TRNSYS type

.\TRNLib\CallingPython-Cffi\Documentation\
      Directory with the documentation

.\TRNLib\CallingPython-Cffi\Examples\
      Directory with various example files described below

.\TRNLib\CallingPython-Cffi\PythonInterface\
      Directory with files required to regenerate the Python interface
      and batch files to run conda environments

.\TRNLib\CallingPython-Cffi\SourceCode\
      Directory with Fortran source code for the TRNSYS type and an "interface tester"

.\UserLib\DebugDLLs\Type3157.dll
      Precompiled debug version of the type

.\UserLib\ReleaseDLLs\Type3157.dll
      Precompiled release version of the type
```

To test that the files were extracted at the correct location, you can launch the Studio and check that the type appears in the direct access tool (list on the right), as shown in Figure 3. You can also open one of the examples in `.\TRNLib\CallingPython-Cffi\Examples`, but it will not run without regenerating the Python interface, unless you have properly installed and configured Python 3.10.

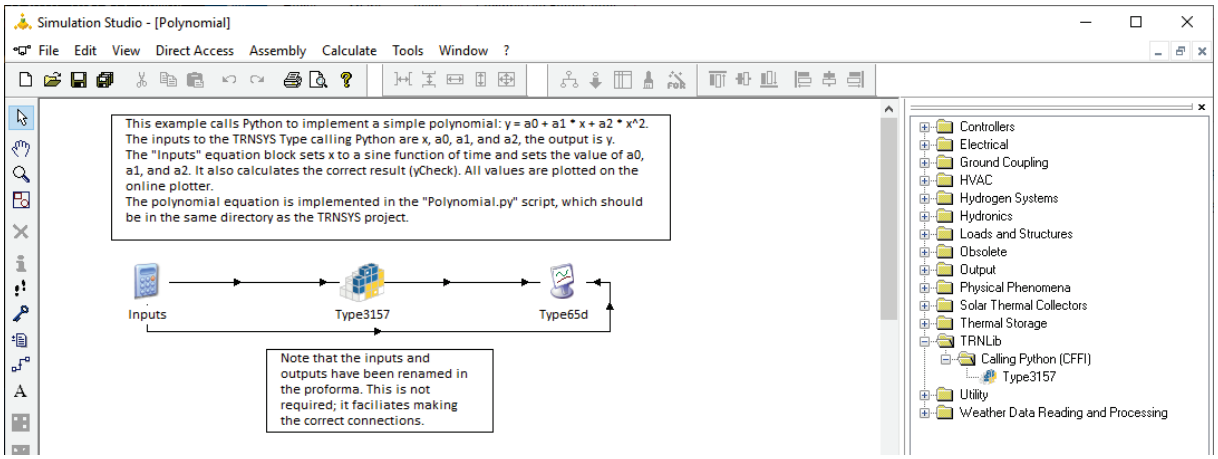

Figure 3: Simulation Studio with the correctly installed component (also showing one of the examples)

4. Rebuilding the Python interface

If you are using a Python version different from the one used in the distributed archive, you will need to regenerate the Python interface. This requires the Visual Studio C++ compiler, which is available freely as part of Microsoft Visual Studio Community.

4.1. Installing Visual Studio

- Download the Visual Studio Community installer from https://visualstudio.microsoft.com/ and launch it
- Select "Desktop development with C++" (this is the only "Workload", do not be fooled by the Python development environment, which is not required).
- If you are trying to save space on your drive, you can unselect optional components on the right, but you must keep the first two optional packages, "build tools" and "Windows SDK" (see Figure 4).
- Click on "Install"

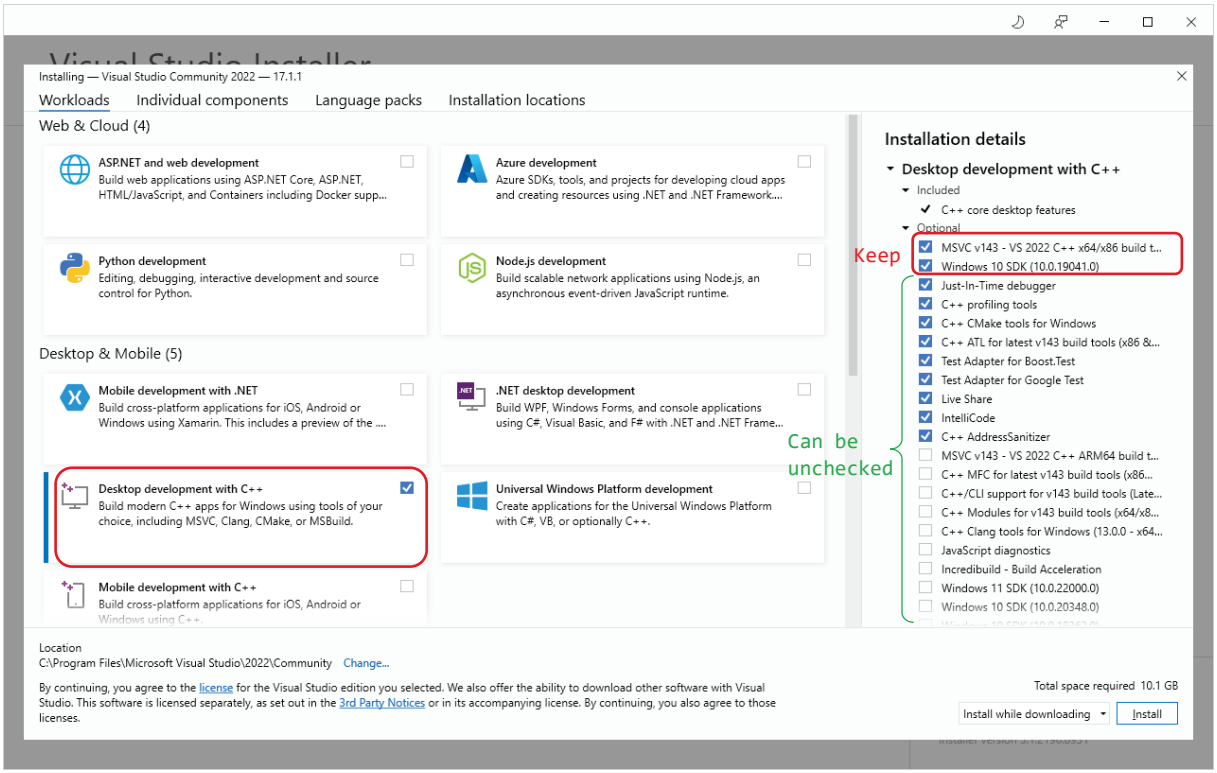

Figure 4: Visual Studio installation with optional components selected by default. You must select "Desktop development with C++, and you must keep the first two optional components (build tools and Windows SDK).

4.2. Rebuilding the Python interface

Open a command prompt (Search for "cmd"). Type the following commands:

- `cd "C:\Trnsys18\TRNLib\CallingPython-Cffi\PythonInterface"`
- `py TrnsysPythonInterfaceBuilder.py`
- This should rebuild PythonInterface.dll and copy it to the proper directory – if the last lines displayed do not indicate that the file was successfully copied, you can manually copy PythonInterface.dll to the TRNSYS18 .\Exe directory

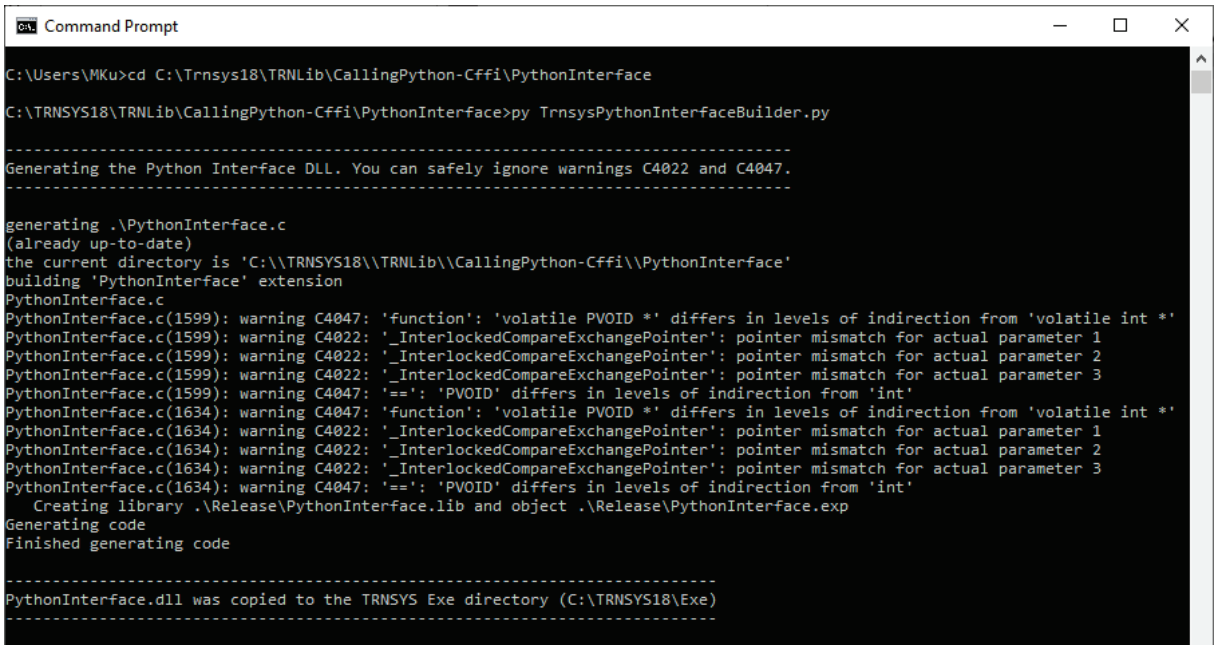

Figure 5: Regenerating the Python interface DLL (PythonInterface.dll, which must be placed in the TRNSYS .\Exe directory)

4.3. Testing the Python interface

In the same command prompt, type `PythonInterfaceTester.exe`. This will launch an executable program written in Fortran that tests the interface outside of TRNSYS, which can facilitate diagnosing problems. You can also use that small program to test some instructions in your Python scripts, by modifying the code in PythonInterfaceTester.py.

If the interface works properly, you should see some results from the data exchange between the Fortran program and the Python environment, as shown in Figure 6.

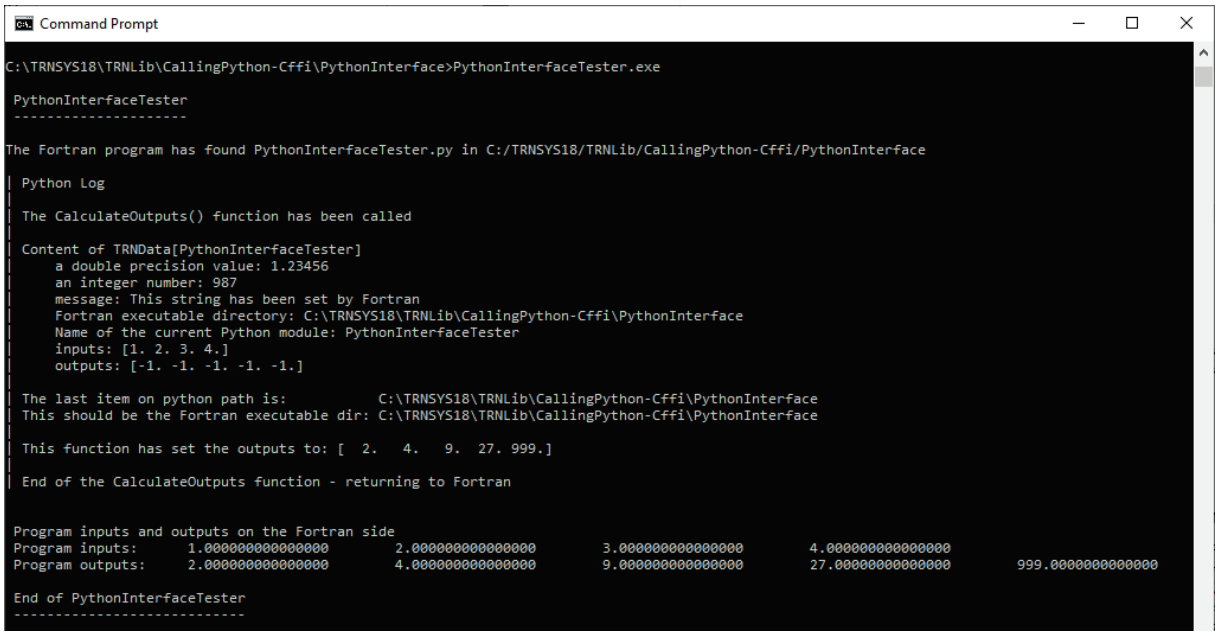

Figure 6: Running the PythonInterfaceTester program to diagnose problems

5. Running TRNSYS examples

The examples are located in .\TRNLib\CallingPython-Cffi\Examples.

5.1. Simple polynomial

The first example implements a simple polynomial equation. The component is configured with 4 inputs and 1 output and is set to operate as a normal iterative component (Parameter 4, Iteration mode, is set to 1, which is the default). The inputs and outputs have been renamed in the proforma, but this is not required and has no impact on the simulation (by default, inputs will be named input-1, input-2, etc.). The "special card" gives the name of the main Python module file. This must comply with Python rules for naming modules, so no spaces or special characters are allowed (and the name is case sensitive). The component configuration is shown in Figure 7.

Note: inputs vs. parameters

If a standard TRNSYS component was created to implement a polynomial equation, calculating $y = a_0 + a_1 x + a_2 x^2$, x and y would probably be selected as input and output, and a_0, a_1, and a_2 would be defined as parameters, as they do not change during the simulation. In this "Calling Python" type, the parameters are used to define how the component is configured and are not passed to Python. So all variables that would typically be considered as "parameters" must be considered as "inputs".

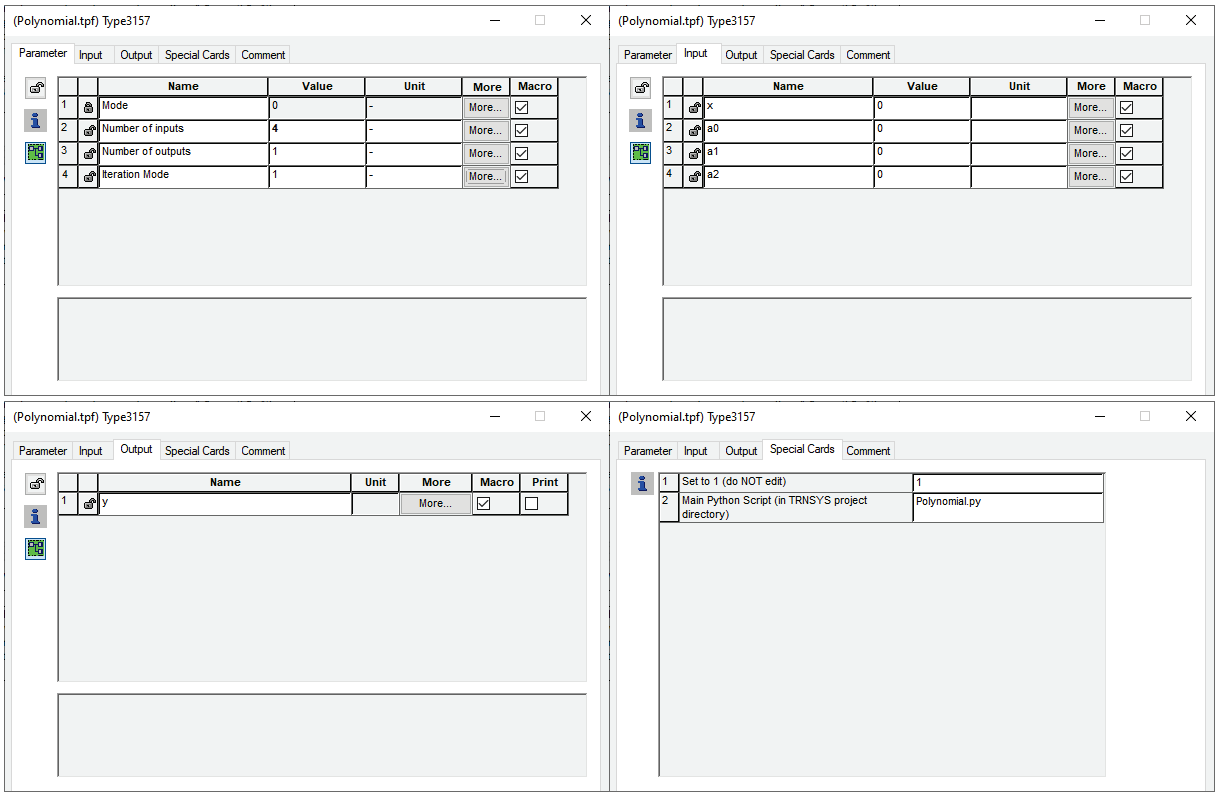

Figure 7: Component configuration in the "Polynomial" example

The TRNSYS project, which is shown Figure 3 (page 2), provides a time-varying value for x (sinusoidal function of the simulation time) and constant values for a_0, a_1, and a_2. The online plotter represents the inputs and outputs, and compares the output calculated by the Python script to a value calculated in an equation block. If the Python component runs correctly, the two values will be the same (the green curve "yCheck" is hidden by the blue curve "y" in .

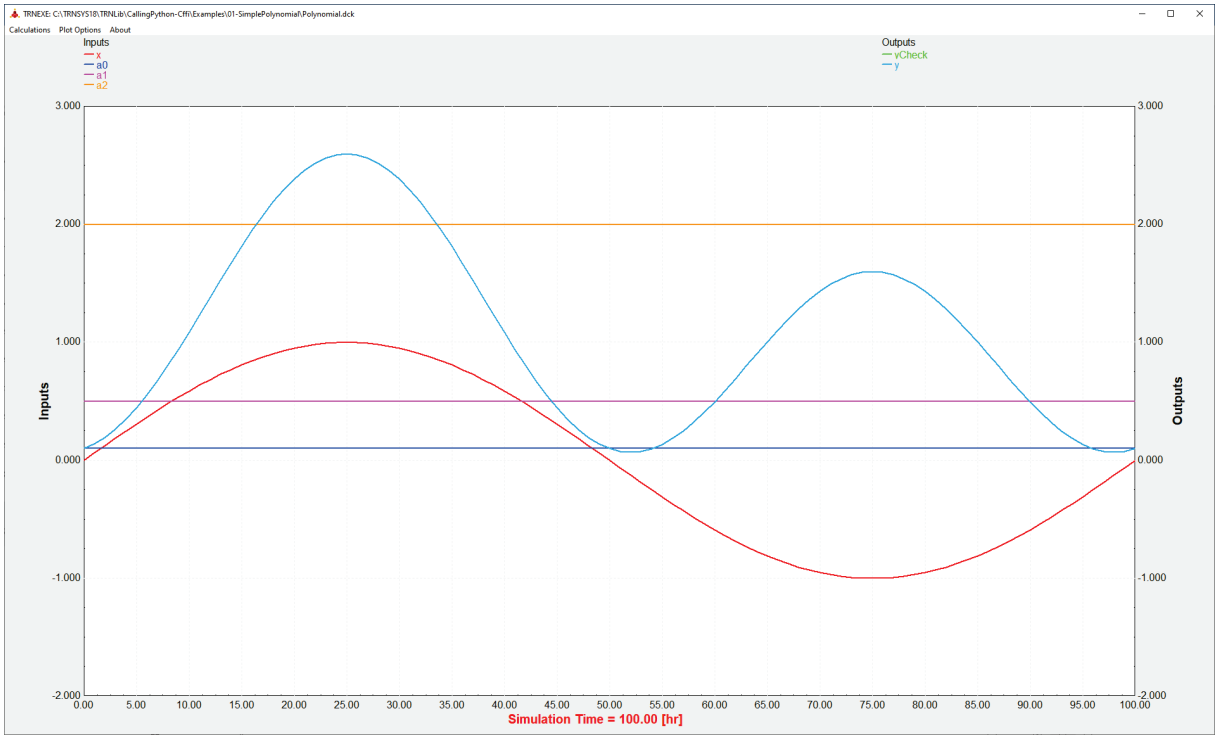

Figure 8: Online plotter showing inputs and calculated results (the two curves for y, calculated by Python, and yCheck, calculated in an equation block, are superimposed)

5.2. Using multiple Python components in the same simulation

The second example shows that multiple Python components can be used in the same simulation. The two components can be configured with a different number of inputs and outputs, and they refer to different python files, as shown in Figure 9. The online plotter again compares the results of the two Python components to values computed in TRNSYS equation blocks. The curves for the Python values (left axis) and the values computed in TRNSYS should be superimposed. Note that in Figure 10, the left axis limits have been slightly modified to cause a small offset so that all lines are visible. In the actual example, the lines will be perfectly superimposed.

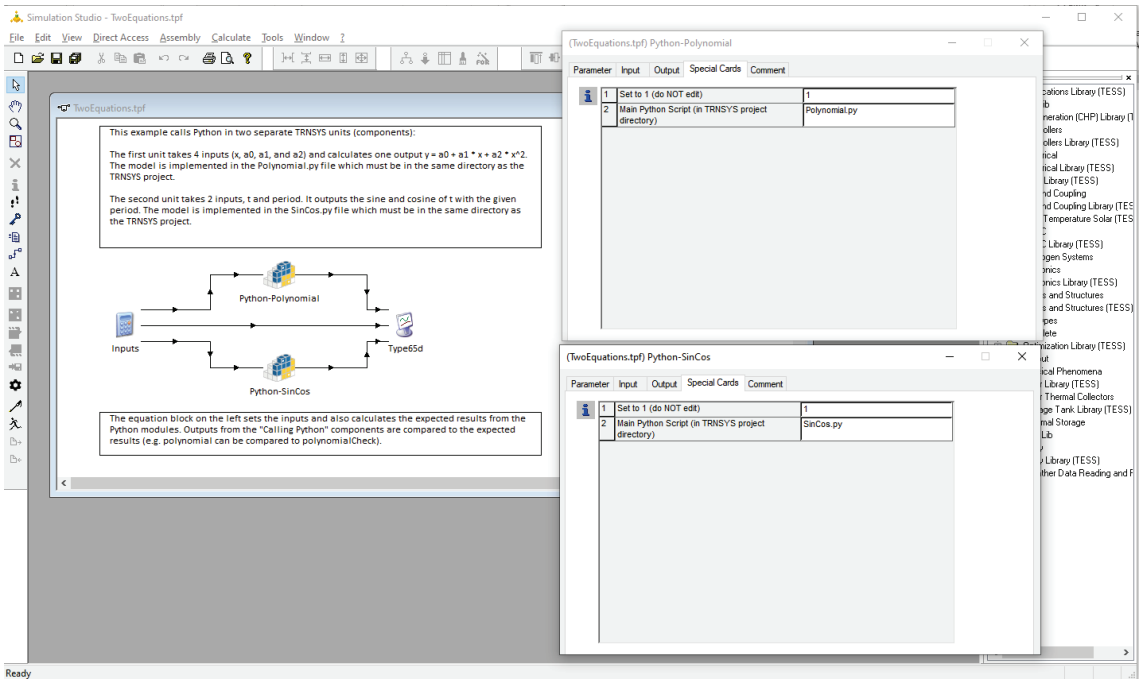

Figure 9: Example showing two Python components in the same simulation. The first component calls a Python module in Polynomial.py, while the second component calls the module in SinCos.py.

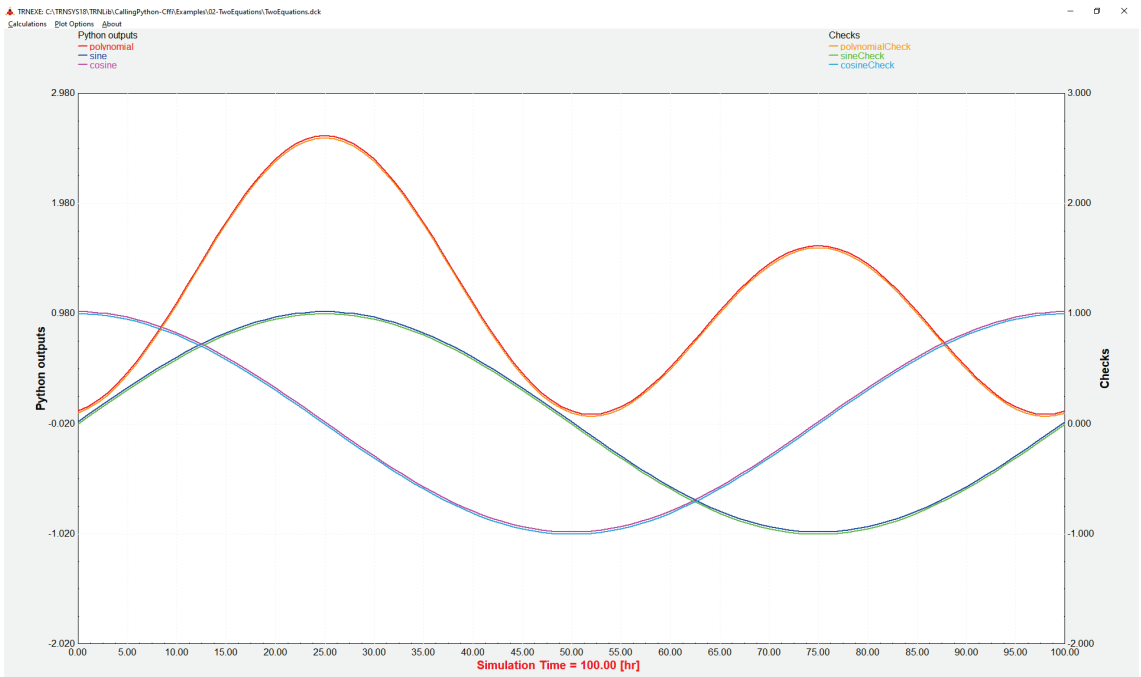

Figure 10: Comparison between the values calculated by the two Python scripts and values calculated in TRNSYS. The left axis limits have been slightly modified so that all lines are visible. In reality, the left variables and the matching right variables are exactly the same.

5.3. Other examples

Other examples can be found in the .\TRNLib\CallingPython-Cffi\Examples directory.

They illustrate various features that can be used in Python scripts.

- Logging
  - This example calls Python to produce log files with simulation history. the Python file illustrates how variables can be stored from one time step to the other and printed to a file at the end of the simulation. Two files are created, a log file with the history of calls, and a CSV file with the history of inputs and outputs. The Python code also shows how to use the TRNSYS input (deck) file name, for example to imitate the result of naming a file ***.out in TRNSYS (where *** means "the name of the deck file)
  - The log file also contains some information on the Python environment (version, location (path), and content of the “sys.path” variable)

6. Coding models in Python

Python modules implementing component models for TRNSYS must respect 2 rules:

- Implement 5 predefined functions corresponding to different types of calls in TRNSYS
- Use a dictionary to exchange data with TRNSYS. Variables in the dictionary are predefined and must respect some conditions (for example, the number of outputs provided by the Python code must match what is specified in the TRNSYS component).

As mentioned above, the module must be implemented in a .py file with a name that is an acceptable module name in Python (no spaces, no special characters).

6.1. Importing Python packages

Your TRNSYS type can import and use any Python package available in your system. Some Python packages are known to generate floating-point exceptions when they are imported (see e.g. https://github.com/numpy/numpy/issues/20504). The Fortran code calling Python implements a workaround for this by disabling floating-point exceptions before calling the "Initialization" function and re-enabling them afterwards. Therefore, **it is highly recommended to import all the required packages right at the top of your module or within the Initialize function**, even if you do not use them there. Do not import them in other functions described below, such as StartTime(), as this could generate floating-point errors that would stop the program.

6.2. Python functions defined in the module

Internally, and in Python's main scope, the data exchange dictionary is called TRNData, but since this is a parameter passed to the functions, you can in fact name it anything you want. In the following, the data exchange dictionary is always called TRNData for clarity, but this is not a requirement. The name of each function, however, must be what is specified below, since those functions are called by the Fortran code and their name is hardcoded in both the Fortran code and the Python interface created by CFFI. The 5 functions that must be defined are:

- **Initialization**(TRNData)
  This function is called during the TRNSYS initialization call, which is not an actual time step. TRNSYS variables such as TIME or the inputs / outputs of a Type should not be accessed during that call, which is reserved to open files, dimension and initialize variables, etc.
- **StartTime**(TRNData)
  This function is called once only, during the TRNSYS "initial time" call, which is not an actual time step either. No iterations are performed, and TRNSYS outputs should just be set to their initial values. Because there are no iterations, the value of the component inputs at that call will depend on the component order in the input file and should not be relied upon.
- **Iteration**(TRNData)
  This function is called at each iterative call to a component. This includes the first call at each time step, and between 0 and "many" calls at the same time step (depending on the iterations required for the simulation to converge at that time step). The component should read its inputs and calculate new outputs, but not update state variables. Let's assume, for example, that you want to implement a counter of the number of time steps in the simulation. If the code was adding one to the counter in this function, you would add one at each iteration,

and not each time step – so at the end of the simulation you would have the total number of iterative calls to this component in particular, and not a count of the number of time steps.

- **EndOfTimeStep**(TRNData)
  This function is called at the end of each time step, i.e. when the simulation has converged or when the TRNSYS solver has given up for that particular time step. The component should not perform new calculations that depend on the inputs, because in theory the inputs will be the same (within the convergence tolerance) as during the last iterative call. The component should update state variables (such as the counter in the example above) and possibly set output values that should be used at the next time step (for example for a controller that mimics an real-time device that receives measurements and applies a control signal based on those measurements for the next time step, and not for the current time step, as real-time devices cannot perform iterations…
- **LastCallOfSimulation**(TRNData)
  This function is called once at the very end of the simulation. Components should not perform new calculations depending on the inputs, as they have no meaning at this point. Outputs also have no meaning. This call should be used for cleanup tasks, for example closing files that were opened during the simulation. It is important to realize that this call occurs after the user has closed the online plotter (or the dialog box if there is no online plotter) at the end of the simulation. If, for example, you run a simulation to the end but then keep the online plotter open to examine results, this call will not be performed until you close the plotter. So in the meantime, output files for example will remain open within TRNSYS, and you should not access them. Another peculiarity of this call is that, as it happens after the main executable program for TRNSYS is closed, there is no Windows program to send error messages to, and potential Python errors happening during this call will not cause any message. For those reasons, it is recommended to implement "end of simulation" tasks in the "EndOfTimeStep" function, within a conditional block that tests whether the TRNSYS time is the final simulation time. But even if it is empty, this function must be present in every Python module implementing a TRNSYS component.

6.3. Variables in the data exchange dictionary

Each module (i.e. each instance of the Type calling Python) uses a nested dictionary within the main dictionary. For example, if a simulation uses a component calling "Polynomial.py" and another component calling "SinCos.py", the data for the first component will be in TRNData["Polynomial"] and the data for the other component will be in TRNData["SinCos"]. The variables present within these dictionaries are predefined. In the following, they are just described by their name, but they must be accessed by their name within the dictionary. For example, the inputs for the "SinCos" module must be accessed as TRNData["SinCos"]["inputs"].

- **TRNSYS input file path (string)**
  Full path to the TRNSYS input file
  Example: C:\TRNSYS18\TRNLib\CallingPython-Cffi\Examples\01-Polynomial\Polynomial.dck
  The Logging example shows how to keep only the input file name without the path and extension
- **number of inputs (integer)**
  Number of inputs defined in the TRNSYS input file
- **number of outputs (integer)**
  Number of outputs defined in the TRNSYS input file
- **simulation start time (real, double precision)**
  TRNSYS simulation start time, in hours (0 if the simulation starts at time=0)
- **simulation stop time (real, double precision)**
  TRNSYS simulation stop time, in hours (e.g. 8760 for a 1-year simulation)
- **simulation time step (real, double precision)**
  TRNSYS simulation time step, in hours (e.g. 0.05 for a 3-min time step)

- **total number of time steps (integer)**
  Total number of time steps to be completed in the simulation (can be useful to dimension arrays intended to keep the history of simulation results, for example)
- **current time step number (integer)**
  Number of the current time step
- **time (real, double precision)**
  TRNSYS time, in hours (current time in the simulation, in hours)
- **inputs (numpy array of double precision numbers)**
  Inputs of this component
- **outputs (numpy array of double precision numbers)**
  Outputs of this component

Some examples:

To access the number of inputs for a component calling a file named "MyModule.py":

```
TRNData["MyModule"]["number of inputs"]
```

To know the length of the TRNSYS time step, in hours:

```
TRNData["MyModule"]["simulation time step"]
```

In the examples, the Python module name is placed in a variable, to simplify transferring code between different modules:

```
thisModule = os.path.splitext(os.path.basename(__file__))[0]
# The name of the current module is now in the thisModule variable
```

So that all variables can be accessed by using TRNData[thisModule]…, for example, to get the number of outputs:

```
TRNData[thisModule]["number of outputs"]
```

To access the first input of the current module (assuming thisModule is set to the module name as explained above):

```
TRNData["MyModule"]["inputs"][0]    # Do not forget that Python indexing starts at 0!
```

6.4. Minimum working example

The following code will run, provided it is placed in a .py file (e.g. MyModule.py) and called from a TRNSYS simulation with a properly configured component (i.e. correct number of inputs and outputs, 1 each). This component simply outputs the square of the input value.

```
# Python module for the TRNSYS Type calling Python using CFFI
# Data exchange with TRNSYS uses a dictionary, called TRNData in this file
# (it is the argument of all functions).

import numpy
import os

# For convenience the module name is saved in thisModule
thisModule = os.path.splitext(os.path.basename(__file__))[0]

# Initialization: function called at TRNSYS initialization
# -------------------------------------------------------
def Initialization(TRNData):
    # This model has nothing to initialize
    return

# StartTime: function called at TRNSYS starting time
# (not an actual time step, initial values should be reported)
# -------------------------------------------------------------
def StartTime(TRNData):
    # Output an initial value = 0
    TRNData[thisModule]["outputs"][0] = 0.0
    return

# Iteration: function called at each TRNSYS iteration within a time step
# ----------------------------------------------------------------------
def Iteration(TRNData):

    # Calculate outputs and set them in TRNData
    TRNData[thisModule]["outputs"][0] = numpy.power(TRNData[thisModule]["inputs"][0], 2)
    return

# EndOfTimeStep: function called at the end of each time step,
# after iteration and before moving on to next time step
# ------------------------------------------------------------
def EndOfTimeStep(TRNData):
    # This model has nothing to do during the end-of-step call
    return

# LastCallOfSimulation: function called at the end of the simulation (once)
# -------------------------------------------------------------------------
def LastCallOfSimulation(TRNData):
    # This model has nothing to do at the end of the simulation
    return
```

You can explore the provided examples for more complete code.

7. Type 3157 Parameter-Input-Output reference

Table 1: Parameters, inputs and outputs

Parameters						
No	**Name**	**Dimensions**	**Units**	**Type**	**Range**	**Default**
1	Mode	Dimensionless	-	integer	[0;0]	0
	Not implemented yet					
2	Number of inputs	Dimensionless	-	integer	[1;200]	1
	Number of inputs sent to Python (nInputs)					
3	Number of outputs	Dimensionless	-	integer	[1;200]	1
	Number of outputs received from Python (nOutputs)					
4	Iteration mode	Dimensionless	-	integer	[1;3]	1
	This parameter describes the iterative behavior of this component. 1: Standard Iterative component (called at each call of each time step) 2: Non-iterative component called at the end of each time step, after integrators and printers - This is suitable for a controller that calculates its outputs for one time step based on the converged ("measured") values of previous time step 3: Non-iterative component called at the end of each time step, before integrators and printers - This is suitable for user-written statistic subroutines that send their output to integrators					

Inputs						
No	**Name**	**Dimensions**	**Units**	**Type**	**Range**	**Default**
1 .. nInputs	Input	Any	-	real	[-Inf;+Inf]	0
	Input sent to Python. the total number of inputs is set by parameter 2 (nInputs)					

Outputs						
No	**Name**	**Dimensions**	**Units**	**Type**	**Range**	**Default**
1 .. nOutputs	Outputs	Any	-	real	[-Inf;+Inf]	0
	Outputs sent to Python. the total number of outputs is set by parameter 3 (nOutputs)					

8. Running conda environments

If you want to run conda environments with the TRNSYS Type calling Python through CFFI, you will need to ensure that the correct Python version (including at the minimum the cffi and numpy packages) is found by the Python interface DLL. The instructions below explain how to install and use a minimum working environment with miniconda.

8.1. Installing conda

Download miniconda from https://docs.conda.io/en/latest/miniconda.html. Make sure you select the 64-bit version.

Install miniconda keeping all defaults.

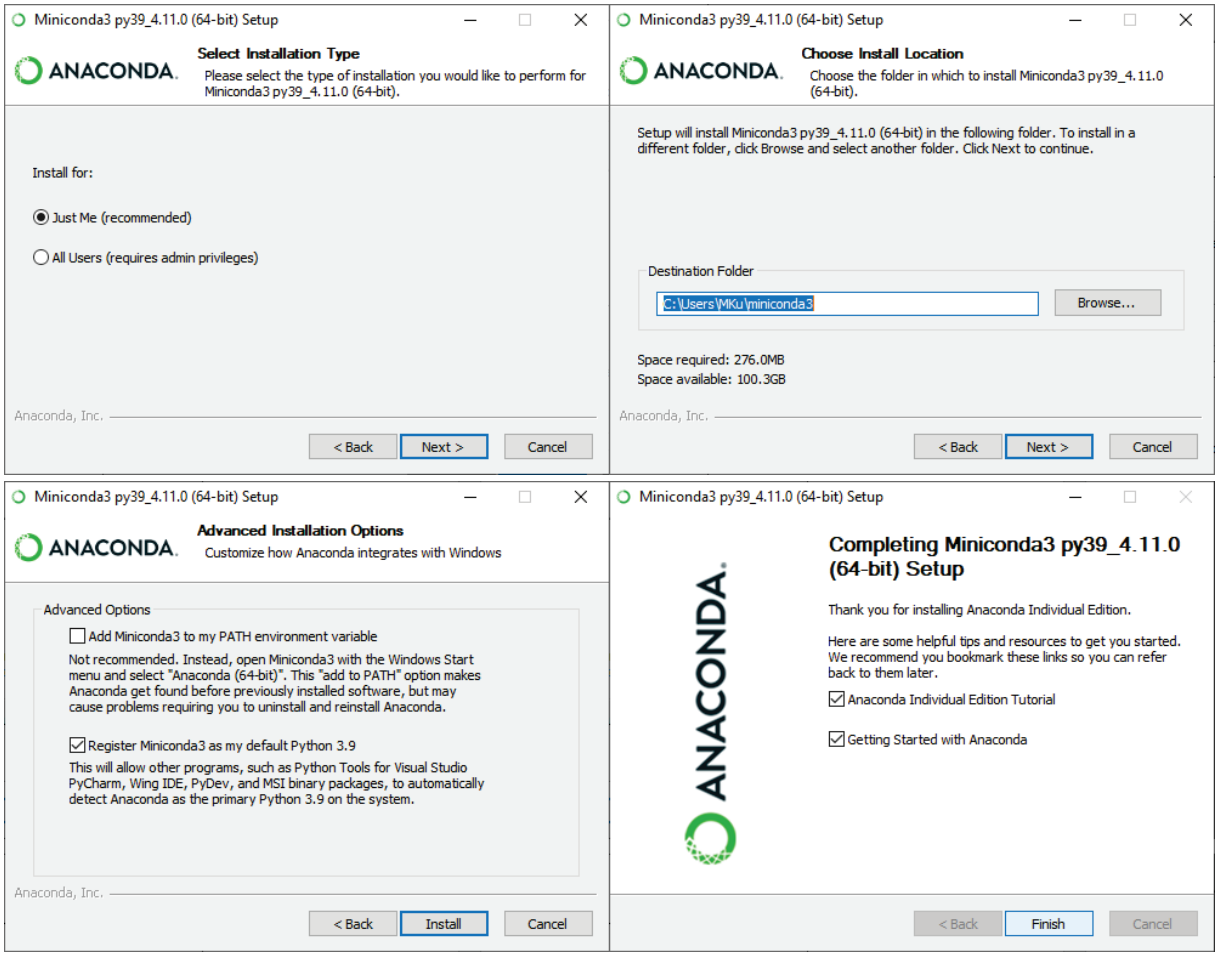

Figure 11: Installing Miniconda

8.2. Creating a conda environment

Launch the Anaconda prompt (not the "regular" Windows command prompt!). In the following, %username% is your Windows user name. In the Anaconda prompt, you need to create an environment (called TRNSYS below), activate it, and finally install numpy and cffi within that environment. The **`black, bold text`** indicates what you need to type, `green text` is comments, and `blue text` is the answer from the conda prompt.

```
# Create the TRNSYS environment
(base) C:\Users\%username%>conda create -n TRNSYS

Collecting package metadata (current_repodata.json): done
Solving environment: done
## Package Plan ##
  environment location: C:\Users\%username%\miniconda3\envs\TRNSYS

Proceed ([y]/n)? y
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

# Activate the TRNSYS environment
(base) C:\Users\%username%>conda activate TRNSYS

# Install numpy within the TRNSYS environment
(TRNSYS) C:\Users\%username%>conda install numpy

Collecting package metadata (current_repodata.json): done
Solving environment: done
## Package Plan ##
  environment location: C:\Users\%username%\miniconda3\envs\TRNSYS
  added / updated specs:
    - numpy
...
The following NEW packages will be INSTALLED:
  blas               pkgs/main/win-64::blas-1.0-mkl
  ...
  wincertstore       pkgs/main/win-64::wincertstore-0.2-py39haa95532_2

Proceed ([y]/n)? y
...
Executing transaction: done

# Install cffi within the TRNSYS environment
(TRNSYS) C:\Users\%username%>conda install cffi

Collecting package metadata (current_repodata.json): done
Solving environment: done
## Package Plan ##
  environment location: C:\Users\%username%\miniconda3\envs\TRNSYS
  added / updated specs:
    - cffi
...
The following NEW packages will be INSTALLED:
  cffi               pkgs/main/win-64::cffi-1.15.0-py39h2bbff1b_1
  pycparser          pkgs/main/noarch::pycparser-2.21-pyhd3eb1b0_0

Proceed ([y]/n)? y
...
Executing transaction: done
```

8.3. Regenerating the Python Interface from the conda environment

The PythonInterface DLL used by TRNSYS must match your Python version, so we need to regenerate it. This will be done from the Anaconda prompt.

Note: in some Windows version, python.exe is redirected to a link to the Microsoft store. You need to remove that link for the miniconda prompt to find python.exe. Go to your Windows settings, search for "App execution aliases", and unselect Python.

Figure 12: Removing Python aliases in Windows

Once that is done, you can launch the Anaconda prompt again, and type the following commands.

```
# Activate the TRNSYS environment
(base) C:\Users\%username%>conda activate TRNSYS

# Change dir to TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface
(TRNSYS) C:\Users\%username%>cd C:\TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface

# Rebuild the Python interface
(TRNSYS) C:\TRNSYS18\...\PythonInterface>python TrnsysPythonInterfaceBuilder.py

--------------------------------------------------------------------------------
Generating the Python Interface DLL. You can safely ignore warnings C4022 and C4047.
--------------------------------------------------------------------------------
generating .\PythonInterface.c
the current directory is 'C:\\TRNSYS18\\TRNLib\\CallingPython-Cffi\\PythonInterface'
running build_ext
building 'PythonInterface' extension
creating Release
"C:\Program Files\Microsoft Visual Studio...\cl.exe" /c /nologo ...\PythonInterface.obj
PythonInterface.c
PythonInterface.c(1599): warning C4047: 'function': 'volatile PVOID *' ...
...
PythonInterface.c(1599): warning C4022: '_InterlockedCompareExchangePointer': ...
"C:\Program Files\Microsoft Visual Studio...\link.exe" /nologo /INCREMENTAL:NO ...
Creating library .\Release\PythonInterface.lib and object .\Release\PythonInterface.exp
Generating code
Finished generating code
--------------------------------------------------------------------------
PythonInterface.dll was copied to the TRNSYS Exe directory (C:\TRNSYS18\Exe)
--------------------------------------------------------------------------
Press ENTER to continue...

(TRNSYS) C:\TRNSYS18\...\PythonInterface>exit
```

8.4. Testing the Python interface with the conda environment

You can test the Python interface by running a batch file that will set some environment variables before launching the tester program. **Do not launch PythonInterfaceTester.exe directly**, it will not run. The batch file is:

`C:\TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface\`**`PythonInterfaceTesterWithCondaEnvironment.bat`**

In this batch file, %username% is an environment variable known by Windows that will be automatically be replaced with your user name. You can replace `TRNSYS` if you selected a different name for your environment.

```
:: Edit the %condaenvname% variable (set to TRNSYS in the example file)
::
:: Set the name of the conda environment to be used
:: (should have cffi and numpy installed at the minimum, edit if required)
set condaenvname=TRNSYS
::
:: Set required environment variables for the conda environment
::
:: Add directory with python to the path (to the front of the path!)
set path=C:\Users\%username%\miniconda3\condabin;%path%
set path=C:\Users\%username%\miniconda3\envs\%condaenvname%;%path%
set path=C:\Users\%username%\miniconda3\envs\%condaenvname%\bin;%path%
set path=C:\Users\%username%\miniconda3\envs\%condaenvname%\Library\mingw-w64\bin;%path%
set path=C:\Users\%username%\miniconda3\envs\%condaenvname%\Library\bin;%path%
set path=C:\Users\%username%\miniconda3\envs\%condaenvname%\Library\usr\bin;%path%
set path=C:\Users\%username%\miniconda3\envs\%condaenvname%\Scripts;%path%
:: Set PYTHONHOME to the same directory
set PYTHONHOME=C:\Users\%username%\miniconda3\envs\%condaenvname%
:: Set PYTHONPATH to the site-packages directory (which is within your environment\Lib)
set PYTHONPATH=C:\Users\%username%\miniconda3\envs\%condaenvname%\Lib\site-packages
::
:: Run PythonInterfaceTester
::
PythonInterfaceTester.exe
```

You should get an output such as:

```
PythonInterfaceTester
---------------------
The Fortran program has found PythonInterfaceTester.py in
C:/TRNSYS18/TRNLib/CallingPython-Cffi/PythonInterface
| The CalculateOutputs() function has been called
| Content of TRNData[PythonInterfaceTester]
|     a double precision value: 1.23456
|     an integer number: 987
|     message: This string has been set by Fortran
|     Fortran executable directory: C:\TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface
|     Name of the current Python module: PythonInterfaceTester
|     inputs: [1. 2. 3. 4.]
|     outputs: [-1. -1. -1. -1. -1.]
| The last item on python path is:  C:\TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface
| This should be:                   C:\TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface
| This function has set the outputs to: [  2.   4.   9.  27. 999.]
| End of the CalculateOutputs function - returning to Fortran
 Program inputs and outputs on the Fortran side
 Program inputs:  1.000 2.000 3.000  4.000
 Program outputs: 2.000 4.000 9.000 27.000 999.000
 End of PythonInterfaceTester - Press ENTER to continue
 -------------------------------------------------------
```

8.5. Using the conda environment from the TRNSYS Studio

When using TRNSYS from the Simulation Studio, the Python interface will only work if environment variables are set correctly, as was the case for the PythonInterfaceTester program.

A batch file is provided to set the environment variables and then launch the Simulation Studio:

```
C:\TRNSYS18\TRNLib\CallingPython-Cffi\PythonInterface\RunTrnsysStudioWithCondaEnvironment.bat
```

After launching that batch file, the Studio will open, and you can browse to your TRNSYS project file (.tpf) and run it. Note that if you close the Studio, you will need to re-run the batch file to set the environment variables properly. The contents of the batch file are the same as above, except that at the end it launches the Simulation Studio instead of PythonInterfaceTester:

```
:: Launch the TRNSYS Studio (edit path if required)
::
C:\TRNSYS18\Studio\Exe\Studio.exe
```

8.5.1. Logging example

The Logging example prints some diagnostic messages to the Logging-PythonLog.log file. When run from the "TRNSYS" conda environment, the first lines of the log file should indicate the Python version and where the environment is located, e.g.:

```
Python version:
3.9.12

Path to Python:
C:\Users\%username%\miniconda3\envs\TRNSYS

Python `sys.path`:
C:\Users\%username%\miniconda3\envs\TRNSYS\Lib\site-packages
C:\Users\%username%\miniconda3\envs\TRNSYS\python39.zip
C:\Users\%username%\miniconda3\envs\TRNSYS\DLLs
C:\Users\%username%\miniconda3\envs\TRNSYS\lib
C:\TRNSYS18\Exe
C:\Users\%username%\miniconda3\envs\TRNSYS
C:\TRNSYS18\TRNLib\CallingPython-Cffi\Examples\03-Logging
```

8.6. Troubleshooting conda environments

Unfortunately, if you have different Python versions installed at the same time or more complex environments, it is likely that you will encounter problems. The easiest method to solve those problems is to do a clean installation of the standard Python version from python.org and add it to the path, as shown in section 2. If you do not want to do that, you will have to ensure that all environment variables are set properly. One method is to open a (regular) Windows command prompt and to copy the lines in the two batch files one at a time, and then launch either the PythonInterfaceTester or the Simulation Studio.

Some known errors are shown below.

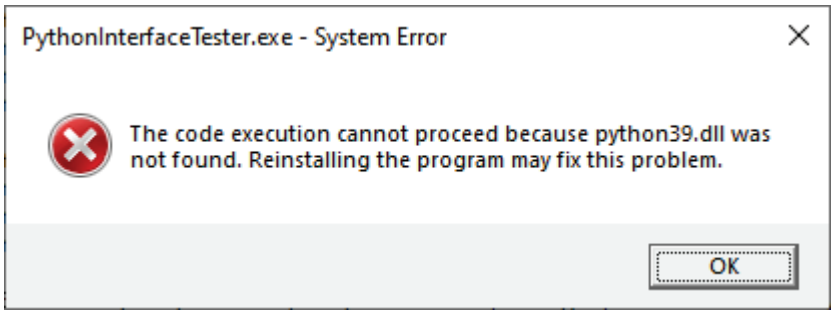

Figure 13: Example of error. The main Python DLL is not found. This is an indication that Python is not on the path and that the environment variables are not correctly set if you are trying to run conda environments. Try to locate the Python DLL with "where python310.dll" (change 310 with the appropriate version number).

Figure 14: Example of error. The main Python DLL is found, but PYTHONHOME and PYTHONPATH are not set correctly, which causes problems with conda environments and results in cryptic error messages on init_fs_encoding. Make sure that all environment variables are set (see batch files provided as examples).

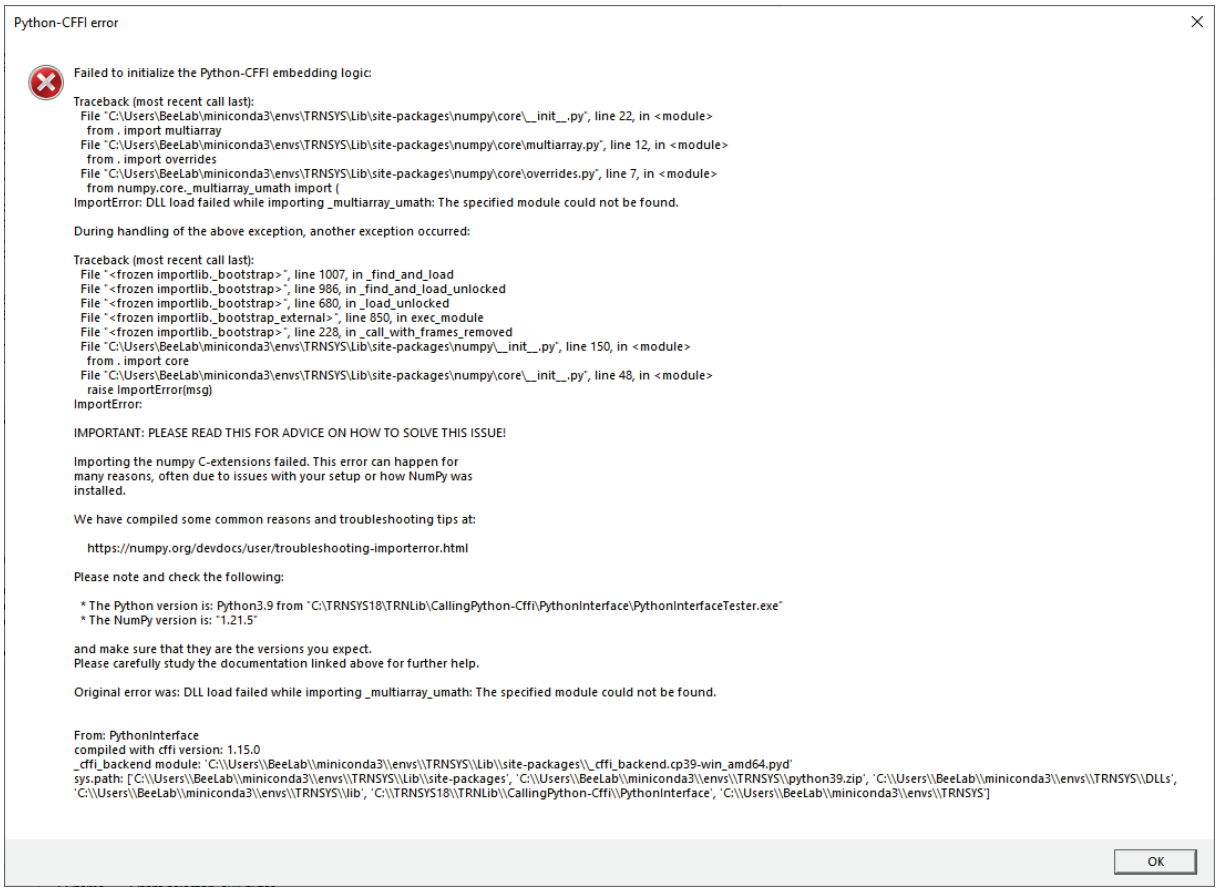

Figure 15: Example of error. Numpy or CFFI errors can be caused by incorrect environment variables, but they can also happen if numpy and cffi were not installed within the proper conda environment. Try reinstalling them by following the steps in section 8.2.

9. References

Rigo, A., & Fijalkowski, M. (2022a). *CFFI - Foreign Function Interface for Python calling C code* (1.15) [Computer software]. https://pypi.org/project/cffi/

Rigo, A., & Fijalkowski, M. (2022b). *Using CFFI for embedding*. CFFI 1.15.0 Documentation. https://cffi.readthedocs.io/en/latest/embedding.html