CHAPTER – 3

LABVIEW FOR PC BASED POLARIMETER
3.1 INTRODUCTION TO LabVIEW

In the earlier chapter, the design (optical and electronic) and development of PC based polarimeter based on Malus law is presented.

As discussed in section 2.3, storing the values of \( I_1 = I \) (reading of \( S_1 \) with out the sample), calculation of optical rotation (\( \theta \)) from \( I = I_1 \cos^2 \theta \), and maintaining the intensity of \( I_1 \) as constant irrespective of the sample and comparing the \( S_2 \) readings to the values given in the below graph (Fig 2.9) can be done by PC using LabVIEW software, a virtual instrumentation package. This chapter now deals with the development of the above software (LabVIEW) required for PC based polarimeter using Malus law. Software developed in the study is also included in this chapter.

3.1a. VIRTUAL INSTRUMENTATION

Virtual instrumentation is the use of customizable software and modular measurement hardware to create user-defined measurement systems, called virtual instruments.

Virtual instruments are computer programs that interact with real world objects by means of sensors and actuators and implement functions of real or imaginary instruments. The sensor is usually a simple hardware that acquires data from the object, transforms it into electric signals and transmits to the computer for further processing. Simple virtual measuring instruments just acquire and analyze data, but more complex virtual instruments communicate with objects in
both directions. The outgoing signals execute probing and control by actuators.

Real world signals are of analogue nature, while a computer is a digital instrument; therefore the computer needs also interpreters – analogue-to-digital and digital-to-analogue converters for communication with the object under investigation. ADC and DAC boards that implement this function in inexpensive systems are usually placed inside the computer. Compact external ADC/DAC converters with USB interface are also becoming popular.

‘Traditional’ or ‘natural’ instrumentation systems are made up of pre-defined hardware components, such as digital millimeters’ and oscilloscopes that are completely specific to their stimulus, analysis, or measurement function. Because of their hard-coded function, these systems are more limited in their versatility than virtual instrumentation systems. The primary difference between ‘natural’ instrumentation and virtual instrumentation is the software component of a virtual instrument. The software enables complex and expensive equipment to be replaced by simpler and less expensive hardware; ex: Analog to digital converter can act as a hardware complement of a virtual oscilloscope.

Leveraging commercially available technologies, such as the pc and the analog to digital converter, virtual instrumentation has grown
significantly since its inception in the late 1970s. Additionally, software packages like ‘National Instruments’ LabVIEW and other graphical programming languages helped grow adoption by making it easier for non-programmers to develop systems.

3.1b. VISUAL PROGRAMMING LANGUAGE

A visual programming language (VPL) [49] is any programming language that lets users specify programs by manipulating program elements graphically rather than by specifying them textually. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols. Most VPLs are based on the idea of “boxes and arrows,” that is, boxes or circles or bubbles, treated as screen objects, and connected by arrows, lines or arcs.

VPLs may be rather classified, according to the type and extent of visual expression used, into icon-based languages, for-based languages, and diagram languages. Visual programming environments provide graphical of iconic elements which can be manipulated by users in an interactive way according to some specific spatial grammar for program construction.

Current developments try to integrate the visual programming approach with dataflow languages to either have immediate access to the program state resulting in online debugging (i.e. LabVIEW) or automatic program generation and documentation (i.e. visual paradigm). Dataflow languages also allow automatic parallelization, which is likely to become one of the greatest programming challenges of the future.
3.1c. LabVIEW

LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) \[^{[50]}\] is a platform and development environment for a visual programming language from National Instruments. The graphical language is named “G”. Originally released for the Apple Macintosh in 1986, LabVIEW is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms including Microsoft Windows, various flavors of UNIX, Linux, and Mac OS.

3.1d. Dataflow programming

The programming language used in LabVIEW, called “G”, is a dataflow language \[^{[51]}\]. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

The dataflow completely defines the execution sequence, and that can be fully controlled by the programmer. Thus, the execution sequence of the LabVIEW graphical syntax is as well-defined as with any textually coded language such as C, Visual, BASIC, etc. Furthermore, LabVIEW does not require type definition of the variables; the wire type is defined by the data-supplying node.
LabVIEW supports polymorphism in that wires automatically adjust to various types of data.

Screenshot of a simple LabVIEW program that generates, synthesizes, analyzes and displays waveforms, showing the block diagram and front panel is shown in the fig 3.1. Each symbol on the block diagram represents a LabVIEW subroutine (subVI) which can be another LabVIEW program or a LV library function.
This VI continuously generates two signals: a pure sine wave of variable frequency and amplitude and a white noise signal of variable amplitude. The noise is then added to the pure sine. The sine wave with and without the noise are then shown in a time domain graph. Additionally, an FFT is calculated for both signals and the results are then shown in the frequency domain graph. Note that the square shaped functions are subroutines in the form of subvbs.

Fig 3.1 Screen shot of a simple LabVIEW program
3.1e. Graphical programming

LabVIEW ties the creation of user interfaces (called front panels) into the development cycle. LabVIEW programs/subroutines are called virtual instruments (Vis). Each VI has three components: a block diagram, a front panel and a connector pane. The latter may represent the VI as a sub VI in block diagrams of calling Vis. Controls and indicators on the front panel allows an operator to input data into or extract data from a running virtual instrument. However, the front panel can also serve as a programmatic interface. Thus a virtual instrument can either be run as a program, with the front panel serving as user interface, or, when dropped as a node onto the block diagram, the front panel defines the inputs and outputs for the given node through the connector pane. This implies each VI can be easily tested before being embedded as a subroutine into a larger program.

The graphical approach also allows non-programmers to build programs by simply dragging and dropping virtual representations of the lab equipment with which they are already familiar. The LabVIEW programming environments, with the included examples and the documentation, makes it simpler to create small applications. This is benefit on one side but there is also a certain danger of underestimating the expertise needed for good quality “G” programming. For complex algorithms or large-scale code it is important that the programmer possess an extensive knowledge of the special LabVIEW syntax and the topology of its memory management. The most advanced labVIEW syntax and the topology of its memory management. The most advanced LabVIEW development systems offer the possibility of building stand-alone applications. Furthermore, it is possible to create distributed applications which
communicate by a client/server scheme, and thus is easier to implement due to the inherently parallel nature of G-code.

3.1f. Benefits

One benefit of LabVIEW over other development environments is the extensive support for accessing instrumentation hardware. Drivers and abstraction layers for many different types of instruments and buses are included or are available for conclusion. These present themselves as graphical nodes. The abstraction layers offer standard software interfaces to communicate with hardware devices. The provided driver interfaces save program development time. The sales pitch of National Instruments is, therefore, that even people with limited coding experience can write programs and deploy test solutions in reduced time frame when compared to more conventional or competing systems. A new hardware driver topology (DAQ mxBase), which consists mainly of G-coded components with only a few register calls through NI Measurement Hardware DDK (Driver Development Kit) functions, provides platform-independent hardware access to numerous data acquisition and instrumentation devices. The DAQmxBase driver is available for LabVIEW on Windows, MacOSX and LINUX platforms.

In terms of performance, LabVIEW includes a compiler that produces native code for the CPU platform. The graphical code is translated into executable machine code by interpreting the syntax and by compilation. The labVIEW syntax is strictly enforced during the editing process and compiled into the executable machine code when requested to run or upon saving. In the latter case, the executable machine code when requested to run or upon saving. In the latter case,
the executable and the source code are merged into a single file. The executable runs with the help of the LabVIEW run-time engine, which contains some precompiled code to perform common tasks that are defined by the G language. The run-time engine reduces compile time and also provides a consistent interface to various operating systems, graphics systems, hardware components, etc. The run-time environment makes the code portable across platforms. Generally, LV code can be slower than equivalent compiled C code, although the differences often lie more with program optimization the inherent execution speed.

Many libraries with a large number of functions for data acquisition, signal generation, mathematics, statistics, signal conditioning, analysis, etc. along with numerous graphical interface elements are provided in several LabVIEW package options.

The LabVIEW professional development system allows creating stand-alone executables and the resultant executable can be distributed an unlimited number of times. The run-time engine and its libraries can be provided freely along with the executable.

A benefit of the LabVIEW environment is the platform independent nature of the G-code, which is (with the exception of a few platform-specific functions) portable between the different LabVIEW systems for different operating systems (windows, Mac OSX and Linux).
3.2 ALGORITHMS FOR PC BASED POLARIMETER

As discussed in sections 2.2, 2.3, there are 3 tasks to be performed by the PC.

➢ Task 1: Optical rotation

Read the value of sensor $S_1$, and calculate the "optical rotation".

➢ Task 2: Dextro/Leavo rotation

Read the value of sensor $S_2$, and calculate the Dextro/Laevo nature of the sample.

➢ Task 3: Intensity control

Read the value of sensor $S_3$, and generate the error correcting signal for intensity control circuit.

The above tasks are implemented by PC using the LabVIEW software, for which the algorithms are given below.

3.2a. Optical rotation

1. Read $S_1$

2. Take the average of $S_1$ over 200000 samples per second (to reduce the noise).

3. Subtract the offset of $S_1$ from the above $S_1$ average.(This is the reading of $S_1$ which is free from noise and dark errors)

4. Enter the $V_0$ value.

5. Angle of rotation $= \cos^{-1} \sqrt{S_1/V_0}.$
6. Display angle of rotation.

3.2b. Dextro/Laevo rotation

1. Read $S_2$.

2. Take the average of $S_2$ over 200000 samples per second (to reduce the noise).

3. Subtract the offset of $S_2$ from the above $S_2$ average. (This is the reading of $S_2$ which is free from noise and dark errors)

4. Enter $\text{REF1} = 3.4 \text{mv}$

5. Compare the $S_2$ value with $\text{REF}$

6. If $S_2 > \text{REF1}$, Display “DEXTRO”.

   If $S_2 < \text{REF1}$, Display “LAEVO”.

3.2c. Intensity control

1. Read $S_3$.

2. Take the average of $S_3$ over 200000 samples per second (to reduce the noise).

3. Subtract the offset of $S_3$ from the above $S_3$ average. (This is the reading of $S_3$ which is free from noise and dark errors)

4. Enter $\text{REF2} = 10 \text{mv}$

5. Take $\text{REF2 (10mv)} - S_3 = x$
6. If \( x > 0 \): \( y = y + 1 \), or if \( x < 0 \): then \( y = y - 1 \), if \( x = 0 \), then \( y = y \) (where \( y \) is a counter).

7. Multiply ‘\( y \)’ with the LSB of DAQ card (152uV) and send out.

3.3. FLOWCHARTS FOR PC BASED POLARIMETER

The flowcharts for the above tasks are given below.
3.3a. Angle of rotation

START

Read S1

S1 Average = \( \frac{\sum S1}{200000} \)

S1 Reading = S_{1\text{avg}} - S_{1\text{offset}}

Enter V0 value

Angle of Rotation \( \theta = \cos^{-1} \frac{S1}{V0} \)

Display \( \theta \)
3.3b. Dextro/Laevo rotation

START

Read S2

S2 Average = \( \frac{\sum S2}{200000} \)

S2 Reading = S2avg - S2offset

Enter REF=3.4mv

Is S2>REF

No, Display 'LAEVO'

Yes, Display 'DEXTRO'
3.3c. Intensity control

START

Read S3

S3 Average = \( \frac{\sum S3}{200000} \)

S3 Reading = S3 avg - S3 offset

Enter REF2 = 10mv

\[ X = \text{REF2} - S3 \]

If \( x > 0, y = y + 1 \), if \( x < 0, y = y - 1 \)

If \( x = 0, y = y \)

\[ Y = Y \times \text{LSB of DAQ(152uv)} \]

OUT Y
3.4 LabVIEW SOFTWARE FOR PC BASED POLARIMETER

The LabVIEW software for PC based polarimeter is given in the fig 3.2 in terms of blocks. This is called the 'block diagram'. Each symbol on the block diagram represents a LabVIEW subroutine (subVI) which can be another LabVIEW program or a LV library function which performs a specific function. The meaning of each block is discussed below. The front panels of PC based polarimeter are presented in Fig 3.1a and 3.1b.
ANGLE OF ROTATION

Result: angle of rotation (DEG)

21.3

offset (mv)

-0.06

Vo (mv)

8.2

s10/p (mv)

7.1

Fig 3.1a
Fig 3.2 labVIEW software for PC based polarimeter
3.5 DESCRIPTION OF BLOCKS/UNITS IN THE SOFTWARE

3.5a. DAQ Assistant

The block diagram of DAQ assistant is shown in the Fig 3.3. It creates, edits, and runs tasks using NI-DAQmx. In NI-DAQmx, a task is a collection of one or more channels, timing, triggering, and other properties that apply to the task itself. Conceptually, a task represents a measurement or generation you want to perform. For example, you can create a task to measure temperature from one or more channels on a DAQ device.

When you place this VI (virtual instrument) on the block diagram, the DAQ Assistant launches to create a new task. After you create a task, you can double-click the DAQ Assistant VI to edit that task. For continuous measurement or generation, place a while loop around the DAQ Assistant VI. The following table list out various inputs and outputs available for the above block.

Block Diagram Inputs:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Contains samples to write to the task. data is an output for measurement tasks and an input for analog and digital output tasks. data does not appear for counter output tasks.</td>
</tr>
<tr>
<td>error in</td>
<td>Describes error conditions that occur before this Express VI runs.</td>
</tr>
<tr>
<td>number of samples</td>
<td>Specifies the number of samples to acquire or generate for each channel in a finite task. For continuous tasks, NI-DAQmx uses this value to determine the buffer size.</td>
</tr>
</tbody>
</table>
This input does not appear for all channel types and sample timing types.

Specifies the **sampling rate** in samples per channel per second. This input does not appear for some channel types and sample timing types. If you use an external source for the Sample Clock, set this input to the maximum expected rate of that clock.

Specifies to stop the task and release device resources when this Express VI completes execution. For continuous tasks, this input is FALSE by default, meaning the task continues to run until the application stops. To stop the task so you can use the device again in the same application, wire this input to the same stop control you wire to the conditional terminal of the while loop. For single-point and finite tasks, this input is TRUE by default, meaning the task stops after all samples are acquired. To optimize single-point performance when using this Express VI in a loop, wire this input to the same stop control you wire to the conditional terminal of the while loop.

Specifies the amount of time in seconds to wait for the VI to read or write all samples. This VI returns an error if the time elapses. For input operations, the VI also returns any samples read before the time elapses. The default timeout is 10 seconds. If you set **timeout** to -1, the VI waits indefinitely. If you set **timeout** to 0, the VI tries once to read or write the samples and returns an error if unsuccessful. NI-DAQmx performs a timeout check only...
if the VI must wait to read or write samples. This input
does not appear for all channel types and sample timing
types.

Block Diagram Outputs:

Parameter Description

Contains samples read from the task. **data** is an output for
measurement tasks and an input for analog and digital
output tasks. **data** does not appear for counter output
tasks.

Contains error information. If **error in** indicates that an
error occurred before this Express VI ran, **error out**
contains the same error information. Otherwise, it
describes the error status that this Express VI produces.

Indicates whether the task stopped. The task stops if the
**stop** input is set to TRUE or an error occurs. This output
appears for continuous or hardware-timed single-point
tasks only.

Contains a reference to the task after this VI completes
execution. Wire this output to other NI-DAQmx VIs to
perform other operations with this task.

3.5b. Split Signals

Splits two or more signals into component signals. [See the Fig 3.4
for block diagram of ‘split signals’]. It consists of the following inputs
and outputs.
1.1.

Fig. 3.3 DAQ assistant
Fig 3.4 Split signals

Fig 3.5 Convert from dynamic data
Block Diagram Inputs:

**Combined signal** contains a signal you want to split into component signals.

Block Diagram Outputs:

**Output signal** returns the component signals.

3.5c. Convert from Dynamic Data [Fig 3.5]

Converts the dynamic data type to numeric, Boolean, waveform, and array data types for use with other VIs and functions.

Block Diagram Inputs:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Channel</strong></td>
<td>Specifies the channel from which you want to retrieve the data. The value you wire to this input overrides the value you set in the configuration dialog box.</td>
</tr>
<tr>
<td><strong>Dynamic Data Type</strong></td>
<td>Contains the input signal formatted as dynamic data.</td>
</tr>
</tbody>
</table>

Block Diagram Outputs:

<table>
<thead>
<tr>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the output signal converted into an array of numeric values. You also can configure the Convert from Dynamic Data Express VI to return the following outputs:</td>
</tr>
</tbody>
</table>

- **Scalar**—Returns the output signal converted into a numeric value.
• **2D**—Returns the output signal converted into a 2D array of numerics.

• **Waveform**—Returns the output signal converted into a waveform.

### 3.5d. Add Array Elements [Fig 3.6]

Returns the sum of all the elements in numeric array. The connector pane displays the default data types for this polymorphic function.

**Block Diagram Inputs:**

**Numeric array** can have any number of dimensions.

**Block Diagram Outputs:**

### 3.5e. Sum

It is of the same data type and structure as the elements in **numeric array**.
Fig 3.6 Add array elements

NUMERIC ARRAY

\( \sum \)

SUM

Fig 3.7 Divide

\( \frac{X}{Y} \)
3.5f. Divide [Fig 3.7]

Computes the quotient of the inputs. If you wire two waveform values or two dynamic data type values to this function, **error in** and **error out** terminals appear on the function.

Block Diagram Inputs:

- x can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on.

- y can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on.

Block Diagram Outputs:

- x/y is a double-precision, floating-point number if both x and y are integers. In general, the output type is the widest representation of the inputs if the inputs are not integers or if their representations differ.

3.5g. Multiply [Fig 3.8]

Returns the product of the inputs. If you wire two waveform values or two dynamic data type values to this function, **error in** and **error out** terminals appear on the function. The connector pane displays the default data types for this polymorphic function.

Block Diagram Inputs:

- x can be a scalar number, array or cluster of numbers, array of cluster of numbers, and so on.
y can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on.

Block Diagram Outputs:

x*y is the product of x multiplied by y.

3.5h. Subtract [Fig 3.9]

Computes the difference of the inputs. If you wire two waveform values or two dynamic data type values to this function, error in and error out terminals appear on the function. Subtracting two time stamp values yields a numeric value (difference in time), and subtracting a numeric value from a time stamp value yields a time stamp. You cannot subtract a time stamp value from a numeric value.

Block Diagram Inputs:

x can be a scalar number, array or cluster of numbers, array of clusters of numbers, a time stamp, and so on.

y can be a scalar number, array or cluster of numbers, array of clusters of numbers, a time stamp, and so on.

Block Diagram Outputs:

x-y is the difference between x and y.
Fig 3.8 Multiply

Fig 3.9 Subtract
3.5i. Formula Node [Fig 3.10]

Evaluates mathematical formulae and expressions similar to C on the block diagram. The following built-in functions are allowed in formulas: abs, acos, acosh, asin, asinh, atan, atan2, atanh, ceil, cos, cosh, cot, csc, exp, expm1, floor, getexp, getman, int, intrz, In, Inpl, log, log2, max, min, mod, pow, rand, rem, sec, sign, sin, sinc, sinh, sqrt, tan, tanh.

Note: The Formula Node accepts only the period (.) as a decimal separator. The node does not recognize localized decimal separators.

3.5j. Feedback Node [Fig 3.11]

In For Loops and While Loops, transfers values from one loop iteration to the next.

Note: you can place a Feedback Node only in a For Loop or while Loop.

The node appears automatically in the loop if whenever you wire the output of a subVI, function, or group of subVIs or functions to the input of that same VI, function or group. You can replace the Feedback Node with a shift register or replace shift registers with a Feedback Node.
Fig 3.10 Formula node

Fig 3.11 Feedback node
3.5k. Greater? [Fig 3.12]

Returns TRUE if \( x \) is greater than \( y \). Otherwise, this function returns FALSE. You can compare an array or cluster of a data type to a scalar of the same data type and produce an array of cluster of Boolean values.

Block Diagram Inputs:

\( x \) and \( y \) must be of the same type.

Block Diagram Outputs:

\( x > y? \) Returns the Boolean result of the operation. When you compare arrays, \( x > y? \) is a scalar in Compare Aggregates mode and a Boolean array in Compare Elements mode (default). You can set this function to compare Aggregates or Compare Elements mode.

3.5L. Select [Fig 3.13]

Returns the value wired to the \( t \) input or \( f \) input, depending on the value of \( s \). If \( s \) is TRUE, this function returns the value wired to \( t \). If \( s \) is FALSE, this function returns the value wired to \( f \).
Fig 3.12 Greater?

Fig 3.13 Select
Block Diagram Inputs:

\[ t \] is the value that this function returns if \( s \) passes a TRUE value, \( t \) and \( f \) must be of the same type, but they can have different numeric representations.

\( s \) determines whether the function returns the value of \( t \) or \( f \) in \( s \)? \( t:f \), if you wire an error cluster to \( s \) and an error occurs, the error cluster passes a TRUE value to the function. Otherwise, the error cluster passes a FALSE value to the function.

\( f \) is the value that this function returns if \( s \) passes a FALSE value, \( t \) and \( f \) must be of the same type, but they can have different numeric representations.

Block Diagram Outputs:

\( s? t:f \) is the value wired to \( t \) if \( s \) is TRUE, \( s? t:f \) is the value wired to \( f \) if \( s \) is FALSE.

3.5m. Formula [Fig 3.14]

Uses a calculator interface to create mathematical formulas. You can use this Express VI to perform most math functions that a basic scientific calculator can compute.
Block Diagram Inputs:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(XI-X8)</td>
<td>are the input values for the formula you specified in the configuration dialog box.</td>
</tr>
<tr>
<td>XI</td>
<td></td>
</tr>
<tr>
<td>error in</td>
<td>(no) Describes error conditions that occur before this VI or function runs.</td>
</tr>
<tr>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>

Block Diagram Outputs:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Returns the resulting data based on the configuration of the Express VI.</td>
</tr>
<tr>
<td>error out</td>
<td>Contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces.</td>
</tr>
</tbody>
</table>

3.5n. While Loop [Fig 3.15]

Repeats the sub diagram inside it until the conditional terminal, an input terminal, receives a particular Boolean value. The Boolean value depends on the continuation behavior of the While Loop. Right-click the conditional terminal and select Stop if True or Continue if True from the shortcut menu. You can also wire an error cluster to the conditional terminal, right-click the terminal, and select Stop on Error or Continue while Error from the shortcut menu. The While Loop always executes at least once. The iteration (i) terminal provides the current loop iteration count, which is zero for the first iteration.
If you select a While Loop on the Execution Control Express VIs and Structures palette and place it on the block diagram, a stop button also appears on the block diagram and is wired to the conditional terminal. If you select a While Loop on the Structures palette and place it on the block diagram, a stop button does not appear.

After you create a While Loop, you can use shift registers to pass values from one iteration to the next. If you wire an array to a While Loop, you can read and process every element in that array by enabling auto-indexing.
error in ~ (no error) $F(X \ Y)$

Fig 3.14 Formula

![Diagram of a component with inputs $x_1$ and $x_2$, and outputs 'result' and 'error out'.](image)

Fig 3.15 While