

11.8 Remote TCP/IP interface.

FIMMWAVE can be accessed remotely via TCP/IP. To do this, you must start `fimmwave.exe` with the `-pt` argument:

```
-pt <portNo>
```

`<portNo>` is the TCP/IP port number on which FIMMWAVE should communicate. FIMMWAVE will connect to the host machine through its domain name via port `<portNo>`.

Once started this way, an external client program or script will be able to communicate with FIMMWAVE on the specified port number - see below on how to write a client program.

11.8.1 Operation

When accessing the FIMMWAVE *Command-Line* remotely, all options that interrupt the flow of the program are disabled. In particular:

1. If a command produced an error, dialog boxes will not appear (thus interrupting the flow of the program) and the execution of batch commands will not be interrupted. FIMMWAVE will return the message “ERROR:<the error message that FIMMWAVE generated>”. The user can then decide what action to take if an error occurred.
2. Option “ask user to continue at each command” is also disabled if sending a message remotely.
3. Option “enable recording window resizing, etc...” is also switched off.

11.9 Scripting

In principle, you can write a client program or script in any computer language that can implement a TCP/IP connection, however we recommend using PYTHON, which we provide on the CD Image.

The python interface has been designed to be very simple, yet very powerful - most of the commands pass through one central function, which can accept arbitrary PYTHON expressions. It automatically returns any results from the PD application as appropriate types (strings, real, complex numbers, lists or 2D arrays thereof, etc).

As an alternative, a sample client program is also provided written in C++ together with the necessary supplementary source files (see §11.9.5).

11.9.1 Installing Python

The option to install PYTHON is given to you upon first running the `setup.exe` on the CD Image. If you chose not to install it at that time then you can obtain PYTHON either by:

- a.) In the `PythonInstall` subdirectory of the Application CD, the windows installer has been supplied. To install PYTHON, run the file: `python16.exe` and follow the online instructions.
- b.) Downloading the installer from www.python.org/1.6.

We also suggest that you install the Scripting Environment PYTHONWIN.

- To do this, run the file win32all-133.exe.

In addition to installing the above programs, you must create **two** new **environment variables**.

- **PYTHONPATH** - This must point to the directory where all PYTHON modules distributed with the application exist.

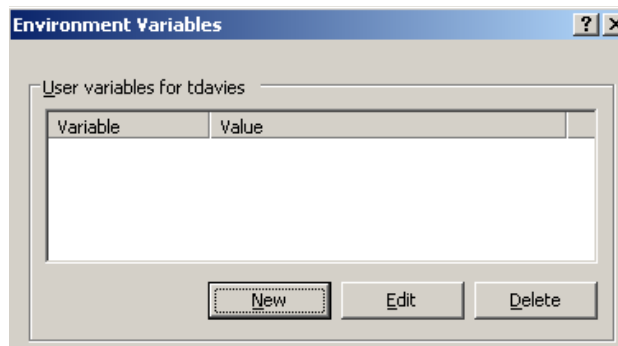
This allows PYTHON to find any extension modules without you having to explicitly reference their paths – *see the PYTHON help files for more details*.

- **PATH** – This must point to a directory where the Python executable is placed.

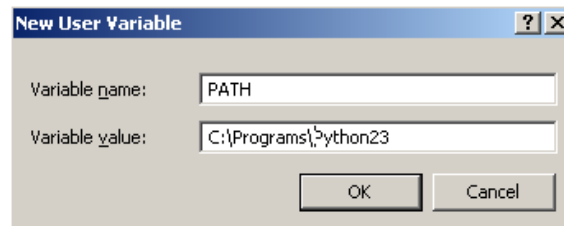
This allows you to run a script automatically from the Application main window.

This is done as follows:

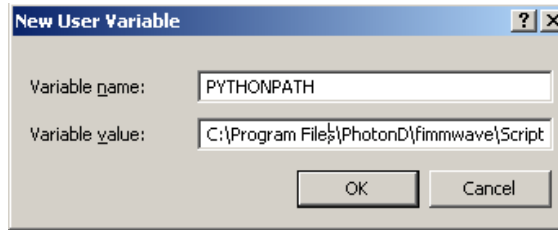
- Select Settings/ControlPanel/System/Advanced/Environment Variables. (MS Windows 2000)



- Click **New**.
- In the **Variable name** box type PATH
- In the **Variable value** box type the path to where the python executable lies. E.g. C:\Programs\Python23



- Click **OK**.
- Click **New**.
- In the **Variable name** box type PYTHONPATH
- In the **Variable value** box type the path to where the python modules lie. E.g. C:\Program Files\PhotonD\firmwave\Scripts



➤ Click **OK**.

See the figure below for reference.

| Variable | Value |
|------------|---|
| path | C:\Programs\Python23 |
| pythonpath | C:\Program Files\PhotonD\firmwave\Scripts |

Note. The PC now has to be restarted.

11.9.2 Example Python Script

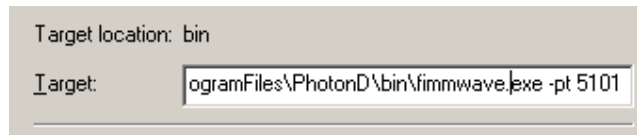
Included in the Scripts directory is an example PYTHON script file called `MMIexample.py`. This script performs the following operations:-

1. Connects to FIMMWAVE (already running with the project open)
2. Sets up a loop that varies the length of the middle section of the device “MMI” and obtains the S11 matrix coefficient for each length

Note that this example assumes that you have the FIMMPROP extension module.

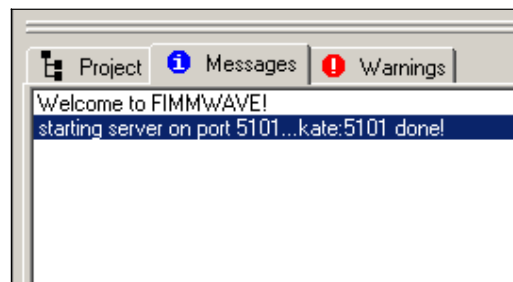
The first thing that needs to be done is to start FIMMWAVE ensuring that it is serving on port 5101. This is simply done:

- If FIMMWAVE is started via a shortcut key on the desktop, then the target (**right click/properties...**) should be set to be “... \FIMMWAVE.exe -pt 5101.” See the figure below for reference.



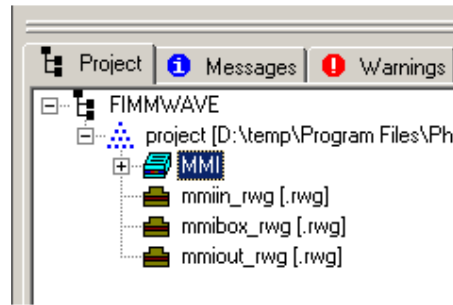
- Alternatively, if you start FIMMWAVE from a DOS box then use the command “firmwave.exe -pt 5101 localhost.”

If FIMMWAVE has been successfully started on port 5101 you will see a message in the Messages window on the main window confirming this.



Before we create and run the script, we need to open the relevant project.

- Open the project file “MMIproj.prj” from the\ScriptsMMIexample directory.



We will now write the Python Script that will connect to the program, calculate the transmission for a variety of different lengths and extract the data.

- Using notepad or some other text editor, open the project file “MMIExample.py” from the ...ScriptsMMIexample directory. You should see the following.

```
from pdPythonLib import *
fimm = pdApp()
fimm.ConnectToApp()
```

The first few lines of MMIExample.py

The first line imports all the necessary functions and classes from the previously written module file (**pdPythonLib.py**) for running the script.

The **pdPythonLib.py** module file contains one class called **pdApp**. This class contains all the functions needed to start, connect to and send messages to FIMMWAVE. A summary of these functions is given in §11.9.3.

The second line in the script:

```
fimm = pdApp()
```

declares an instance of the **pdApp** class and stores the reference to this instance in the variable *fimm*.

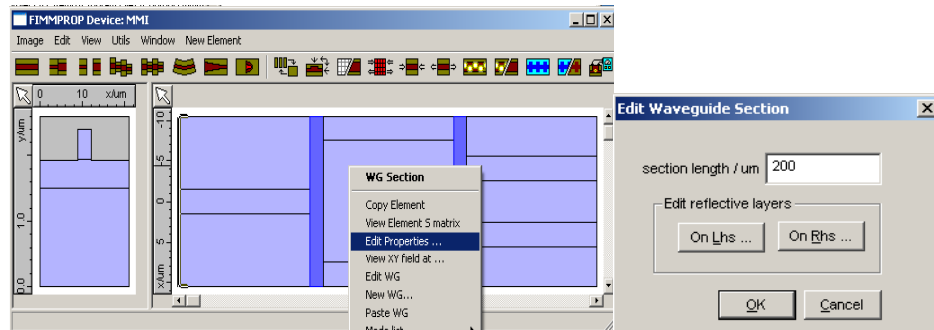
The third line connects to FIMMWAVE using one of the functions given in the table in §11.9.3 assuming that the program is already running and using the default TCP/IP port (which is port no. 5101).

Note. You do not have to always connect to an already running version of the program. There is also command available to start FIMMWAVE. This is **StartApp()**. See the table in §11.9.3 for more details on this function.


What we wish to do now is determine the commands to change the length of the middle section of the MMI device and calculate the scattering matrix. As mentioned in §11.2 most actions performed via the GUI will echo the relevant command in the Command-Line Window.

- Right-click on the centre section of the MMI device and select **Edit Properties...**
- Type 200 in the **section length/um** box.
- Click OK

This will change the length of the middle section of the FIMMPROP device.



Notice that the command `“app.subnodes[1].subnodes[1].cdev.eltlist[3].length=200”` was generated in the *Command-Line*.

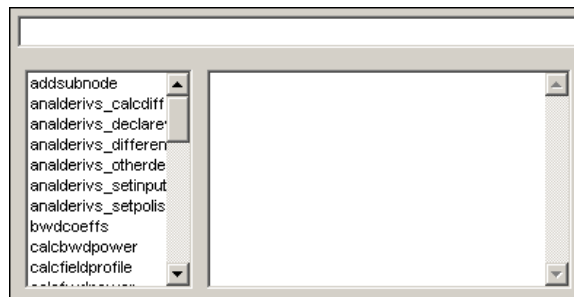
- Similarly, click on  icon on the toolbar to calculate the scattering matrix for this device.

This generates the command `“app.subnodes[1].subnodes[1].update”`

All we need now is the command for extracting the data that we wish to collect. For this example we wish to collect the transmission in the fundamental mode (at the RHS).

Perhaps the easiest way to find this is to use the *command completion box* shown in §11.2.2.

- In the *Command-Line* type `“app.subnodes[1].subnodes[1].”` This is the location of the FIMMPROP device in the *Project Tree*.
- Press the <TAB> key. You should see the following:



- Scroll down to the command `“cdev.”` This allows you to access all properties of the FIMMPROP device.
- Press the <TAB> key again to insert this into the *Command-Line*.
- Append a `“.”` to the command, and press the <TAB> key again to show the *command completion box* again.

We are now interested in the coefficients of the scattering matrix, so we need to insert the command `“smat.”`

- Following the above steps you should complete the following command.

`“app.subnodes[1].subnodes[1].cdev.smat.l[1][1]”`

Where `l[1][1]` denotes the fundamental mode (mode 1 to mode 1- `“[1][1]”`) coefficient for propagation from left to right (`“h”`)

We now have all the commands we need, so lets finish writing a script to loop over a range of lengths and extract this coefficient. This is done using the **Exec** method to send commands to the application.

➤ In your text editor write the following lines:

```
for i in range (100,700,50):
```

```
    fimm.Exec("app.subnodes[1].subnodes[1].cdev.eltlist[3].length={i}")
```

```
    fimm.Exec("app.subnodes[1].subnodes[1].update")
```

```
    a = fimm.Exec("app.subnodes[1].subnodes[1].cdev.smat.lr[1][1]")
```

```
    print a
```

Note

- 1) You need to include the indents for all commands in the “for loop”.
- 2) Also note that the user can also conveniently embed any python expression in the string by enclosing it in {} as shown above. This embedding procedure is used in the first line in the loop.
- 3) Later within the loop the following command:-

```
    a = fimm.Exec("app.subnodes[1].subnodes[1].cdev.smat.lr[1][1];")
```

assigns the complex number returned by *fimm.Exec* into the PYTHON variable *a*. Note that the returned variable “a” is a complex number.

➤ Then, outside the loop (I.e. with no indent), write

```
raw_input(">>>>")
```

This will wait for the user to press return on the keyboard before continuing

➤ Next, write

```
del fimm
```

This will delete the connection and close the script.

You should have the following:

```
from pdPythonLib import *
fimm = pdApp()
fimm.ConnectToApp()

for i in range (100,700,50):
    fimm.Exec("app.subnodes[1].subnodes[1].cdev.eltlist[3].length={i}")
    fimm.Exec("app.subnodes[1].subnodes[1].update")
    a = fimm.Exec("app.subnodes[1].subnodes[1].cdev.smat.lr[1][1]")
    print a

raw_input(">>>>")
del fimm
```

We can now run the script.

- From the FIMMWAVE Main Window select **Scripts/Run a script...**
- Select MMIEExample.py

After a short time you should see the following:

```

c:\D:\Programs\Python23\python.exe
Attempting to connect to application on TCP/IP Port No. 5101
(0.09545079992223+0.478466926972j)
(-0.579250724867+0.0163821805621j)
(0.153777022081-0.441072892826j)
(0.20386112289+0.4670343244j)
(-0.600474948199-0.052169864344j)
(0.429468315286-0.603964933894j)
(0.505972323342+0.756338289869j)
(-0.896654318555+0.383256768734j)
(-0.0864984206396-0.746073745941j)
(0.568976785034+0.245516259731j)
(-0.427235703777+0.290732540683j)
(0.0202487412426-0.452860603689j)
>>>> -

```

Notice that the script is waiting for action from the user before exiting.

Perhaps a more convenient way to extract data is to use the output log file described in §11.2.4

- From the FIMMWAVE main window select **Options/Command-Line...**
- In the output logfile box define the name and convenient location of an output log file. E.g. C:\temp\output.txt
- Click

Now all output will be written to this file. For convenience, we are going to add one command to our python script.

- In the text editor, add the line
`“fimm.Exec(“app.subnodes[1].subnodes[1].cdev.eltlist[3].length”)`

directly after the line

`“fimm.Exec(“app.subnodes[1].subnodes[1].cdev.eltlist[3].length={i}”)`

- Now run the script again.

An output logfile should be generated and should contain the following data.

```

1.00
(0.0954507999222985,0.478466926971846)
1.50
(-0.5792507248672749,0.01638218056210624)
2.00
(0.1537770220807167,-0.4410728928259971)
2.50
(0.2038611228895896,0.4670343243999329)
3.00
(-0.6004749481989091,-0.05216986434396303)
3.50
(0.4294683152863829,-0.6039649338944343)
4.00
(0.5059723233424911,0.7563382898687802)
4.50
(-0.8966543185545255,0.3832567687335584)
5.00
(-0.08649842063963451,-0.7460737459406878)
5.50
(0.5689767850338615,0.2455162597307719)
6.00
(-0.4272357037770407,0.2907325406827299)
6.50
(0.02024874124257894,-0.4528606036885614)

```

11.9.3 pdApp Class Member Summary

This section contains a summary of the four member functions of the `pdApp` class used to send messages to the Photon Design application.

StartApp

```
StartApp( pathname [, <portNo> ] )
```

This function starts and connects to the Photon Design application using the executable file that is given in *pathname*. Upon execution, the current directory becomes the directory where the Photon Design application is located. The optional parameter *portNo* specifies which port to use. The default value for *portNo* is 5101. If *portNo* is unavailable, an alternative port is automatically selected.

Note. In future versions, if you have a network license, it will be accessible as a computation server from anywhere on your LAN.

ConnectToApp

```
ConnectToApp( [hostname] [,portNo] )
```

This function connects to an already running the Photon Design application that is serving on *hostname* on the port given by *portNo*. Both parameters are optional with the default *hostname* being “localhost” and the default *portNo* being 5101. If the port is unavailable, the function tries to connect with a Photon Design application that might be running on the next available port. If successful, the function returns an empty string. Upon failure, it returns a string containing an error message.

Note. In future versions, if you have a network license, it will be accessible as a computation server from anywhere on your LAN.

AddCmd

```
AddCmd( commStr [,varList] )
```

This command appends *commStr* to the current list of commands that are waiting to be sent to the Photon Design application (which are done so via a call to *Exec* - see below).

The variable, *commStr* can be any valid Photon Design application *Command-Line* expression, with the addition that any expression encapsulated in {} is first evaluated using the PYTHON interpreter (unless the {} are themselves encapsulated in “”).

For example the following lines:-

```
from pdPythonLib import *
f = pdApp()
f.ConnectToApp()
a = 1.103
f.AddCmd("MMI.lambda = {a}")
a = 1.55
f.Exec()
```

will set *MMI.lambda* equal to 1.103.

The optional parameter *varList* is a list of PYTHON expressions that are resolved before being embedded in *commStr*. The function will sequentially replace every % symbol encapsulated within {} with the next evaluated expression in the list. Complicated expressions can also be included in *varList*. For example:

```
from pdPythonLib import *
def myfunc:
    a = 0.103
    b = 1
    f = pdApp()
    f.ConnectToApp()
    f.AddCmd("MMI.lambda = {% + %}", [a,b])
```

This will first resolve a, then b and insert them into the string before resolving the whole expression within the {}.

varList is also useful since the pdApp class cannot resolve PYTHON expressions that involve local or module variables. For example:

```
from pdPythonLib import *
def myfunc():
    a = 1.103
    f = pdApp()
    f.ConnectToApp()
    f.AddCmd("MMI.lambda = {a}")
```

will not convert {a} to 1.103 since the variable a goes out of scope upon execution of *AddCmd*. However:

```
from pdPythonLib import *
def myfunc:
    a = 1.103
    f = pdApp()
    f.ConnectToApp()
    f.AddCmd("MMI.lambda = {%}", [a])
```

will work since a is now one of the members of *varList* which gets parsed within *AddCmd*.

Exec

```
Exec( commStr [,varList] )
```

This function sends commands to the Photon Design application. Following the evaluation of all expressions encapsulated in {}, *commStr* is appended to the current list of commands and then the whole list is sent to the Photon Design application. The function then receives the response from the Photon Design application. It returns this response as any of the following basic types: *a string, a floating point number, an integer or a complex number*. It can also return 1d arrays or 2d arrays of any of these basic types. The array indices have a one-to-one correspondence with those in the Photon Design application.

For example if in FIMMWAVE one were to type the following command

```
project.MMI.cdev.smat.lr
```

into the Command Line Window, FIMMWAVE might return the following:-

```

lr[1][1]      (0.468,0.00745)
lr[1][2]      (0.216,0.0698)
lr[2][1]      (0.872,0.0003)
lr[2][2]      (0.134,0.00629)

```

If the equivalent PYTHON command is sent (assuming the user has an instance of FIMMWAVE referenced by the PYTHON variable *f*):-

```
a = f.Exec("project.MMI.cdev.smat.lr")
```

then *a* will be a **2d array** with the following values:-

```

a[1][1] = (0.468 + 0.00745j)
a[1][2] = (0.216 + 0.0698j)
a[2][1] = (0.872 + 0.0003j)
a[2][2] = (0.134 + 0.00629j)

```

Finally, mixed-type lists can also be returned. For example:

```

f.AddCmd("myfile_rwg.evlist.mlp.evstart")
f.AddCmd("myfile_rwg.evlist.mlp.evend")
f.AddCmd("myfile_rwg.evlist.svp.lambda")
f.AddCmd("myfile_rwg.evlist.mlp.nx")
[start,end,wavelength,nx,nslice]= f.Exec("myfile_rwg.nslice")

```

The last command in this list sets each of the PYTHON variables in the list to the returned value of the *AddCmd* line.

Note the optional parameter *varList* (described above) can also be used with the *Exec* command. This allows the calling of *Exec* from within local functions of user-defined modules and classes. See *AddCmd* for more details. The table below is a summary of these four commands.

| Function Name | Parameters | What it Does | Return Value |
|---------------|--|--|---|
| StartApp | pathname: (No default), portNo: (default = 5101) | Starts the Photon Design application on the path given by pathname and connects to it using the port given by portNo. If this port is unavailable, it uses the next available port | An empty string if successful or a string containing an error message if it has failed at some stage. |
| ConnectToApp | Hostname: (default = "localhost"), portNo: (default = 5101) | Connects to an already running version of the Photon Design application given by the hostname and serving on the port given by portNo. | An empty string if successful or a string containing an error message if it has failed at some stage. |
| AddCmd | commStr: (No default) , varList: (default = []) | Firstly, this function evaluates any python expressions embedded in commStr. It then appends the command given in commStr to the current list of commands waiting to be executed (by a call to Exec) | None |
| Exec | commStr: (No default), varList: (default = []) | Firstly, this function evaluates any python expressions embedded in commStr. It then appends this to the current list of commands before sending this entire list to FIMMWAVE. Finally it receives the response from the Photon Design application and flushes the | This function returns one the following basic types: a <i>complex number</i> , an <i>integer</i> , a <i>floating point number</i> or a <i>string</i> . However, it can also return a <i>1d array</i> or a <i>2d array</i> of any of those basic types or a <i>mixed-type list</i> containing any of |

| | | | |
|--|--|---------------|--------------|
| | | command list. | these types. |
|--|--|---------------|--------------|

11.9.4 Python Client

PdAppclient.py can be used as a PYTHON interface to the scripting engine of all Photon Design (PD) applications (such as FIMMWAVE). This makes it possible to communicate with a running executable remotely via TCP/IP and use the power of the fully-fledged scripting language to write complex FIMMWAVE routines with relative ease.

To run this client program from the command prompt, simply open the Scripting Environment PYTHONWIN. Open and run the script PdAppclient.py giving as arguments:

- Port number - the TCP/IP port number
- Hostname – the name or IP address of the machine on which the PD application is running.

Alternatively, navigate to the directory that contains PdAppclient.py and type:

```
python pdAppclient.py <portNo> <hostname>
```

Another example of what can be achieved using PYTHON clients is given at the following website: www.python.org/workshops/1997-10/proceedings/beazley.html.

Please note we are not able to provide technical support in debugging your client programs or scripts (unless technical support is required on the syntax or the *Command-Line* features of FIMMWAVE commands).

11.9.5 Using C++

As an alternative to PYTHON, we provide on the CD Image a c++ client.

There is additional documentation on writing client programs in the file examples\SourceCode\pdAppClient.cpp. If you wish to use another language other than C++ or PYTHON, we suggest you study the C++ example carefully.

In order to send messages to FIMMWAVE, you should use the *Client* class provided with the distribution (see client.cpp, client.h). The console skeleton program (pdAppClient.cpp) is provided showing you how to use it. To compile it, you will need all the other source files provided in the distribution. The client program is self-explanatory and provides a *Command-Line* interface for sending commands remotely to a running FIMMWAVE program. We suggest that you first familiarise yourself with this program by compiling it and running it. You can then use it as a starting point for writing your own more complex routines.

The Client class

The client class is quite simple to use: it only has 2 members that need explanation

```
bool Connect(int portNo, const char* serverName=NULL);
const char* sendMessage(const char* command);
```

The first simply initialises the client and connects via TCPIP port “portNo” to a running FIMMWAVE program situated on the computer with IP server name “serverName”.

The second sends the command “command” to FIMMWAVE. FIMMWAVE will then return the output once execution of the command (or command batch) is completed.

The return string consists of a series of lines of the following format:

```
RETVAL: output string
```

or

```
ERROR: error message
```

The first will be appended for every command sent that makes FIMMWAVE return a non-empty string. The return string is exactly the same as the one written to the output pane in the FIMMWAVE command window. The second will be appended whenever an error occurs.