Echoview Tutorial:
Introduction to the Code operator

Contents

Overview

This tutorial is optimized for Echoview 13.1 and provides an introduction to the Code operator, a virtual variable in Echoview.

This tutorial is not intended as a comprehensive user manual.

Further information on Echoview tools and topics can be found in the latest version of the Echoview help file. This can be viewed online and is installed with Echoview. Press F1 when using Echoview to open the help file and read context-sensitive information.

Throughout this tutorial further reading is referred to the Echoview help file or other Echoview training materials: https://echoview.com/support/tutorials/

Prerequisites

This tutorial assumes you have Echoview installed, and the following skills and knowledge:

- Familiarity with the basic operation of Echoview. We strongly recommend that you complete the "Getting Started" tutorial before beginning this tutorial.
- Familiarity with a supported Microsoft Windows™ operating system. For more information refer to the Echoview Help file page: Computer requirements.
- A basic understanding of echosounding techniques and hydroacoustic surveying.

Echoview modules

This tutorial requires a license with the Essentials and Advanced Operators modules.

If you do not yet have access to Echoview licenses with these modules, please contact info@echoview.com to request an evaluation license.

Contacting Echoview

For assistance with this tutorial please contact support@echoview.com.

Troubleshooting

The files for a tutorial are typically located in C:\Echoview Software\Tutorials\. The file path may be different if you chose to install the tutorials elsewhere on your system. If the files are not in this folder, use Windows Explorer to search for them. If they are not loaded on your machine, download and reinstall the tutorial from www.echoview.com or from the Echoview USB drive.

If you receive a message saying that the version of Echoview you are running cannot read the file you have opened, you may be running an old version of Echoview. You can download the latest version of Echoview from www.echoview.com.

Part I
Preamble

Introduction to the Code operator

Echoview variables[1] represent time–series measurements from a data file. Echoview utilizes these variables to store, manipulate and display that data—which are typically ping samples. This implementation ensures that raw data from the data file remains unmodified.

The Code operator is an Echoview virtual variable (refer to the About virtual variables online Help page) that is a Python application programming interface (API). It is specifically devised to transform echograms according to a filter or algorithm of your design. The operator accepts other Echoview variables as input, performs computations on the associated ping samples in Python, and returns the output to Echoview. The ping samples are the S$_v$, TS, power or other values of the data.

Python installation information

Echoview is installed with Python3, NumPy and SciPy for the Code operator. In Echoview, refer to the Python section of the General page of the Echoview Configuration dialog box for the version information.

Downloading and extracting the tutorial package

If you haven't already, download the tutorial package from

https://update.echoview.com/Tutorials/IntroToCodeOperator.exe

and double-click on the file to extract its contents. You may need to right-click and choose **Run as administrator** from the drop-down menu for the extractor to write to the default location at

```
C:\Echoview Software\Tutorials\IntroToCodeOperator
```

Code operator resources

In addition to this tutorial, there are other resources available for learning about the Code operator

i. The Using the Code operator online Help page
ii. The Introduction to the Code operator: A Python interface for Echoview video

You may also direct any questions you have to support@echoview.com.

[1]Not to be confused with the computer science definition where variable refers to a storage location symbolized by a name.

Objective of this tutorial

The objective of this tutorial is to equip you with the essential knowledge needed to write your own Code operator programs or modify existing ones. We will achieve this by

i. dissecting the commands in the default Code operator Echoview Python source file,
ii. developing fundamental Code operator skills through practices, and
iii. providing Code operator exercises for you to test your learning. There are four exercises with increasing difficulty:
    - Exercise 1a—Mean operator & Exercise 1b—Mean operator replicate Echoview's XxY statistic and Minus operators.
    - Exercise 2—Gaussian blur operator demonstrates how to implement a custom filter.
    - Exercise 3—Multibeam circle filter illustrates creating no data regions for multi-beam data. Note the significance, as regions are not supported for multibeam data in Echoview.

Outside of the basic objective of this tutorial, note that to fully exploit the power of the Code operator requires some proficiency in

- programming in Python,
- theory behind object-oriented programing (OOP), and
- OOP within the Python framework.

It is beyond the scope here to provide instruction on these topics. However, we provide some Python and OOP-related information for the Code operator in appendices B and C, that you may find useful for your learning as you work through the content.

Python programming syntax

In the interests of adhering to our objective, we will neither be reviewing how to program nor providing instruction on using Python[2]. Ideally, you will have some programming experience, including basic knowledge of Python such as

- importing libraries (using different syntaxes)
- knowledge of Python's built-in functions (https://docs.python.org/3/library/functions.html)
- assigning variables to names
- executing mathematical operations
- indenting blocks of code
- dealing with Boolean types
- constructing conditional statements
- defining functions
- manipulating lists and NumPy arrays

[2]Resources such as https://www.python.org/about/gettingstarted/ serve this purpose.

Part II
The Code operator

1 Getting started

While working through this tutorial remember that you can visit the online Echoview Help file to clarify any details along the way.

https://support.echoview.com/WebHelp/_Introduction/About_Echoview.htm

1.1 Launching Echoview and loading the data

1. To begin, open Echoview, create a new EV file, and with the Fileset window add[3]

```
C:\Echoview Software\Tutorials\IntroToCodeOperator\Simulated-data\example.sv.csv
```

   Echoview creates the raw variable

   Fileset 1: Sv comma-separated values

   Find this raw variable on the Dataflow window and rename it to

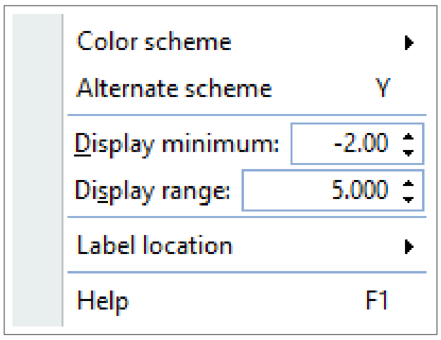
   Fileset 1: Sv CSV $\Longleftrightarrow$ Fileset 1: Sv CSV

   by right-clicking on it to open the shortcut menu and selecting **Rename**. (Or left-click to select it and press F2.)

2. Take a glance at the data.

   a. In the Dataflow window double-click on Fileset 1: Sv CSV to open the echogram for this datafile.

   b. On the Color legend set Display minimum to -2 and Display range to 5 (Figure 1).



Display minimum: -2.00
Display range: 5.000

Figure 1

[3]Default installation path.

c. Adjust the zoom to display all the pings and ping samples in the echogram.

   You can see that this example data consists of 10 pings with 5 samples per ping (Figure 2).

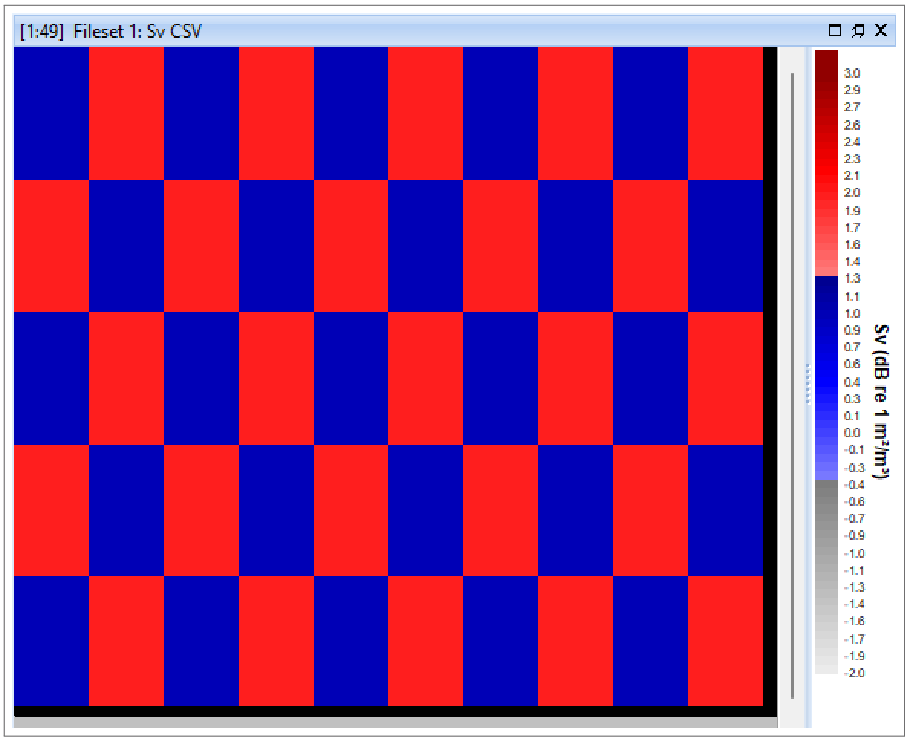


Figure 2: Echogram of the Fileset 1: Sv CSV variable with the Color legend Display minimum set to -2 and Display range set to 5. The echogram comprises pings 0 to 9 (left to right) with five ping samples per ping, alternating between 1 and 2 dB.

3. Press `Ctrl+S` and Echoview will open a Windows dialog box to save the new EV file.

   a. With the Windows dialog box first create a new folder called

   ```
   code-operator-getting-started
   ```

   click into it and enter the file name

   ```
   getting-started
   ```

   b. When you click **Save** Echoview creates `getting-started.EV`. Remember to keep saving your progress.

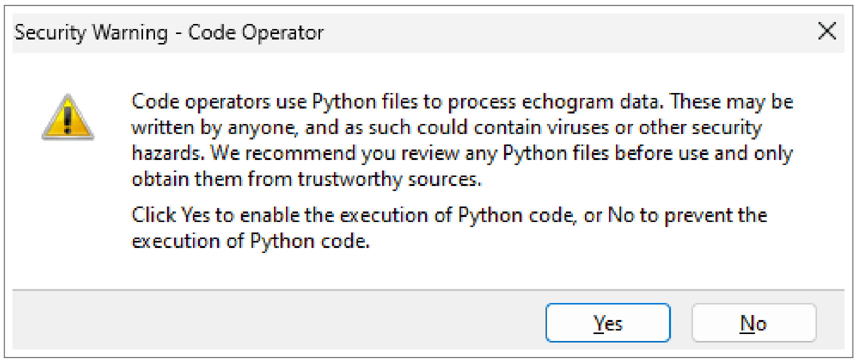
1.2 Creating a Code operator virtual variable in Echoview

4. In the Dataflow window click on

to preset it as Operand 1 for the next step.

5. Using the Dataflow toolbox or the New virtual variable dialog box, create a Code operator virtual variable.

   a. You will receive a security warning (Figure 3) advising you against executing untrusted Python code.



Security Warning - Code Operator

Code operators use Python files to process echogram data. These may be written by anyone, and as such could contain viruses or other security hazards. We recommend you review any Python files before use and only obtain them from trustworthy sources.

Click Yes to enable the execution of Python code, or No to prevent the execution of Python code.

Figure 3

   b. Click Yes to authorize Echoview to execute Python code for the Code operator and close the Security warning dialog box.

   c. You will now see a Code operator on the Dataflow window. Rename the new Code operator variable to

Code: Getting started ⟺

   d. Your Dataflow window should resemble Figure 4 .

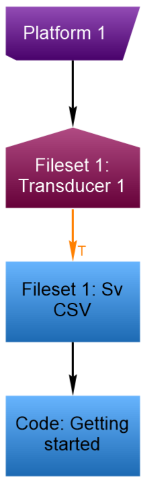


Figure 4

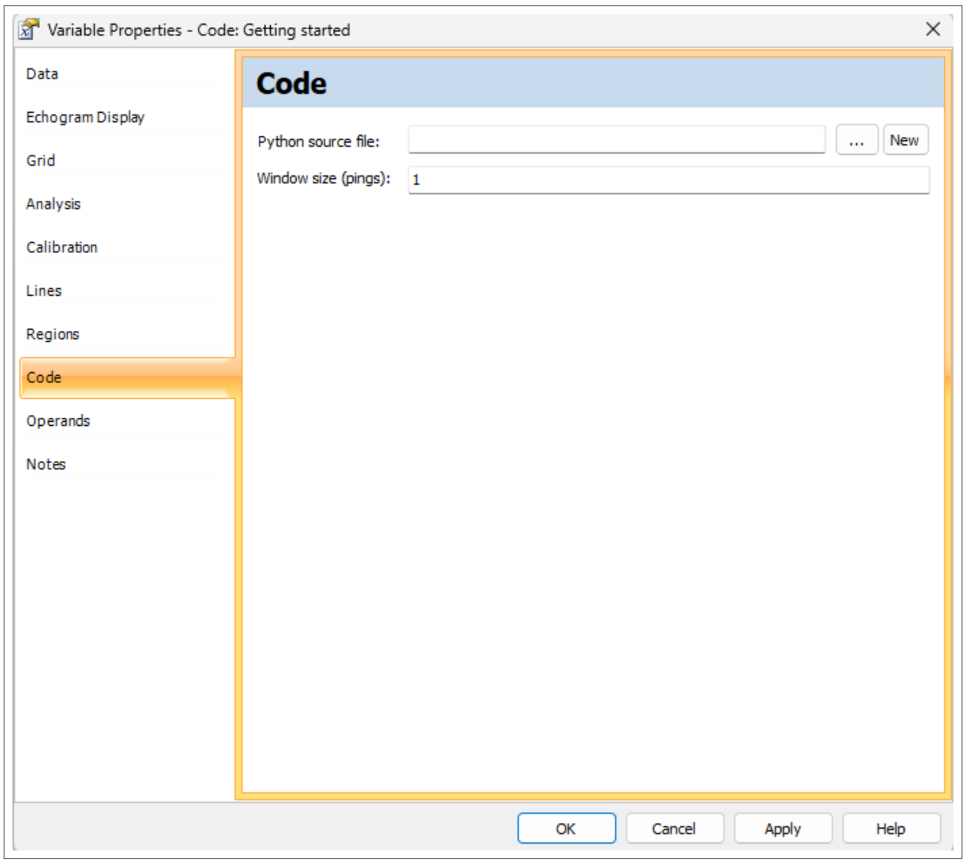
6. Save your changes to `getting-started.EV`.

2 Configuring settings for the Code operator

We will now introduce the settings on the Code page of the Variable properties dialog box for the Code operator.

1. Right-click on

and select **Variable properties** from the drop-down menu (or left-click and press F8) to open the Variable properties dialog box[4].
2. Click onto the **Code** page (Figure 5) where you will see the settings
   a. Python source file
   b. Window size (pings)



Python source file:
Window size (pings): 1

Figure 5: The Code page of the Variable Properties dialog box.

[4]The Variable Properties dialog box contains many settings to configure Echoview variables.

2.1 The Python source file setting

Use this to generate a new default Echoview Python source file (which we will now do) or specify the path to an existing one.

2.1.1 Generating a default Echoview Python source file

3. On the Code page (Figure 5) click on New. Echoview will open a Windows dialog box for you to save your new Echoview Python source file.
   a. In the Windows dialog box, navigate into

      `code-operator-getting-started` (from step 3 in section 1.1)

      and enter the file name

      `getting-started`

   b. When you click **Save** Echoview will generate a new Echoview Python source file, append the required `.py` extension and open `getting-started.py` in Windows Notepad (by default)[5].
   c. You will now see the path to `gettings-started.py` in the **Python source file** text box.

In the next section, we will outline what the code in `getting-started.py` does.

(Note for your future reference that you can use the ... button (Figure 5) to open the Echoview Python source file specified in the text box.)

2.1.2 About the commands in the default Echoview Python source file

Your new default Echoview Python source file contains commands to interface between Python and Echoview, comments to describe the code[6], and a simple program for the Code operator.

We will execute `getting-started.py` shortly in section 3. Meanwhile, here is an outline of what it does. Don't worry if you do not understand all the Python commands, we will clarify the code as we progress through the tutorial.

(If you are completely new to OOP, you may wish to first read appendix B, which introduces the technical terms coming up in this section, and that we will use during this tutorial.)

[5]To change this (and we recommend you do), specify the path to a text editor of your choice on the Echoview Configuration dialog box. Use the **Python source file editor** setting on the General page. There are plenty of freely available text editors that you can search for and download from the internet.

[6]You can refer to appendix A.1 for a condensed version of the default Echoview Python source file with the comments removed.

i. The code utilizes these `import` statements

```
from typing import List
import echoview as ev
import numpy as np
```

to access the `typing`, `echoview` and `numpy` Python packages. The `echoview` Python package (appendix C) contains the `OperatorBase`, `OperandInput`, `Measurement` and `MeasurementType` classes for the Code operator.

ii. Next, it defines the `Operator class`. The syntax

```
class Operator(ev.OperatorBase):
```

shows we derive `Operator` from the `OperatorBase class` of the `echoview` Python package.

You can think of `Operator` as the Python equivalent of the Code operator in Echoview.

iii. Now look at the `eval` method, which is where you program the function of the Code operator in Echoview.

```
def eval(self, inputs: List[ev.OperandInput]):
    first_input = inputs[0]
    return first_input.measurement.data
```

Here, the Python command copies Operand 1 (`inputs[0]`) and assigns it to the variable `first_input`.

Finally, the `return` statement communicates the ping sample data for `first_input` via `eval` to Echoview.

Note that the commands in points i. and ii. set up the interface between Python and Echoview. They do not vary (in essence) between different Echoview Python source files.

In contrast, the commands that constitute the `eval` method in point iii. characterize the function of your Code operator—i.e., it is in `eval` that you write your Code operator program.

Tabs vs spaces The default Echoview Python source file uses tabs for Python code indentation. You may choose to maintain using tabs or replace them with spaces instead. However, you cannot use a mix of both when indenting.

While attempting the practices and exercises in this tutorial be careful to maintain consistency in employing one or the other.

File encoding The default Echoview Python source file uses the UTF-8 character encoding system (refer to the Glossary entry on the online Help page). If you copy the text from the default Echoview Python source file to another file, you must save that file in the UTF-8 format as well.

Comments The Python interpreter ignores the `#` character and any text that follows it.

You can utilize this feature to add notes to your Echoview Python source or to comment out your code for testing.

```
# This is a comment that is ignored by the Python interpreter.
```

You can also encapsulate comments within triple quotes[7]

```
"""
Any text within the triple quotes are comments.
"""
```

2.2 The Window size (pings) setting

The Code operator iterates through an operand's echogram ping-by-ping (not necessarily in sequence[8]). If there are multiple operands, each iteration simultaneously reads the pings with identical ping indexes from all the input echograms.

The window is an aperture that parses the current ping (from each operand) and its adjacent pings into the operator.

4. Set aside or close `getting-started.py` and return to the Code page (Figure 5) where you can also see an input for **Window size (pings)**.

   For now, leave Window size (pings) as 1 to restrict the window to the current ping only. We will explore it in detail in section 9.

3 Executing the default Echoview Python source file

We will check one more setting on the Variable Properties dialog box before executing `getting-started.py` via Code: Getting started.

1. Click on the Operands page and verify that the Operands section shows

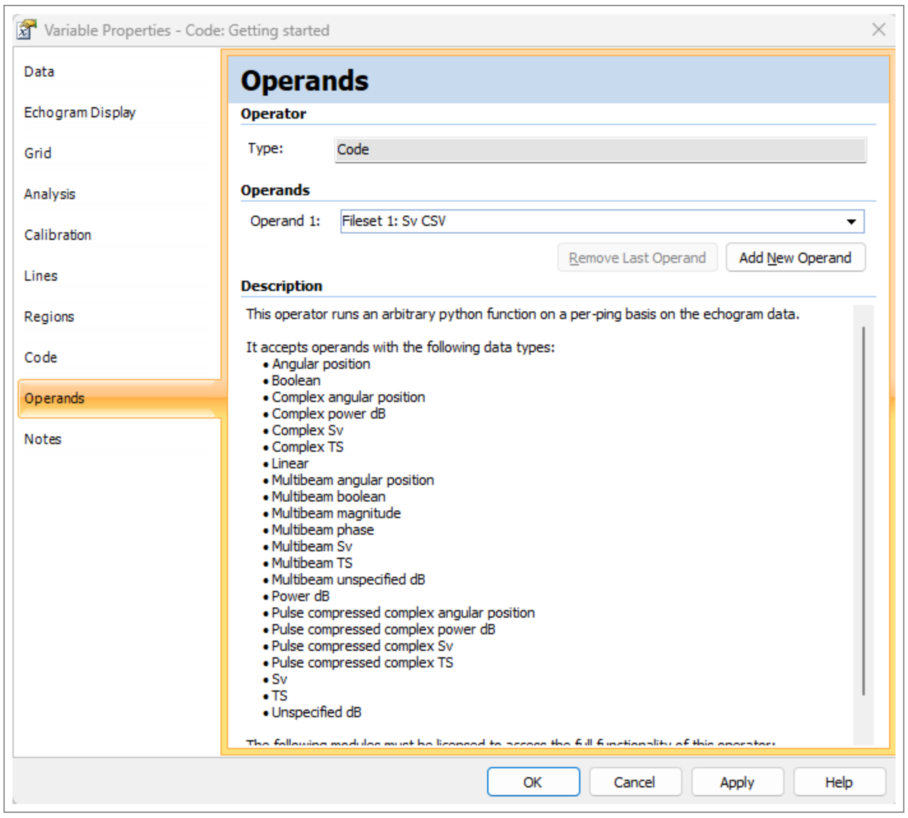
   ```
   Fileset 1:  Sv CSV
   ```

   Otherwise, click to open the drop-down menu for Operand 1 (Figure 6) and select this variable.

   `Fileset 1:  Sv CSV` now corresponds to `inputs[0]` in `getting-started.py`. (Refer to point iii. in section 2.1.2.)

[7]This is technically an unassigned string in the code.

[8]Mainly because Echoview processes pings on demand. For example, if the echogram view is only looking at the last few pings, then only those are processed.


Variable Properties - Code: Getting started

Data
Echogram Display
Grid
Analysis
Calibration
Lines
Regions
Code
Operands
Notes

Operands

Operator

Type: Code

Operands

Operand 1: Fileset 1: Sv CSV

Remove Last Operand | Add New Operand

Description

This operator runs an arbitrary python function on a per-ping basis on the echogram data.

It accepts operands with the following data types:
- Angular position
- Boolean
- Complex angular position
- Complex power dB
- Complex Sv
- Complex TS
- Linear
- Multibeam angular position
- Multibeam boolean
- Multibeam magnitude
- Multibeam phase
- Multibeam Sv
- Multibeam TS
- Multibeam unspecified dB
- Power dB
- Pulse compressed complex angular position
- Pulse compressed complex power dB
- Pulse compressed complex Sv
- Pulse compressed complex TS
- Sv
- TS
- Unspecified dB

The following modules must be licensed to access the full functionality of this operator:

OK | Cancel | Apply | Help

Figure 6: The Operands page of the Variable Properties dialog box.

2. Click OK to save your changes and close the Variable properties dialog box, and also save your changes to `getting-started.EV`.

3. In the Dataflow window double-click on Code: Getting started to execute the commands in `getting-started.py` and open an echogram of the result.

   a. On the Color legend for the echogram verify or set Display minimum to -2 and Display range to 5 (Figure 1). Adjust the zoom to display all the pings and ping samples in the echogram. You see what is shown in Figure 7.

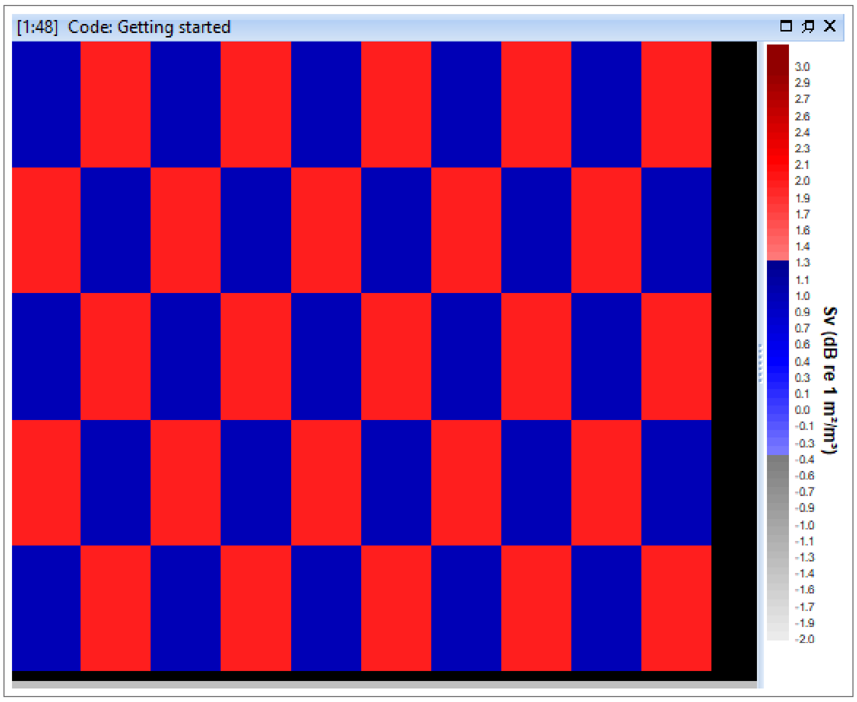


Figure 7: Echogram showing that the default Echoview Python source file returns a copy of the ping samples from Operand 1 to the Code operator. The echogram comprises pings 0 to 9 (left to right) with five ping samples per ping, alternating between 1 and 2 dB.

b. Compare it to the echogram for Fileset 1: Sv CSV in Figure 2. They are identical.

You can see that the default Echoview Python source file returns a copy of the ping samples from Operand 1.

3.1 Afterword: the Code operator as an Echoview variable

At this point, take stock of the similarity between the Code operator and Echoview's other virtual variables. In either case, you create an operator, and specify one or more operands. The operator performs a computation on the pings from the operands, and you visualize the output in the resulting echogram.

Whereas other virtual variables work to a specification, you have the freedom to program the function of the Code operator.

Also note that you can only execute Echoview Python source files via the Code operator (as you did in step 3 above). However, you may use your preferred text editor or integrated development environment (IDE) to write the Python code.

4 Practice—Adding 1 dB

We will now program Code: Getting started to add 1 dB to each sample of all the pings of Fileset 1: Sv CSV. If you do not understand the Python commands in the upcoming steps, we will examine them in section 4.1.1.

4.1 Reading, manipulating and returning pings with the `eval` method

1. Return to or open `getting-started.py` (section 2.1.1).
2. Look for the `eval` code block.

    ```
    def eval(self, inputs: List[ev.OperandInput]):
        ...
        ...
        return first_input.measurement.data
    ```

    This method reads in pings from all the input operands, executes your Python commands on the pings and communicates the resultant ping samples to Echoview.

    (Refer to appendix C.1.1 for more details on `eval` including the syntax for how it is defined.)
3. Find the `return` statement

    ```
    return first_input.measurement.data
    ```

4. Add 1 dB

    ```
    return first_input.measurement.data + 1
    ```

    and save the changes to `getting-started.py`.

While you have the echogram for Code: Getting started open, any changes that you save in the Echoview Python source will trigger the Code operator to reevaluate in Echoview. Look at its echogram now. You can see you have added 1 dB to each sample for all the pings (Figure 8).

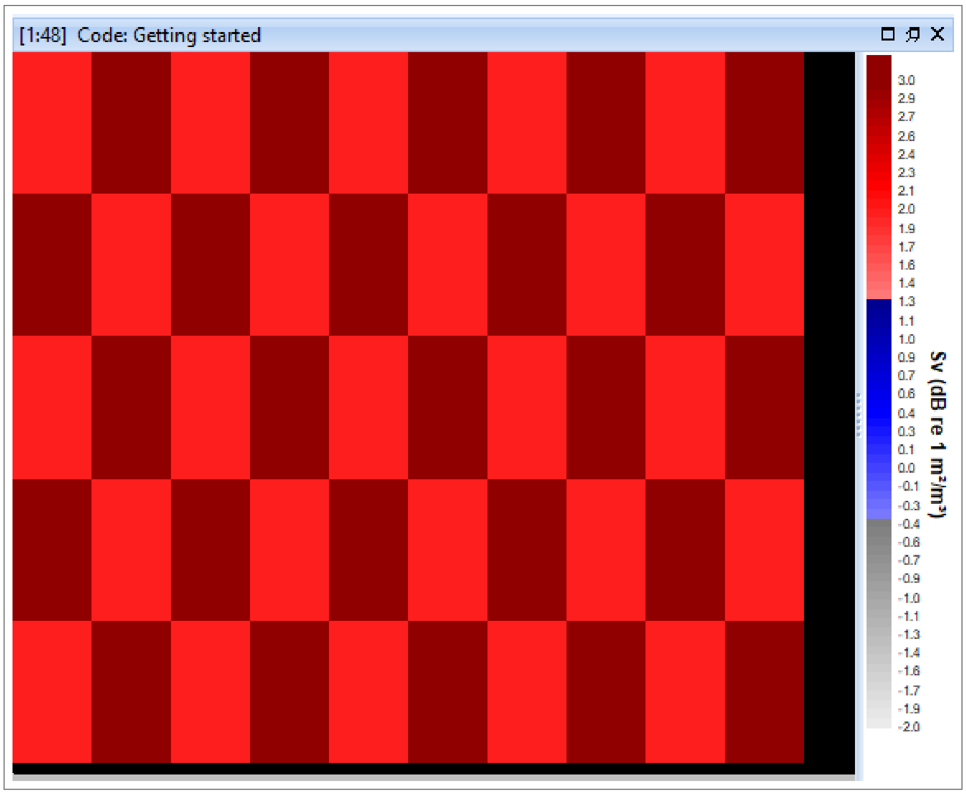


Figure 8: Echogram showing 1 dB added to each ping sample from the input echogram (shown in Figure 2).

4.1.1 Explaining the Python commands

In section 2.2 we specified Window size (pings) as 1. Therefore, the Code operator reads in one ping at a time from Operand 1—which is Fileset 1: Sv CSV.

When reading the first ping, 0 for example, we have

$$\left[\ 0\ \right]$$

where the square brackets represent the window, and the number in the middle the ping index.

Refer to the `return` statement

```
return first_input.measurement.data + 1
```

Recall from point iii. in section 2.1.2 that `first_input` corresponds to Operand 1. (We will explore this further in section 5.) Then

```
first_input.measurement
```

refers to the current ping[9] in the window, i.e.,

0

A ping is comprised of many attributes[10] such as the ping index, datetime, GPS distance and more[11]. However, the `return` statement for `eval` must communicate the ping samples to Echoview.

To access the ping samples use the `data` attribute

```
first_input.measurement.data
```

This is a NumPy array[12] with each element in the array corresponding to an individual ping sample. So for 0 the sample values are

```
first_input.measurement.data = [1, 2, 1, 2, 1]
```

To visualize `first_input.measurement.data`, you can compare the array with the value of each sample for 0 (leftmost ping) in Figure 7.

When we add 1 db with

```
first_input.measurement.data + 1
```

we are utilizing a shortcut feature (bypassing a `for` or `while` loop) available to NumPy arrays to add 1 to each ping sample to give

```
first_input.measurement.data + 1 = [2, 3, 2, 3, 2]
```

The `return` keyword communicates the above result to Echoview. Again, you can visualize `first_input.measurement.data + 1` by reading the value of each sample for 0 (leftmost ping) from the Color legend in Figure 8.

The Code operator repeats the computation for all the pings in Fileset 1: Sv CSV.

4.2 Returning the NumPy array of ping samples to Echoview

The array in the `return` statement needs to be a NumPy array of the same shape[13] as the input ping.

We will now force an error in Echoview by changing the shape of

[9] Technically, each item in the `inputs list` is an `OperandInput` object (section 10).
[10] The term *attributes* can have a technical meaning in programming. Refer to appendix B.2 for a description.
[11] You can find the complete list on the Using the Code operator online Help page.
[12] A popular data structure in Python.
[13] The length of the array in each dimension.

`first_input.measurement.data`. You may simply follow the steps for now, we will explain the Python commands in section 4.2.1.

5. In `getting-started.py` replace

   ```
   return first_input.measurement.data + 1
   ```

   with

   ```
   return first_input.measurement.data[1:4]
   ```

   and save the change.

In the Echoview messages dialog box, you receive the error

Code: Getting started: returned array must have the shape (5) but was (3).

and the echogram appears uniformly black.

Undo the change to `getting-started.py` and save to revert to the default Echoview Python source file. The echogram for Code: Getting started will reappear (Figure 7).

Note that you can double-click on error messages in the dialog box to expand them to show more details.

4.2.1 Explaining the Python commands

Python uses zero-indexing for all its array-like objects. Hence,

```
first_input.measurement.data[1:4]
```

uses NumPy's slicing feature to select the second up to the fourth[14] ping sample, so for ping **0** (leftmost ping) in Figure 7 for example, we have

```
first_input.measurement.data[1:4]= [2, 1, 2]
```

In trying to communicate `first_input.measurement.data[1:4]`, to Echoview, we caused an error. The messages dialog box reports that the input ping had five samples, however, the output ping only had three samples.

You also see that Echoview produces a uniformly black echogram if it encounters any errors in the Echoview Python source file.

4.3 Indentation and encoding errors

At the end of section 2.1.2, we explained that the default Echoview Python source file uses tabs for Python code indentation, and the UTF-8 character encoding system.

[14]Note that slicing `[start:stop]` in Python selects the ping sample corresponding to the start index up to but not including the ping sample at the stop index.

You will get an error if you mix tabs and spaces, or save the default file using an encoding system that does not support the special character within.

We recommend you gain some experience in recognizing both these errors.

4.3.1 Mixing tabs and spaces

Firstly, try replacing the tabs for one of the commands in `getting-started.py` with an equivalent number of spaces to maintain the indentation. Notice the error in the messages dialog box when you save the change.

Double-click on the error in the dialog box to open it. The exact message may vary, however, it will hint at a problem with the indentation, and indicate the line and column number with the problem. Undo the change to revert the error.

4.3.2 Invalid encoding

Next, use your text editor to change the character encoding system to one that does not support the ® special character in `getting-started.py`. Note that in this instance, the error message may not to indicate the actual line and column number containing the problem, but where the Python interpreter failed. Return the file character encoding system to UTF-8 to revert the error.

5 Practice—Loading additional operands into the Code operator

You are not constrained by the number of operand inputs into the Code operator. However, all the inputs must have the same number of pings and ping samples. (Refer to the Using multiple operands online Help page.)

We will now load a second operand into Code: Getting started. Then using `getting-started.py`, we will `return` the ping samples from Operand 2.

5.1 Loading the data

1. In Echoview go to the Fileset window. Create Fileset 2 and add[15]

```
C:\Echoview Software\Tutorials\IntroToCodeOperator\Simulated-data\example.ts.csv
```

Echoview creates the raw variable

Fileset 2: TS comma-separated values

[15] Default installation path.

Find this raw variable on the Dataflow window and rename it to

Fileset 2: TS CSV $\Longleftrightarrow$ Fileset 2: TS CSV

a. Double-click on Fileset 2: TS CSV to open its echogram.
b. On the Color legend set Display minimum to -2 and Display range to 5 (Figure 1).
c. Adjust the zoom until you display all the pings and ping samples in the echogram.

   You can see in the echogram that this example data consists of 10 pings, with 5 samples per ping (Figure 9).

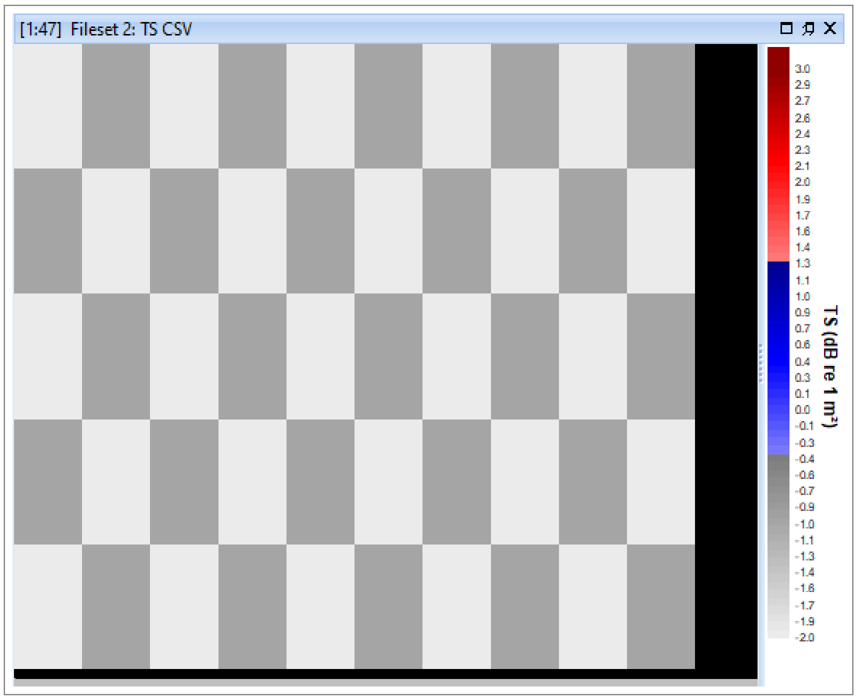


Figure 9: Echogram of Fileset 2: TS CSV with the Color legend Display minimum set to -2 and Display range set to 5. The echogram is comprised of pings 0 to 9 (left to right) with five ping samples per ping, alternating between -1 and -2 dB.

2. Save your changes to `getting-started.EV`.

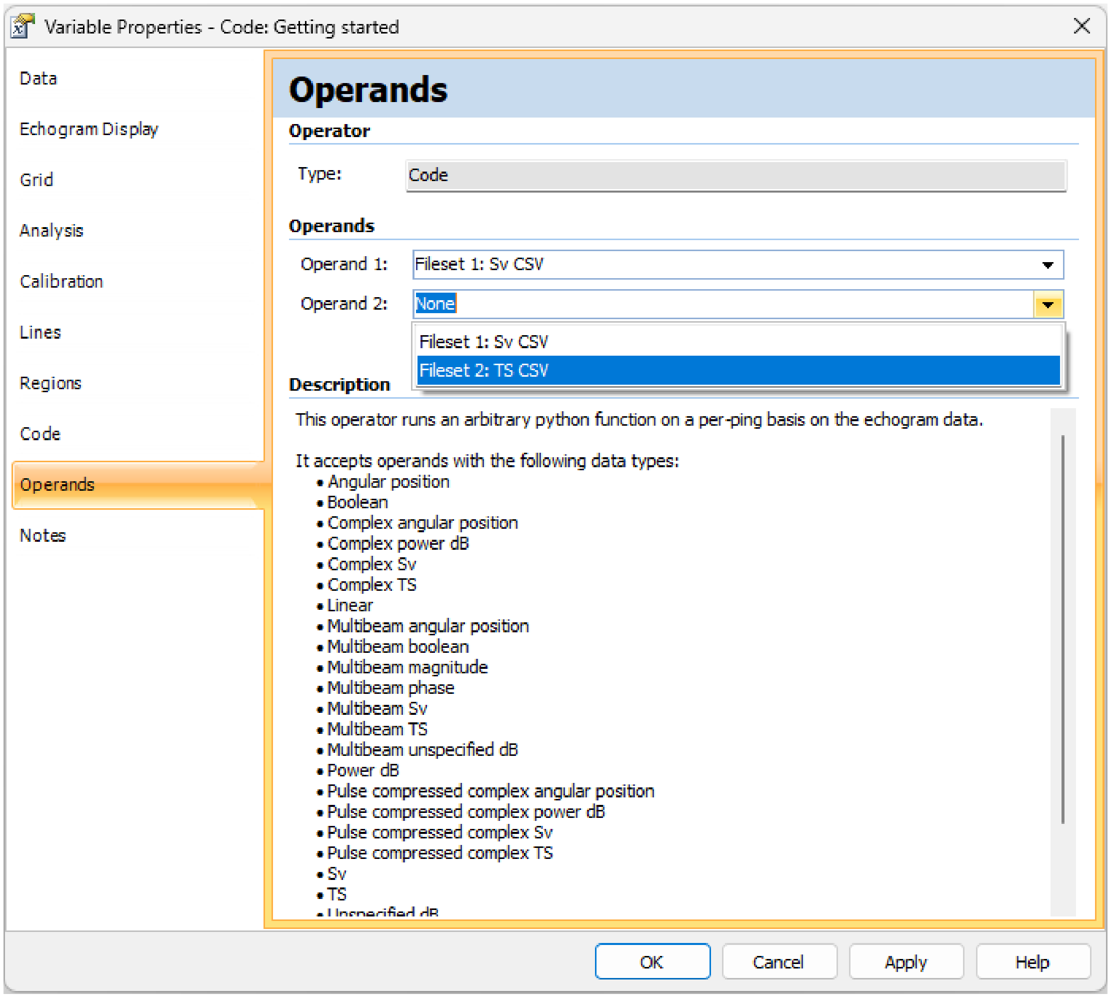
5.2 Adding operands in Echoview

3. Now open the Variable Properties dialog box for Code: Getting started and navigate to the Operands page.

4. In the Operands section click on the Add New Operand button. You will see a drop-down menu for Operand 2 appear (Figure 10).

5. From the drop-down menu for Operand 2 choose

```
Fileset 2:  TS CSV
```



Operands

Operator

Type: Code

Operands

Operand 1: Fileset 1: Sv CSV

Operand 2: None

Fileset 1: Sv CSV

Fileset 2: TS CSV

Description

This operator runs an arbitrary python function on a per-ping basis on the echogram data.

It accepts operands with the following data types:
- Angular position
- Boolean
- Complex angular position
- Complex power dB
- Complex Sv
- Complex TS
- Linear
- Multibeam angular position
- Multibeam boolean
- Multibeam magnitude
- Multibeam phase
- Multibeam Sv
- Multibeam TS
- Multibeam unspecified dB
- Power dB
- Pulse compressed complex angular position
- Pulse compressed complex power dB
- Pulse compressed complex Sv
- Pulse compressed complex TS
- Sv
- TS
- Unspecified dB

Figure 10: Adding an operand to the Code operator.

6. Check on the Code page (Figure 5) that the Python source file setting is still showing the path to `getting-started.py`. (If not add it or create it by referring to section 2.1.1.)

7. Click OK to save your edits and close the Variable Properties dialog box, and also save your changes to `getting-started.EV`.

Your Dataflow window should now resemble Figure 11.

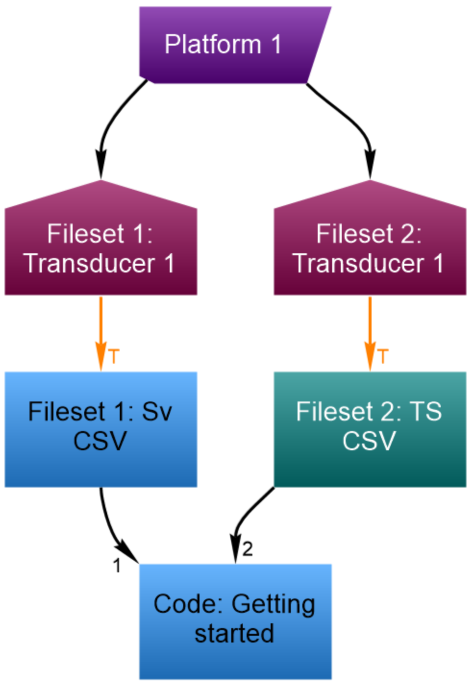


Figure 11: The Dataflow window showing Code: Getting started with two input operands.

5.3 Adding operands in the Echoview Python source file

Open `getting-started.py` in a text editor and look for

```
def eval(self, inputs: List[ev.OperandInput]):
```

The `inputs` parameter in the `eval` method is a Python `list`. Each item in the `inputs` `list` corresponds to an operand on the Operands page (Figure 10).

So after step 5 from section 5.2, you can conceptualize the `inputs` `list` as

`inputs` = [Fileset 1: Sv CSV , Fileset 2: TS CSV]

Given that Python uses zero-indexing, `inputs[0]` is Fileset 1: Sv CSV and `inputs[1]` is Fileset 2: TS CSV. If you add more operands, they are appended to the `inputs` `list`, such that `inputs`$[N-1]$ corresponds to Operand N.

5.4 Returning the ping samples from another operand

8. In `getting-started.py` look for

   ```
   first_input = inputs[0]
   ```

   and immediately below it insert

   ```
   second_input = inputs[1]
   ```

   Ensure you maintain the indentation level, and that you use tabs or spaces—in a consistent way—to indent (section 2.1.2). The code snippet should now look like

   ```
       ...
       ...
   # Access the input ping of operand 0
   first_input = inputs[0]
   second_input = inputs[1]
   # Calibration settings can be obtained from a ping using:
       ...
       ...
   ```

   You can refer to appendix A.2 for the completed Python code showing the change.

   We can now access the input pings from Fileset 2: TS CSV with `second_input`.

9. To communicate the ping samples for `second_input` to Echoview, change the `return` statement for `eval` from

   ```
   return first_input.measurement.data
   ```

   to

   ```
   return second_input.measurement.data
   ```

   and save `getting-started.py`.

10. In Echoview open the echogram for Code: Getting started and compare it to the echogram for Fileset 2: TS CSV (Figure 9). They are identical.

11. Save your changes to `getting-started.EV`.

6 Practice—Changing the data type of the Code operator

At the end of the previous section, we `return` TS ping sample values to Echoview. However, if you hover your cursor over Code: Getting started, the Details dialog box reports that its data type is Sv.

The Code operator supports variables of different data types, but it cannot automatically distinguish between the data types from multiple input operands. So it automatically defaults to the data type of Operand 1.

There are two ways we can rectify this.

i. Reorder the operands on the Code page (Figure 5) of the Variable Properties dialog box so that Fileset 2: TS CSV is Operand 1.

ii. Use `getting-started.py` to change the data type with the `result_type` method, which we will now do. Afterwards, in section 6.1.1, we will explain the Python commands in detail.

6.1 Changing the data type with the `result_type` method

1. Return to or open `getting-started.py` (section 2.1.1) and find

   ```
   def eval(self, inputs: List[ev.OperandInput]):
   ```

   and immediately above it insert the following code block

   ```
   def result_type(self, input_types: List[ev.MeasurementType]):
       return input_types[1]
   ```

   (Refer to appendix C.1.2 for more details on `result_type` including the syntax of the Python code.)

   Ensure you maintain the indentation level, and that you use tabs or spaces—in a consistent way—to indent. The code snippet should now look like

   ```
       ...
       ...
   def result_type(self, input_types: List[ev.MeasurementType]):
       return input_types[1]

   def eval(self, inputs: List[ev.OperandInput]):
       """Returns a numpy array representing the output ping's samples.
       ...
       ...
   ```

2. When you save your code changes in `getting-started.py`, the Code operator converts from Sv to TS in Echoview if the echogram for the operator is open.

   Code: Getting started $\Longrightarrow$ Code: Getting started

   You may need to close and reopen the Dataflow window to trigger the update.

3. Save your changes to `getting-started.EV`.

Important: Note that `result_type` only changes the data type it does not perform the computations to convert the ping samples.

6.1.1 Explaining the Python commands

The parameter `input_types` in this method is a Python `list` corresponding to the data types of all the operands from the Operands page of the Variable Properties dialog box[16] (Figure 10), i.e.,

`input_types` = [Fileset 1: Sv CSV data type , Fileset 2: TS CSV data type]

Given that Python uses zero-indexing, with the command

```
return input_types[1]
```

we communicate

Fileset 2: TS CSV data type

to Echoview.

Analogously to the `eval` method, if you add more operands, they are appended to the `inputs_types` `list`, such that `inputs_types`$[N-1]$ corresponds to the data type of Operand N.

7 Practice—Working with ping samples from different operands

We will now perform an example computation with the ping samples of Fileset 1: Sv CSV and Fileset 2: TS CSV.

[16]Each item of this `list` is an object of the `MeasurementType` `class`—refer to appendix C.4.

For demonstration purposes we shall assume conditions allowing us to subtract TS from S_v to estimate the number of targets per unit volume, ρ_v. Since S_v and TS are in units of dB, we then convert the result to the linear domain[17]. Expressed mathematically

$$\rho_v = 10^{\left(\frac{\mathsf{S}_v - \mathsf{TS}}{10}\right)}$$

If you encounter any problems in the following steps, refer to appendix A.3 for the completed Python code for this computation. We will also explain the Python commands in detail in section 7.1.1.

7.1 Computing the number of targets per unit volume

Before proceeding, recall that `first_input.measurement.data` are the ping samples from Fileset 1: Sv CSV and `second_input.measurement.data` are the ping samples from Fileset 2: TS CSV (section 5.4).

1. Return to or open `getting-started.py` (section 2.1.1) and look for the line

   ```
   second_input = inputs[1]
   ```

   Immediately below this command insert

```
targets_per_unit_volume_log = first_input.measurement.data - second_input.measurement.data
```

2. Since the ping samples are in units of dB, we need to convert the samples to the linear domain. So on a new line immediately below the previous addition, insert

   ```
   targets_per_unit_volume_linear = 10**(targets_per_unit_volume_log/10)
   ```

3. Finally, change

   ```
   return second_input.measurement.data
   ```

   to

   ```
   return targets_per_unit_volume_linear
   ```

   and save the modifications.

[17] Recall that subtraction in the logarithmic domain is equivalent to division in the linear domain. That is, if $X = \log_{10} x$ and $Y = \log_{10} y$, then $X - Y = \log_{10} \frac{x}{y}$. And $10^{\frac{X-Y}{10}}$ converts logarithmic to linear.

4. In Echoview open the echogram for Code: Getting started (Figure 12) to see the result.

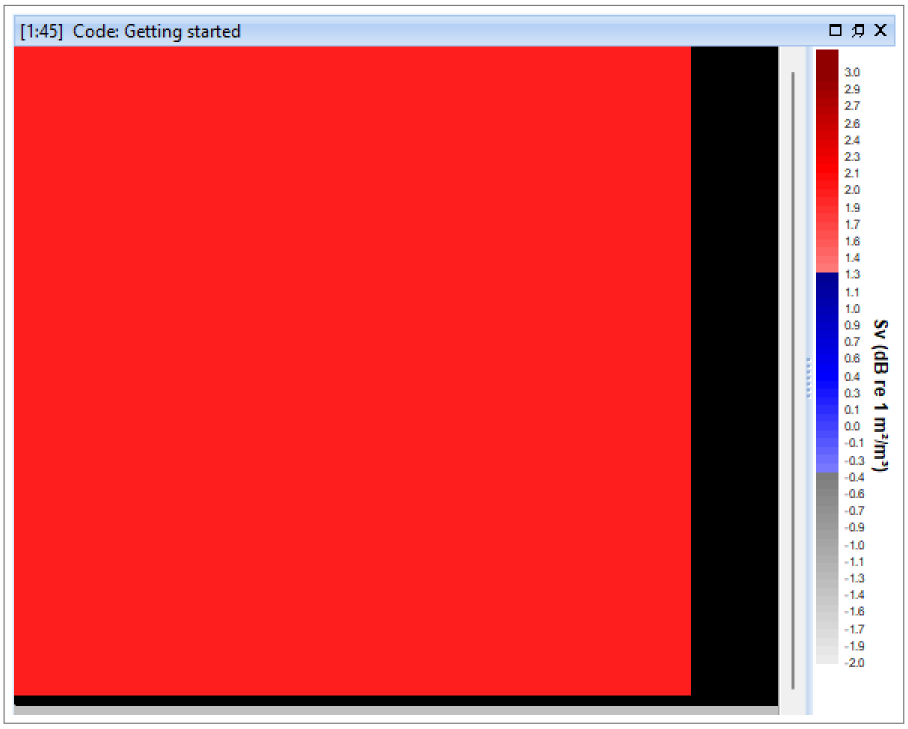


Figure 12: Echogram showing the result from subtracting the ping samples from Fileset 1: Sv CSV and Fileset 2: TS CSV, and converting the result to linear form.

5. Save your changes to `getting-started.EV`.

7.1.1 Explaining the Python commands

Recall that the ping samples for

`first_input.measurement.data` and `second_input.measurement.data`

are in NumPy arrays (e.g., section 4.1.1), and that the Code operator reads the equivalent ping from both operands as it iterates through the pings (section 2.2).

We can then perform element-wise subtraction for the corresponding ping samples with the command

```
first_input.measurement.data - second_input.measurement.data
```

So taking the ping samples for ping 0 (leftmost pings in Figures 2 and 9) as an example,

```
first_input.measurement.data - second_input.measurement.data
 = [1, 2, 1, 2, 1] - [-2, -1, -2, -1, -2]
 = [3, 3, 3, 3, 3]
```

We assign this result to `targets_per_unit_volume_log`.

To convert from logarithmic to linear form, the next command performs the element-wise operation of dividing the ping samples by 10, and raising result to the power[18] of `10`.

`10**(targets_per_unit_volume_log/10)`

$$= \left[10^{\frac{3}{10}}, 10^{\frac{3}{10}}, 10^{\frac{3}{10}}, 10^{\frac{3}{10}}, 10^{\frac{3}{10}}\right]$$

`= [1.995..., 1.995..., 1.995..., 1.995..., 1.995...]`

We assign the result to `targets_per_unit_volume_linear`, which `return` communicates to Echoview.

The Code operator repeats the computation for all the pings in Fileset 1: Sv CSV and Fileset 2: TS CSV, resulting in identical ping sample values for all the pings in Figure 12.

7.2 Choosing a data type with the `result_type` method

Now return to `getting-started.py`. We need to fix one more thing.

Echoview will maintain the data type for Code: Getting started from section 6, which is TS. However, this is incorrect for ρ_v, which is a linear quantity.

6. To rectify this, modify the `return` statement from

   ```
   def result_type(self, input_types: List[ev.MeasurementType]):
       return input_types[1]
   ```

   to

   ```
   def result_type(self, input_types: List[ev.MeasurementType]):
       return ev.MeasurementType.SINGLE_BEAM_LINEAR
   ```

7. When you save your code changes in `getting-started.py`, the Code operator converts from TS to Linear in Echoview if the echogram for the operator is open

   Code: Getting started $\Longrightarrow$ Code: Getting started

   You may need to close and reopen the Dataflow window to see the change.

[18] The Python syntax for raising x to the power of y is `x**y`.

7.2.1 Explaining the Python commands

We did not conjure `ev.MeasurementType.SINGLE_BEAM_LINEAR` out of nowhere.

`MeasurementType` is a `class` that belongs to the `echoview` package (appendix C), which we imported with

```
import echoview as ev
```

in `getting-started.py` (look for it near the top). The syntax

```
ev.MeasurementType
```

accesses the `MeasurementType` `class` from the `echoview` package which contains all the different data types that the Code operator supports.

In specifying

```
ev.MeasurementType.SINGLE_BEAM_LINEAR
```

we chose the correct data type for our result.

You can find the complete list of supported data types on the Using the Code operator online Help page.

8 Review

We have now covered some of the basic tasks for working with the Code operator. Before proceeding, here is a review of the material so far.

- We created a Code operator variable in Echoview (section 1.2).
- We introduced the settings on the Code page of the Variable Properties dialog box for the Code operator (section 2):
  - Python source file
  - Window size (pings)
- We generated a default Echoview Python source file, which contains commands to interface between the Code operator and Echoview (section 2.1.1).
- We gained some experience in
  - executing the Echoview Python source file (section 3),
  - modifying the ping samples of the input echogram (section 4),
  - adding a second operand to the Code operator (section 5.2),
  - working with ping samples from multiple operands (section 7), and
  - changing the data type of the Code operator (sections 6 and 7.2).

You can now close `getting-started.EV` and `getting-started.py`. We will not need them for the next sections. Certainly revisit the content if you need as you work through the rest of this tutorial.

9 Introducing the window of measurements

Recall that the Code operator iterates through an echogram ping-by-ping, and that there is a window that parses the current ping and any adjacent pings into the operator (section 2.2).

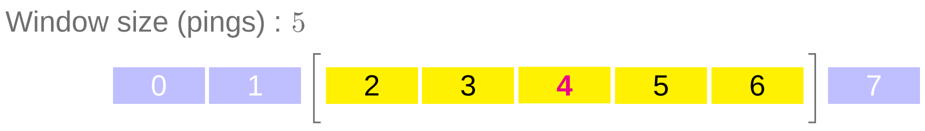
You use the Window size (pings) setting on the Code page (Figure 5) to specify the extent of the window, and hence, the number of pings that the Code operator reads adjacent to the current ping.

For technical reasons, we refer to the window as the *window of measurements*[19]. However, you can also interpret it as the *window of pings*.

9.1 Understanding the window of measurements in concept

The snapshot below illustrates the window of measurements—symbolized by the square brackets—for the example when Window size (pings) is 5.

This diagram is for a single operand. If you specify multiple operands for the Code operator, each operand has its own window.



The row of boxes represents all the pings from an operand. However, only five pings (in yellow)—based on our specified window—are currently processed in the Code operator via the window of measurements.

As the Code operator iterates through the echogram, the current ping—illustrated above as **4** —is called the *matched* ping.

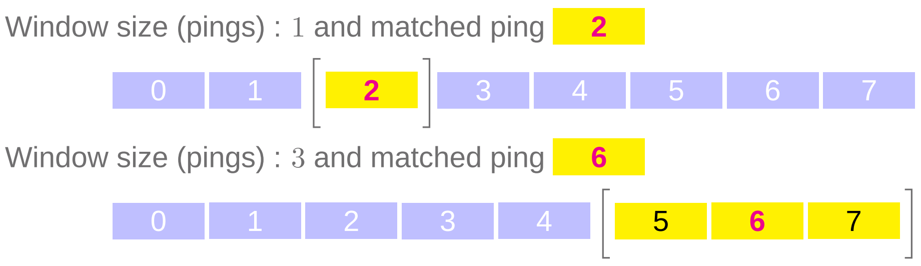
You can imagine that with each iteration, the window of measurements encompasses a new matched ping and the corresponding group of adjacent pings into the Code operator.

Since the window of measurements is composed of the matched ping in the middle and an equal number pings on either side[20], the Window size (pings) setting is always an odd

[19] In section 10 we will learn what a *measurement* is in the context of the Code operator.

[20] In section 11 we will explore the edge cases when the size of the window is less than Window size (pings).

number, e.g.,


Window size (pings) : 1 and matched ping 2
0
1
2
3
4
5
6
7
Window size (pings) : 3 and matched ping 6
0
1
2
3
4
5
6
7

Note that you can expect the Code operator's speed to decrease if there are many pings in the window. Hence, it is generally best to keep Window size (pings) to the minimum you need.

Also, keep for future reference that you obtain the value in Window size (pings) with the attribute

```
self.window_size
```

in the Echoview Python source file. We will utilize it in section 11.1.

9.2 Utilizing the window of measurements

Referring to the default Echoview Python source file (appendix A.1) as an example, Operand 1 is `inputs[0]` (point iii. in section 2.1.2) and is assigned to `first_input`, i.e.,

```
first_input = inputs[0]
```

Given the above, there are three attributes for `first_input`.

i. `first_input.measurement_window`

   This is the window of measurements.

   It is a Python `list`, meaning you can iterate over these measurements, select individual ones or generally interact with the window as you would a `list`. For example,

   ```
   first_input.measurement_window[0]
   ```

   corresponds to the first measurement.

   Depending on your choice of Window size (pings)—using the illustrations from section 9.1—`first_input.measurement_window` will be

Window size (pings) : 5

$$\begin{bmatrix} 2 & 3 & \mathbf{4} & 5 & 6 \end{bmatrix}$$

Window size (pings) : 1

$$\begin{bmatrix} \mathbf{2} \end{bmatrix}$$

Window size (pings) : 3

$$\begin{bmatrix} 5 & \mathbf{6} & 7 \end{bmatrix}$$

ii. `first_input.measurement_index`

This is the index position of the matched ping (section 9.1) in the window of measurements.

For each of the examples in point i., the `first_input.measurement_index` attribute evaluates to a Python `int` (integer), corresponding to the index position of pings **4**, **2** and **6** in the window—i.e., 2, 0 and 1.

iii. `first_input.measurement`

This is the matched ping.

For each of the examples in i., `first_input.measurement` evaluates to pings **4**, **2** and **6** respectively.

As a small exercise, you may wish to refer to the diagrams in section 9.1 and try to figure out what

- `first_input.measurement_window`,
- `first_input.measurement_index`, and
- and `first_input.measurement`

evaluate to for different matched ping and Window size (pings) combinations.

9.3 Summarizing notes on the window of measurements

If we now analyze the `return` statement in the default Echoview Python source file (appendix A.1)

`first_input.measurement`

is the matched ping, and

`first_input.measurement.data`

is the ping samples of the matched ping.

Look at the echogram in Figure 2. If the matched ping were 6 then

```
first_input.measurement = 6
```

and

```
first_input.measurement.data = [1, 2, 1, 2, 1]
```

10 The `Measurement class`

All the pings in the window of measurements such as

```
first_input.measurement_window[0]
```

or

```
first_input.measurement
```

are objects of the `Measurement class` (appendix C.3).

Aside from the sample data

```
first_input.measurement.data
```

with the attributes of the `Measurement class`, you can access other aspects of the ping, such as the ping index

```
first_input.measurement.index
```

Refer to the complete list of the `Measurement class` attributes on the Using the Code operator online Help page.

11 Practice—Outputting messages to a log

You can output messages to a log file to facilitate checks on your Code operator programs. Log messages are helpful to debug your program if it is not working, or not working as expected. In this practice, we will program a Code operator to create a log and output messages to it.

We will utilize our Code operator to determine how the window of measurements (section 9.1) behaves at the edges of an echogram, where there may not be an equal number of pings flanking the matched ping.

If you do not understand the Python commands in the upcoming steps, we will examine them in section 11.1.1. Then, in section 11.2 we will interpret the log.

You can find the completed Echoview Python source file commands for this practice in appendix A.4.

11.1 Logging the number of pings in the window of measurements

1. To begin, open Echoview, create a new EV file, and with the Fileset window add

```
C:\Echoview Software\Tutorials\IntroToCodeOperator\Simulated-data\example.sv.csv
```

   Echoview creates the raw variable

   Fileset 1: Sv comma-separated values

   Find this raw variable on the Dataflow window and rename it to

   Fileset 1: Sv CSV $\Longleftrightarrow$ Fileset 1: Sv CSV

2. Take a glance at the data.

   Double-click on the raw variable to open its echogram, set the Color legend Display minimum to -2 and Display range to 5 (Figure 1), and adjust the zoom until you display all the data in the echogram (Figure 2).

3. Press `Ctrl+S` and Echoview will open a Windows dialog box to save the new EV file.

   a. With the Windows dialog box first create a new folder called

      `code-operator-window-edges`

      click into it and enter the file name

      `window-edges`

   b. When you click **Save** Echoview appends the required `.EV` extension. Remember to keep saving your progress.

4. Create a Code operator virtual variable (section 1.2)

   a. Rename the new Code operator variable to

      Code: Window $\Longleftrightarrow$ Code: Window

   b. Navigate to the Code page (Figure 5) and click on New to generate a new Echoview Python source file.

   c. In the Windows dialog box that pops up, navigate into

      `code-operator-window-edges`

and save the Echoview Python source file as

```
window-edges
```

Echoview will automatically append the required `.py` extension to the file name and open `window-edges.py` in Windows Notepad (by default). Set this aside for now.

d. Still on the Code page
   i. confirm that you see the path to `window-edges.py` in the **Python source file** text box (Figure 5), and
   ii. enter 5 for **Window size (pings)**

e. Finally, click on the Operands page (Figure 6) and verify that Operand 1 for the **Operands** section shows

```
Fileset 1: Sv CSV
```

5. Click OK to save your changes and close the Variable properties dialog box. Also save your changes to `window-edges.EV`.

   Your Dataflow window should resemble Figure 13.

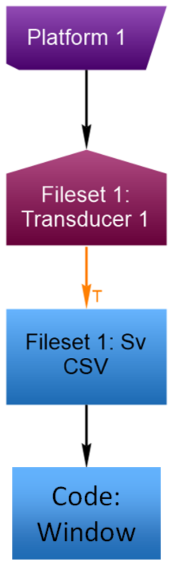


Figure 13

6. Return to `window-edges.py`, where we will now create a program to output debug messages from Code: Window to tell us what the window is doing.

   While editing `window-edges.py`, ensure you maintain the indentation level, and that you use tabs or spaces—in a consistent way—to indent (section 2.1.2).

a. Look for

```
def eval(self, inputs: List[ev.OperandInput]):
```

and immediately above it insert the following code block, whilst replacing the path in the example with a path to a location on your PC instead.

Make sure that the Code operator can write to the location (e.g., on your Desktop or Documents folder). Also take note of the double backslash in the syntax to separate directory names.

```
def __init__(self):
    self.logfile = "C:\\path_to\\code-operator-window-edges\\window-edges.log"
```

The code snippet should now look like

```
    ...
    ...
def __init__(self):
    self.logfile = "C:\\path_to\\code-operator-window-edges\\window-edges.log"

def eval(self, inputs: List[ev.OperandInput]):
    """Returns a numpy array representing the output ping’s samples.
    ...
    ...
```

b. Next, look for the `return` statement and immediately above it insert

```
current_window_length = len(first_input.window_measurements)
with open(self.logfile, 'a') as log:
    log.writelines(
        f"Matched ping: {first_input.measurement.index}\n "
        f"Current window length: {current_window_length}/{self.window_size}\n "
        f"Pings in window:\n ")
    for ping in first_input.window_measurements:
        log.writelines(f"\t Ping {ping.index}\n ")
    log.writelines(f"\n ")
```

7. Save your changes to `window-edges.py` (UTF-8 encoding) then in Echoview double-click on **Code: Window** to execute your program.

If you see a completely black echogram, or errors appearing in Echoview’s Message dialog box, review your commands in `window-edges.py` against the completed Echoview

Python source file in appendix A.4. Also ensure you have not mixed tabs and spaces whilst indenting, and that the file is in the UTF-8 character encoding system.

Otherwise, note that if you save any further changes to `window-edges.py`, Echoview will execute the commands again, causing additional messages to appear in the log.

11.1.1 Explaining the Python commands

Firstly, we created the attribute

```
self.logfile = "C:\\path_to\\code-operator-window-edges\\window-edges.log"
```

which is a Python `str` (string) specifying the path to our log file for Code: Window.

As Code: Window iterates through the pings in the echogram of Operand 1, we utilize Python's built-in `len` (length) function to find the number of pings in the window for a given matched ping. We assign the result to `current_window_length`.

```
current_window_length = len(first_input.window_measurements)
```

Next, the `with` statement

```
with open(self.logfile, 'a') as log:
```

opens `self.logfile` and assigns it to the variable name `log`. The `'a'` argument opens `self.logfile` in append mode (so that existing logs are not overwritten).

The object we call `log`, created by the `with` statement, has a `writelines()` method. Any string we provide as the first argument to `writelines()` (e.g., `log.writelines("Print this message.")`) will be written to `self.logfile`.

To create the log messages, we employ Python's print formatting feature. We compose our messages within the `f"..."` commands. For example

```
f"Current window length: {current_window_length}/{self.window_size}\n "
```

Print formatting replaces the variable within the `{}` brackets with its value.

So `current_window_length` evaluates to the number of pings in the window of measurements for the current matched ping.

And `self.window_size` evaluates to the value of Window size (pings). (Refer to appendix B.2.2 for more details on `self.window_size` including the syntax of the Python code.)

`\n` moves the cursor to a new line in the log file.

Now refer to the `for` loop

```
for ping in first_input.window_measurements:
```

iterates through the pings in the window of measurements (section 9.2), assigning each ping in `first_input.window_measurements` to the variable `ping`.

Within the `for` loop code block

```
log.writelines(f"\t Ping {ping.index}\n ")
```

evaluates the ping index of `ping`. The `\t` command inserts a tab in the log file, which helps with readability.

Finally, as the Code operator iterates through the pings of the echogram, we advance to the next line in the log file for each ping with

```
log.writelines(f"\n ")
```

11.2 Interpreting the log

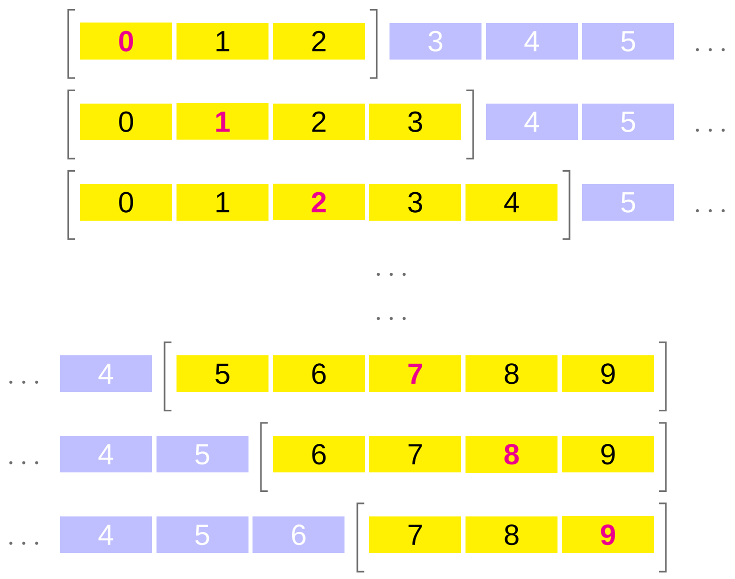
Open the `window-edges.log` file in the destination you specified, and scanning from the top you should see

```
Matched ping:  0
Current window length:  3/5
Pings in window:
     Ping 0
     Ping 1
     Ping 2

Matched ping:  1
Current window length:  4/5
Pings in window:
     Ping 0
     Ping 1
     Ping 2
     Ping 3
    ...
    ...
    ...
```

The log informs us that—for the matched pings at the start and end of the echogram—the window contains fewer pings than value you specify in Window size (pings).

The following schematic illustrates the results from the log. Each row of boxes represents the pings from Operand 1 in the vicinity of the matched ping (e.g., **2**). However, only the yellow pings are currently processed in the Code operator via the window. The window—symbolized by the square brackets—moves with each iteration to a new matched ping.



11.3 Afterword

You can certainly modify the commands in `window-edges.py` to output debugging messages for your own Code operator programs.

However, it is typically not feasible to output information on all the pings or ping samples as we did in section 11.1. Because for real echograms, all the pings and ping samples will generate too much data.

If you do incorporate such debugging statements, you need to output a subset of the data instead. (Refer to section 4.2.1 for an example on how you can use array slicing to do this.)

And remember the Code operator does not always iterate through the pings in an echogram sequentially. Meaning the log messages may not be in numerical ping index order.

You may also refer to the Diagnosis Echoview Python source file example on the online Help page, which provides formalized Python logging paradigms.

12 Review

In the last few sections, we introduced the concept of the window of measurements (section 9). This is an aperture that allows the Code operator to read adjacent pings to the matched ping. The Window size (pings) setting specifies the size of the window.

We learned that the pings in the window of measurements are objects of the `Measurement` `class` (section 10), meaning you can use the attributes of this `class` to access the data and metadata of the pings with the Code operator.

We then utilized a Code operator to create a log file and output messages to it (section 11). We used the log to verify that the matched ping is always at the center of the window of measurements, except at the edges. Here, there are an insufficient number of pings flanking the matched ping, and the window of measurements truncates. You can adopt this program to serve as a template to create log messages for your own Code operator programs.

You can now close `window-edges.EV` and `window-edges.py`. However, certainly revisit the content if you need. In the next section, we will take a closer look at the NumPy arrays of ping samples in the Code operator in preparation for the exercises.

13 Comments on ping samples in NumPy arrays

13.1 Outputting other ping attributes

In the last practice (section 11), we used a `for` statement to loop through all the pings in the window and obtained their ping index

```
for ping in first_input.window_measurements:
    log.writelines(f"\t Ping {ping.index}\n ")
```

What about their sample data? Since this is a very small dataset, you can replace `ping.index` with `ping.data` in `window-edges.py` and see the result in `window-edges.log`.

Try it now and compare the values in `window-edges.log` to the Fileset 1: Sv CSV echogram (Figure 2).

You may also wish to try outputting the other attributes of a ping to `window-edges.log`, to discover all the properties of a ping available to you via the Code operator. You can obtain the list of attributes from the Using the Code operator online Help page.

13.2 Ping sample data from all pings in the window of measurement

Instead of outputting the sample data for the pings in the window of measurements, here is a technique for collating them into a Python `list`, which you can then process.

(There is no need to implement this example in `window-edges.py`, you will use this technique in the upcoming exercises.)

Still considering Fileset 1: Sv CSV from the last practice (section 11), if Window size (pings) = 7, and the current ping is 4 then we have

`first_input.window_measurements` = [1 2 3 4 5 6 7]

We can accumulate all the ping samples in the window into a `list` called `ping_samples_db` with

```
ping_samples_db = [ping.data for ping in first_input.window_measurements]
```

which uses Python's `list` comprehension feature to create

```
ping_samples_db = [[1, 2, 1, 2, 1],
                   [2, 1, 2, 1, 2],
                   [1, 2, 1, 2, 1],
                   [2, 1, 2, 1, 2],
                   [1, 2, 1, 2, 1],
                   [2, 1, 2, 1, 2],
                   [1, 2, 1, 2, 1]]
```

If you are unclear with this output, identify ping 4 (the matched ping) and the adjacent pings in the window of measurements in Figure 2. Then compare the values for the ping samples reported in the Color legend to the numbers in `ping_samples_db`.

Manipulating such arrays is critical for harnessing the Code operator.

You also need to know how to identify the matched ping in such arrays. Use

```
return ping_samples_db[first_input.measurement_index]
```

to select the matched ping in `ping_samples_db` and `return` it via `eval`.

13.3 The shape of an array

We can use NumPy's `shape` method to determine how many ping samples are in an array.

For single beam echosounders the array of ping samples has shape (range), e.g.,

```
first_input.measurement.data = [1, 2, 1, 2, 1]
```

Since the elements correspond to data values as a function of range, it is a one-dimensional array.

```
np.shape(first_input.measurement.data)
```

will return `(5,)`—a Python `tuple`.

With `ping_samples_db` at the end of section 13.1, we created an array with shape (pings, range). Then

```
np.shape(ping_samples_db)
```

returns `(7, 5)`, a two-dimensional array.

You can then utilize Python's indexing to extract the number of elements from this `tuple`. To obtain the number of pings, use

```
np.shape(ping_samples_db)[0]
```

and to determine the number of ping samples use

```
np.shape(ping_samples_db)[1]
```

The `axis` parameter in many NumPy functions offers a convenient way to select the particular dimension in such arrays. For example, to compute the average of the samples along the ping axis use

```
np.mean(ping_samples_db, axis=0) =
                        [1.42857143, 1.57142857, 1.42857143, 1.57142857, 1.42857143]
```

To compute the average along the range axis use

```
np.mean(ping_samples_db, axis=1) = [1.4, 1.6, 1.4, 1.6, 1.4, 1.6, 1.4]
```

Try to answer the question: given `ping_samples_db` at the end of section 13.1, which of these averages can you `return` to Echoview with `eval` without triggering an error? We will see the solution at the end of Exercise 1a—Mean operator.

Part III

Exercises

14 Prelude to the exercises

These exercises are designed to give you some experience in thinking through an algorithm and implementing it in the Code operator. Depending on your familiarity with Python, the Code operator and programming in general, we recommend you set aside at least an hour or more to complete them. We also recommend that you attempt the exercises in order as they increase in complexity down the list.

14.1 Obtaining the exercises

Navigate into the `Exercises` folder that you downloaded with this tutorial, by default

```
C:\Echoview Software\Tutorials\IntroToCodeOperator
```

Inside you will find the following sub-folders

`1a-Mean-operator`
`1b-Mean-operator`
`2-Gaussian-blur-operator` (challenging)
`3-Multibeam-circle-filter` (challenging)

Each of these contains some echosounder data, an Echoview Python source file template (`-exercise.py`) and the Echoview (`.EV`) file for the exercise. You will also find the solution Echoview Python source file (`-solution.py`) for each exercise.

14.2 Completing the exercises

To complete the exercises, open any of the `-exercise.py` Echoview Python source file templates. These contain `'''triple single-quoted'''` text which provide hints to the solution Python code. Replace all `'''triple single-quoted'''` text and quotes with your Python code before loading the `-exercise.py` into the Code operator. If you get stuck, you can find the answer in the corresponding `-solution.py` file.

Tabs vs spaces

The Echoview Python source files you have encountered so far utilize tabs for Python code block indentation (by default). Note that we use spaces instead for all the Echoview Python source files in the exercises.

14.3 Using the NumPy and SciPy packages

These widely used Python packages come preinstalled with Echoview, and we recommend you utilize the numerous built-in functions they provide for operations. You can find the documentation for them at

https://numpy.org/doc/stable/search.html
https://scipy.github.io/devdocs/search.html

As a task, we recommend you visit NumPy's site and search for "numpy.mean" to read and understand the documentation about the averaging function we used just now in section 13.3.

14.4 Programming principles

While guiding you through these exercises, we have attempted to adhere to generally accepted programming principles. There will be alternative solutions that are ostensibly more straightforward and suitable, and you are encouraged to experiment with them as well.

15 Exercise 1a—Mean operator

In the first exercise, we will use the window of measurements (section 9) to implement Echoview's XxY statistic operator's mean algorithm. After that, in section 15.5.2, we will also conduct a test to verify our program.

The Echoview files and Python source files for this exercise are in the `1a-Mean-operator` folder.

15.1 Algorithm

This program computes the mean ($\bar{x}$) of the sample data in the ping dimension for the number of pings specified in Window size (pings) according to

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1}$$

Here, x_i are the ping sample values in linear units, and n is the number of ping samples.

15.2 Setting up the Code operator

1. Open `Mean of pings 1a.EV` in Echoview and in the Dataflow window locate **Exercise 1a - Mean of pings**. This is a Code operator variable.
2. Right-click on this and select Variable properties from the drop-down menu (or left-click and press F8) to open the Variable properties dialog box.
   a. Navigate to the Operands page (Figure 10) and in the Operands section select

      ```
      Fileset 1: Sv pings
      ```

      as Operand 1.
   b. Click on the Code page (Figure 5) then
      i. specify the path to `mean-operator-exercise.py` in the Echoview Python source file text box, and
      ii. set Window size (pings) to 5

   Click **OK** to save these settings and close the Variable Properties dialog box.

15.3 Setting up the Echoview Python source file

To complete this exercise, code the following procedure into the `eval` method in `mean-operator-exercise.py`. You need to replace any triple single-quoted text (e.g., *`'''Ping samples'''`*) with your answer.

15.3.1 Procedure

1. In `eval` assign the input from Operand 1 to the variable

   `first_input`

2. Next we will use Python's list comprehension feature (refer to the example in section 13.2) to create a `list` and populate it with the ping samples from the pings within the window.

   Assign this `list` to the variable

   `ping_samples_db`

3. We will be averaging the ping samples in the linear domain according to the formula

   $$10^{\frac{X}{10}}$$

   where X is the ping samples in dB units. Utilize NumPy's `power` and `divide` functions to implement this formula (attempt the computation in a single line of code) and assign the result to

   `ping_samples_linear`

4. Now compute the mean of the ping samples (equation 1) and assign the result to

   `mean_linear`

5. Convert the averaged result back to units of dB according to

   $$10 \log_{10} X$$

   where X is the ping samples in linear units. Utilize NumPy's `log10` and `multiply` functions to implement this formula (again, attempt the computation in a single line of code) and assign the result to

   `mean_db`

6. Lastly, use the `return` statement to communicate `mean_db` to Echoview.

You can verify your answers against `mean-operator-solution.py`.

15.4 Executing the Echoview Python source file

Back in Echoview double-click on Exercise 1a - Mean of pings to execute `mean-operator-exercise.py` and open the echogram.

Compare the result with the echogram for Fileset 1: Sv pings shown in Figure 14 and see that the data samples are smeared in the ping dimension.

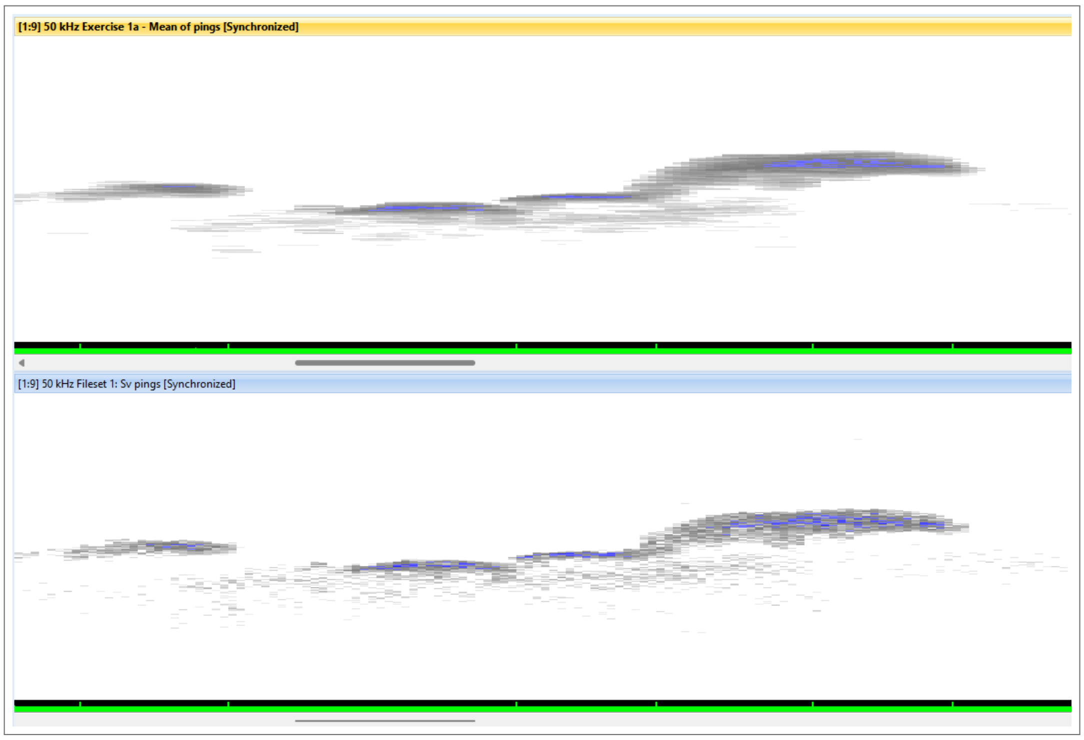

[1:9] 50 kHz Exercise 1a - Mean of pings [Synchronized]
[1:9] 50 kHz Fileset 1: Sv pings [Synchronized]

Figure 14: Echograms for the Code operator Exercise 1a - Mean of pings (top) and Fileset 1: Sv pings (bottom).

15.5 Comparing the result with the XxY statistic operator

We will now use a Minus operator to test Exercise 1a - Mean of pings against Echoview's XxY statistic operator.

15.5.1 Creating an XxY statistic operator to replicate the Code operator

1. In Echoview create a XxY Statistic operator and rename it to XxY statistic - Mean of pings.
2. On the Variable Properties dialog box
   a. click onto the Operands page, and under the Operands section for Operand 1 choose

      ```
      Fileset 1: Sv pings
      ```

      from the drop-down menu.

b. click onto the XxY Statistic page and under the Settings section and set
   i. Statistic to Mean
   ii. Rows (samples) to 1
   iii. Columns (pings) to 5 (identical to the value in step 2(b.)ii in section 15.2.)

c. Click OK to save the changes and close the dialog box. Your Dataflow window should resemble Figure 15.

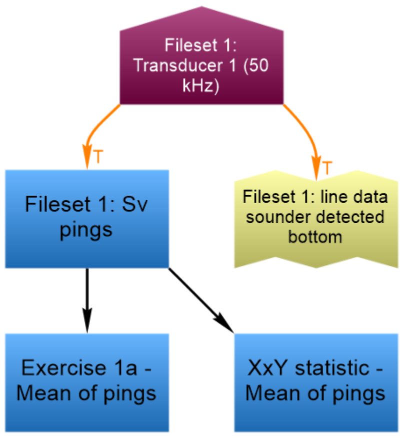

Fileset 1: Transducer 1 (50 kHz)
T
T
Fileset 1: Sv pings
Fileset 1: line data sounder detected bottom
Exercise 1a - Mean of pings
XxY statistic - Mean of pings

Figure 15

Double-click on XxY statistic - Mean of pings to open the echogram and compare it to the echogram for Exercise 1a - Mean of pings. They will be identical.

15.5.2 Creating a Minus operator to test the difference

We will use the Minus operator to compare Exercise 1a - Mean of pings and XxY statistic - Mean of pings.

3. Create a Minus operator and rename it to Difference.

4. On the Variable Properties dialog box click onto the Operands page, and under the Operands section for Operand 1 choose

   ```
   Exercise 1a - Mean of pings
   ```

   from the drop-down menu for Operand 1, and

   ```
   XxY statistic - Mean of pings
   ```

   from the drop-down menu for Operand 2.

5. Click OK to save the changes and close the dialog box. Your Dataflow window should

resemble Figure 16.

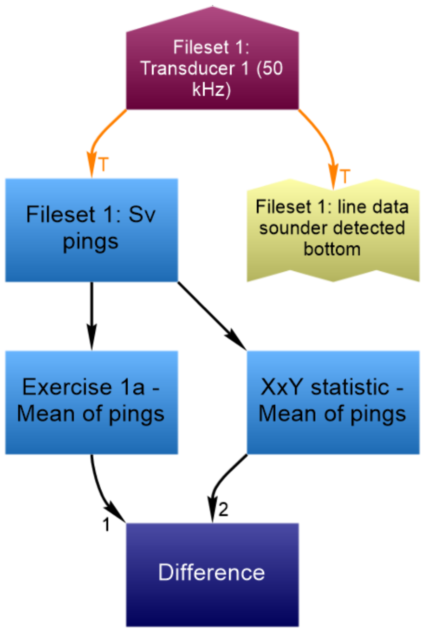

Fileset 1: Transducer 1 (50 kHz)
T
T
Fileset 1: Sv pings
Fileset 1: line data sounder detected bottom
Exercise 1a - Mean of pings
XxY statistic - Mean of pings
1
2
Difference

Figure 16

15.5.3 Visualizing the difference between the XxY statistic and Code operator

On the Dataflow window, double-click on Difference to view its echogram. Set Display minimum to -0.001 dB and Display range to 0.002 dB on the Color legend to reveal any minute deviations.

The echogram for Difference will resemble Figure 17—a uniform color indicating no difference between Exercise 1a - Mean of pings and XxY statistic - Mean of pings .

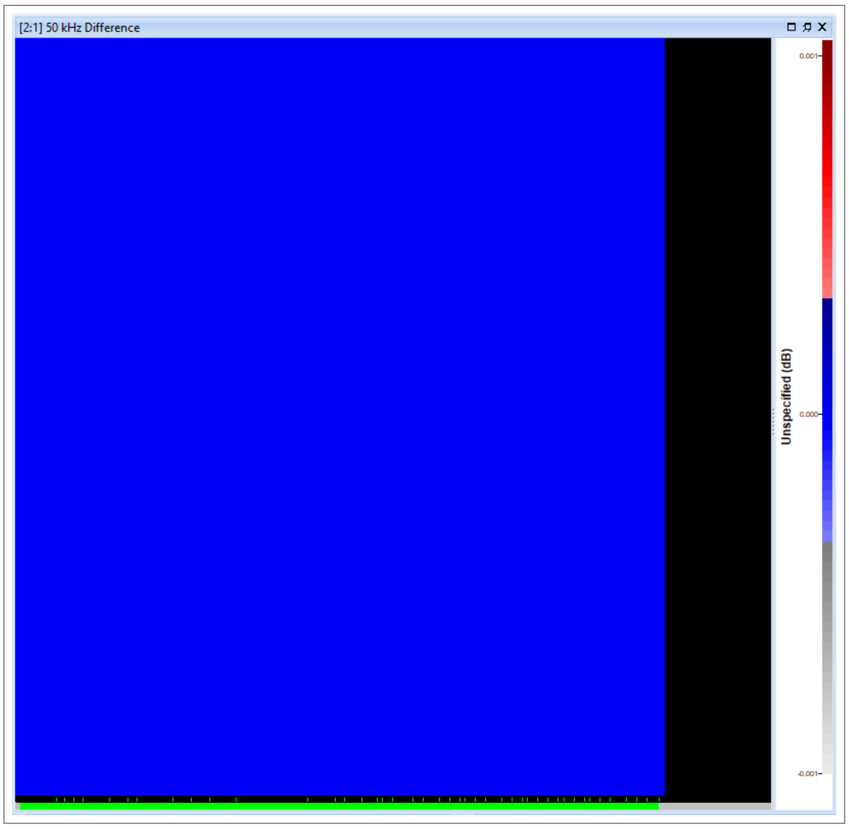


Figure 17: Echogram showing no differences in the ping samples between Exercise 1a - Mean of pings and XxY statistic - Mean of pings .

15.6 Afterword

You can now close the `.EV` and `.py` files for this exercise.

In the next exercise, we will add XxY statistic - Mean of pings as Operand 2 to a Code operator variable. We will then extend the program from this exercise to perform the subtraction in section 15.5.

16 Exercise 1b—Mean operator

This exercise is based on the previous one. However, we will verify that our Code operator algorithm is identical to the one in the XxY Statistic operator with the Code operator itself, instead of the Minus operator.

The Echoview files and Python source files for this exercise are in the `1b-Mean-operator` folder.

16.1 Algorithm

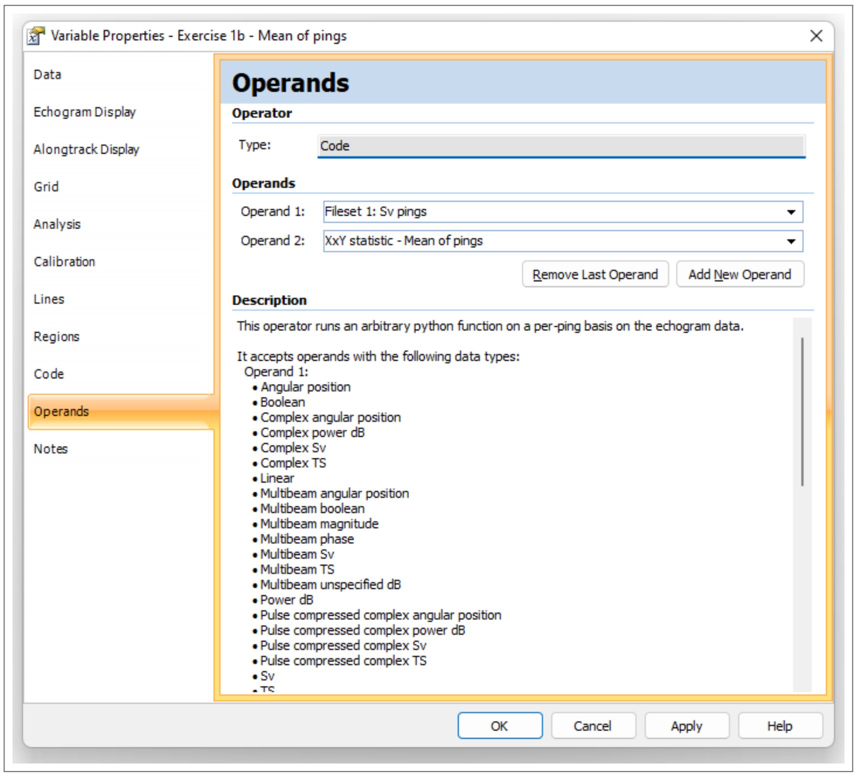
Subtract the XxY statistic - Mean of pings samples from the output of the Exercise 1b - Mean of pings Code operator Echoview variable, and test if the difference is less than a threshold of 10^{-10} dB. Then `return` the Boolean value `True` if so and `False` otherwise.

16.2 Setting up the Code operator

1. Open `Mean of pings 1b.EV` in Echoview and in the Dataflow window locate Exercise 1b - Mean of pings. This is a Code operator variable.
2. Right-click on this and select Variable properties from the drop-down menu (or left-click and press F8) to open the Variable properties dialog box.
   a. Navigate to the Operands page (Figure 10) and in the Operands section
      i. click on the Add New Operand button, and
      ii. from the drop-down menu, choose

         ```
         XxY statistic - Mean of pings
         ```

         as Operand 2 (Figure 18).
         Your Dataflow window should now resemble Figure 19.


Variable Properties - Exercise 1b - Mean of pings

Operands

Operator

Type: Code

Operands

Operand 1: Fileset 1: Sv pings

Operand 2: XxY statistic - Mean of pings

Description

This operator runs an arbitrary python function on a per-ping basis on the echogram data.

It accepts operands with the following data types:
Operand 1:
- Angular position
- Boolean
- Complex angular position
- Complex power dB
- Complex Sv
- Complex TS
- Linear
- Multibeam angular position
- Multibeam boolean
- Multibeam magnitude
- Multibeam phase
- Multibeam Sv
- Multibeam TS
- Multibeam unspecified dB
- Power dB
- Pulse compressed complex angular position
- Pulse compressed complex power dB
- Pulse compressed complex Sv
- Pulse compressed complex TS
- Sv

Figure 18: Exercise 1b - Mean of pings with `Fileset 1: Sv pings` as Operand 1 and `XxY statistic - Mean of pings` as Operand 2.

b. Navigate to the Code page (Figure 5) then

   i. specify the path to `mean-algorithm-comparison-exercise.py` in the Echoview Python source file text box, and

   ii. set Window size (pings) to 5

Click **OK** to save these settings and close the Variable Properties dialog box.

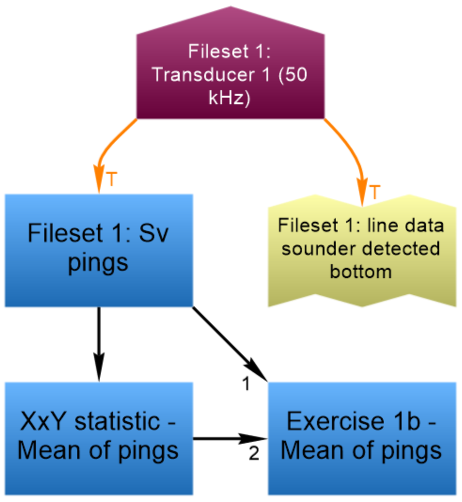

Fileset 1: Transducer 1 (50 kHz)
T
T
Fileset 1: Sv pings
Fileset 1: line data sounder detected bottom
1
XxY statistic - Mean of pings
2
Exercise 1b - Mean of pings

Figure 19

16.3 Setting up the XxY statistic operator

In the Dataflow window (Figure 19) locate XxY statistic - Mean of pings. Open its Variable Properties dialog box and follow the procedure from step 2 in section 15.5.1 to set it up.

16.4 Setting up the Echoview Python source file

To complete this exercise, code the following procedure into the `Operator` `class` in `mean-algorithm-comparison-exercise.py`. You need to replace any triple single-quoted text (e.g., *`'''Ping samples'''`*) with your answer.

Procedure

1. Firstly, program the threshold as an attribute of `Operator`. Use the `__init__` method[21] and create an attribute called

   ```
   self.threshold
   ```

   Assign the value 10^{-10} (i.e., the threshold value in dB) to it.

Now search in the `eval` method for the line of code

```
mean_db = np.multiply(np.log10(mean_linear), 10)
```

and continue with the following steps.

(We are proceeding from the point corresponding to step 5 from section 15.3.)

[21]We first encountered the `__init__` method in section 11.

2. Obtain the ping sample data for XxY statistic - Mean of pings and assign the result to

   `xxy_statistic_samples`

3. Next use NumPy's `subtract` method to perform the element-wise subtraction between `mean_db` and `xxy_statistic_samples` and assign the result to

   `difference`

4. Then call NumPy's `where` method[22] to test when the magnitude of the elements in `difference` are less than `self.threshold`.

   `return` the Boolean values `True` if so and `False` otherwise.

5. Finally, define the `result_type` method for the `Operator class` and choose the appropriate data type attribute from the `MeasurementType class` (appendix C.4) to `return` to Echoview.

You may verify your answers against `mean-algorithm-comparison-solution.py`.

16.5 Executing the Echoview Python source file

When you save your changes in `mean-algorithm-comparison-exercise.py` the color of the Code operator variable will change to Exercise 1b - Mean of pings in the Dataflow window to indicate a Boolean Echoview variable.

Double-click on it to open its echogram. You will see that all samples are True (Figure 20), demonstrating the means, calculated using Exercise 1b - Mean of pings and XxY statistic - Mean of pings differ by less than our threshold of 10^{-10} dB.

[22] This works like a combination of a `for` loop combined with an `if` statement. If you are unsure how it works, refer to the documentation on NumPy's website or to the examples in appendix D.

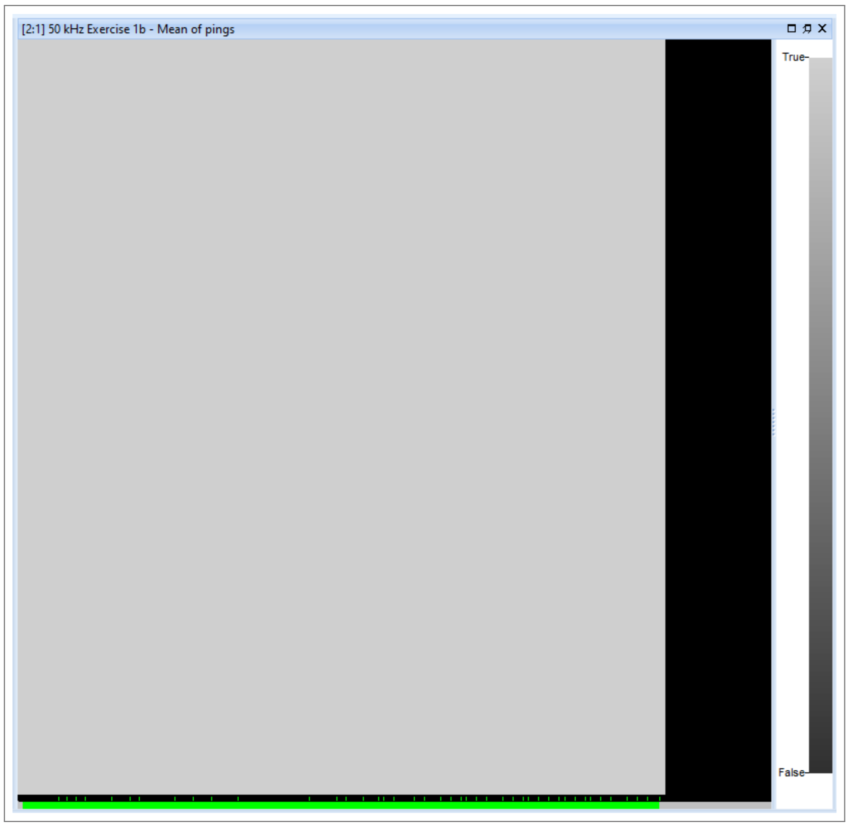


Figure 20: The Echogram populated with `True` for every sample demonstrates the means, calculated using Exercise 1b - Mean of pings and XxY statistic - Mean of pings, differ by less than our threshold of 10^{-10} dB.

17 Exercise 2—Gaussian blur operator

In this exercise you will implement a generalized version of Echoview's XxY convolution operator's Gaussian blur algorithm. You can use this program to fine-tune the Gaussian function used for computing the convolution kernel weights.

A key takeaway from this exercise is that you can replace the Gaussian (equation 3) with any function to use as a filter for your ping data.

The Echoview files and Python source files for this exercise are in the `2-Gaussian-blur-operator` folder.

17.1 Algorithm

This program convolves ping samples in the Code operator's window of measurements (section 9) with a kernel (e.g., the 3$\times$3 kernel in Figure 21) according to

$$f(n_1,n_2) ** g(n_1,n_2) = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} g(k_1,k_2)\, f(n_1-k_1, n_2-k_2) \tag{2}$$

Here, $f(n_1,n_2)$ is the input echogram and $g(n_1,n_2)$ is the convolution kernel, where (n_1,n_2) represents the ping samples' coordinates and (k_1,k_2) represents the kernel coordinates. The kernel values themselves we compute using the normalized[23] 2 D Gaussian equation

$$g(k_1,k_2) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\left[\frac{(k_1-\mu_x)^2}{2\sigma_x^2}+\frac{(k_2-\mu_y)^2}{2\sigma_y^2}\right]} \tag{3}$$

We will define what the variables in equations (2) and (3) represent in step 2 in section 17.3.

17.2 Setting up the Code operator

1. Open `Gaussian blur operator.EV` in Echoview and in the Dataflow window locate Exercise 2 - Gauss operator. This is a Code operator variable.
2. Right-click on this and select Variable properties from the drop-down menu (or left-click and press F8) to open the Variable properties dialog box.
    a. Navigate to the Operands page (Figure 10) and in the Operands section select

    ```
    Fileset 1: Sv pings
    ```

    as Operand 1.

[23]Normalization preserves the magnitude of the output from the convolution vis-à-vis the input. Here $\frac{1}{2\pi\sigma_x\sigma_y}$ accounts for this. However, note that this factor (obtained by integrating the 2 D Gaussian) assumes a continuous function. Since our kernel is discrete, there will be an error in the normalization. The magnitude of this error is inversely proportional to the kernel size.

b. Click on the Code page (Figure 5) then

   i. specify the path to `gaussian-blur-operator-exercise.py` in the Echoview Python source file text box, and

   ii. set Window size (pings) to 5
   Generally, this value should be greater than or equal to the value assigned to k_1 in equation 2 to ensure that the kernel is not wider than the number of pings in the window.

Click **OK** to save these settings and close the Variable Properties dialog box.

17.3 Setting up the Echoview Python source file

To complete this exercise, code the following procedure into the `Operator class` in `gaussian-blur-operator-exercise.py`. You need to replace any triple single-quoted text (e.g., `'''Ping samples'''`) with your answer.

You will be implementing the Python commands according to the following stages.

1. Groundwork
2. Set up the kernel grid
3. Compute the Gaussian weights for the kernel
4. Perform the convolution

Groundwork

1. To perform the convolution in equation 2, we will use the `convolve2d` function. We have imported it for you with the

   ```
   from scipy.signal import convolve2d
   ```

   statement near the top of the file.

2. In `Operator` use `__init__` to define the kernel dimensions that constrain k_1 and k_2 in equation 2, and to fix μ_x, μ_y, σ_x and σ_y for the 2 D Gaussian in equation 3. Use these variable names in `gaussian-blur-operator-exercise.py`

k_1	`kernel_column_span`
k_2	`kernel_row_span`
μ_x	`offset_x`
μ_y	`offset_y`
σ_x	`standard_deviation_x`
σ_y	`standard_deviation_y`

3. In `eval` start by compiling a `list` of ping samples in linear units (follow steps 1—3 from section 15.3) in the window and assign it to

   ```
   ping_samples_linear
   ```

Set up the kernel grid

4. The grid for the kernel is a Python function. Outside of `Operator` and after `eval` create a function called

   `kernel_coordinates`

   that accepts two parameters specifying the kernel's dimensions

   `kernel_column_span`
   `kernel_row_span`

   `kernel_coordinates` will correspond to k_1 and k_2 in equation 2.

To illustrate what we want to achieve, take the example when `kernel_row_span` = `kernel_column_span` = 3 as shown in Figure 21.

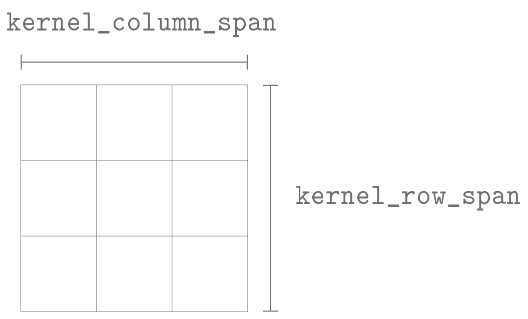
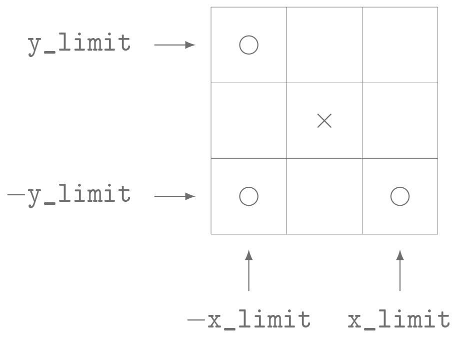


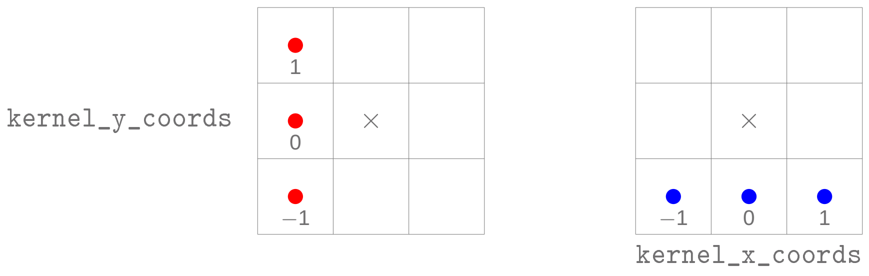
Figure 21: A schematic of a 3×3 convolution kernel. In general, the kernel can comprise as many (odd-numbered) rows and columns as you want.

5. With `kernel_column_span` and `kernel_row_span` deduce the kernel's `x_limit` and `y_limit` relative to the center at (0, 0)[24] marked with ×

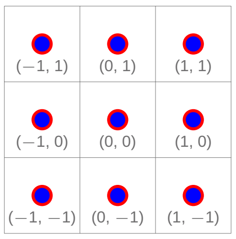


[24]Setting up the kernel relative to these coordinates ensures the 2 D Gaussian will yield weights that are symmetric about the origin.

6. Given the kernel limits, use NumPy's `linspace` to generate each axis assigning the output to `kernel_x_coords` and `kernel_y_coords`



7. `return` the coordinate pairs via NumPy's `meshgrid` which computes the coordinate pairs for the x and y axes.



(Since the kernel is essentially an array of coordinates, this is not the only way of constructing it. There are numerous ways of generating arrays to achieve the same result.)

Compute the Gaussian weights for the kernel

8. We will use another Python function to implement equation 3, to compute the weights for the kernel. Outside of `Operator` create a function called

   `normalized_gaussian_kernel`

   that accepts all the parameters from `__init__` (step 2)

9. Now program

   `normalized_gaussian_kernel`

   to call

   `kernel_coordinates` (step 4)

   and utilize Python's multiple assignment feature to assign the output to `x_coords` and `y_coords`.

10. Then `return` the 2 D Gaussian described by equation 3.

    In this `return` statement, `x_coords` and `y_coords` from the previous step, and the parameters from `__init__` substitute into x, y, μ_x, μ_y, σ_x and σ_y in equation 3.

Perform the convolution

11. Back in `eval` call `normalized_gaussian_kernel` using the attributes in `__init__` as the parameters. Assign the result to

    `kernel`

12. Now perform the convolution by passing `ping_samples_linear` and `kernel` into `convolve2d` (which we imported in step 1). Assign the result to

    `convolution_linear`

13. Convert the result back to units of dB (refer to step 5 in section 15.3). Assign this result to

    `convolution_db`

14. Lastly, use the `return` statement to communicate the matched ping in `convolution_db` to Echoview (refer to section 13 for the example syntax).

You may verify your answers against `gaussian-blur-operator-solution.py`.

17.4 Executing the Echoview Python source file

Back in Echoview double-click on Exercise 2 - Gauss operator to execute the Echoview Python source file `gaussian-blur-operator-exercise.py` and open the echogram.

18 Exercise 3—Multibeam circle filter

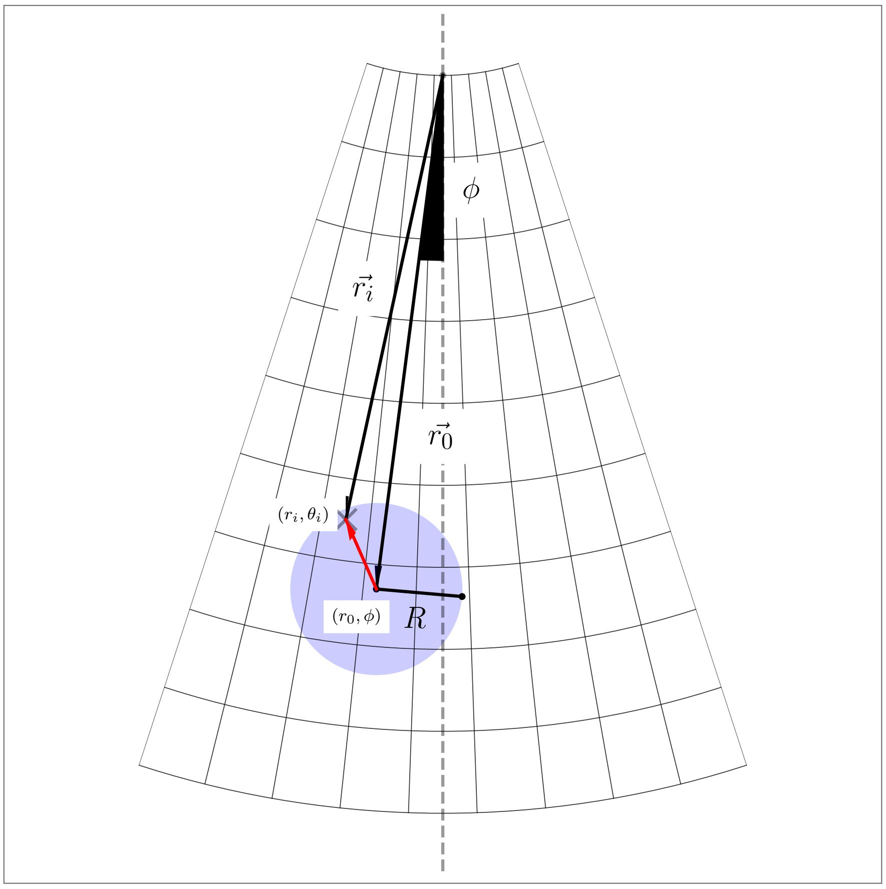
In this exercise you will form a circular mask on a multibeam echogram and set the ping samples within that mask to no data[25].

Since Echoview does not currently support regions for multibeam data, this exercise demonstrates how you can utilize the Code operator to exploit certain features available to regions in your multibeam analysis.

The Echoview files and Python source files for this exercise are in the `3-Multibeam-circle-filter` folder.

18.1 Algorithm

Refer to Figure 22. Using a polar coordinate system, choose the center of the circular filter to be a point (r_0, ϕ) in the multibeam fan given by the position vector $\vec{r_0}$. The center of each sample in the ping has coordinates (r_i, θ_i), given by the position vector $\vec{r_i}$. For a circular filter of radius R, if $|\vec{r_0} - \vec{r_i}| < R$, set ping sample $\vec{r_i}$ to no data.



[25]Refer to the About no data online page Help page)

Figure 22: This describes a multibeam fan comprising 9 beams, with 9 samples per beam. We want a circular mask with center at (r_0, ϕ) and radius R. Our algorithm will test the vector to the center of each sample $\vec{r_i}$ given by (r_i, θ_i) according to $|\vec{r_0} - \vec{r_i}|$, i.e., the magnitude of the red vector. If the result is less than R, set sample $\vec{r_i}$ to no data.

18.2 Setting up the Code operator

1. Open `Multibeam circle filter.EV` in Echoview and in the Dataflow window locate Exercise 3 - Multibeam circle filter. This is a Code operator variable.
2. Right-click on this and select Variable properties from the drop-down menu (or left-click and press F8) to open the Variable properties dialog box.
   a. Navigate to the Operands page (Figure 10) and in the Operands section select

      `Fileset 1: Sv frames`

      as Operand 1.
   b. Click on the Code page (Figure 5) then
      i. specify the path to `multibeam-circle-filter-exercise.py` in the Echoview Python source file text box, and
      ii. set Window size (pings) to 1

   Click **OK** to save these settings and close the Variable Properties dialog box.

18.3 Setting up the Echoview Python source file

To complete this exercise, code the following procedure into the `Operator` `class` in `multibeam-circle-filter-exercise.py`. You need to replace any triple single-quoted text (e.g., `'''Ping samples'''`) with your answer.

Groundwork

1. In `Operator` use `__init__` (appendix B.2.2) to specify the center (r_0, ϕ) of the circular mask in the multibeam fan and its radius R. Use these variable names in your Echoview Python source file

r_0	`center_range`
ϕ	`center_angle`
R	`radius`

Determining the coordinates of each sample

To compute the coordinates (r_i, θ_i) of each sample in the ping we will

- firstly subtract the start and stop ranges of the ping and divide it by the number of samples to get the "thickness" of each sample to compute r_i, then
- use the beam angles provided by the `Measurement` `class` to obtain θ_i.

2. In `eval` assign Operand 1 to

   `first_input`

   and the matched ping to

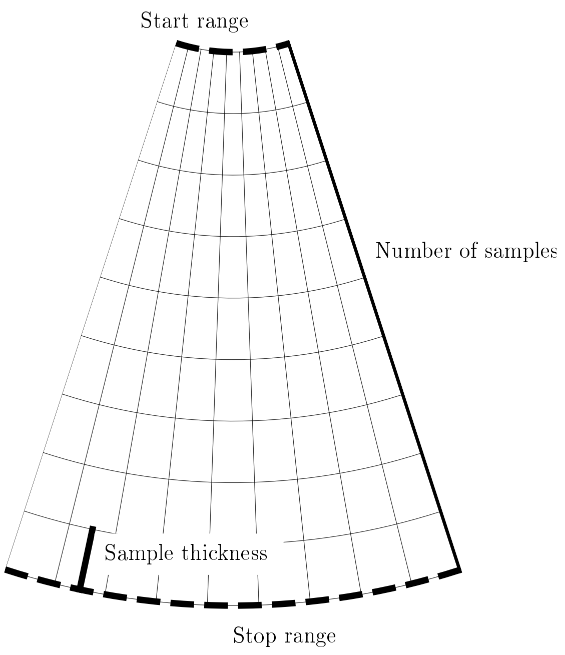
   `ping`

   Since this is a multibeam ping, `ping.data` has shape (range, beams).

3. Use NumPy's `shape` method to find the shape (section 13.3) of the ping. This method returns a `tuple` indicating the number of elements in each dimension. Assign the number of samples in the range dimension to

   `sample_count`

4. Assuming all the samples span an equal range, use $\frac{\text{Start range}-\text{Stop range}}{\text{Number of samples}}$ to compute the 'thickness' of each sample.



   Assign the value to

   `sample_thickness`

   Next halve `sample_thickness` to get the distance to the center of a sample and assign it to

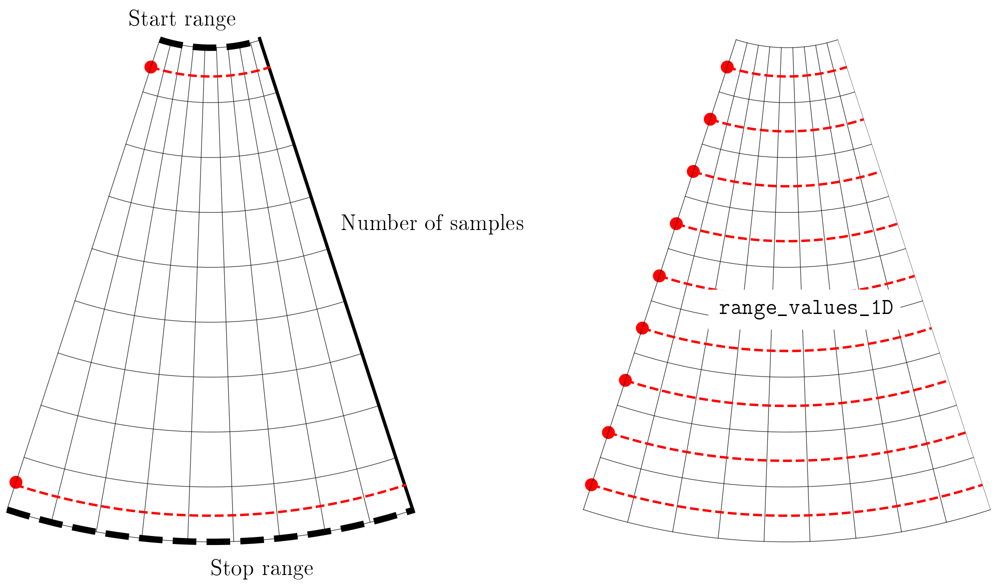
`half_thickness`

5. Then use `half_thickness` to find the range to the center of the first sample and the range to the center of the last sample.

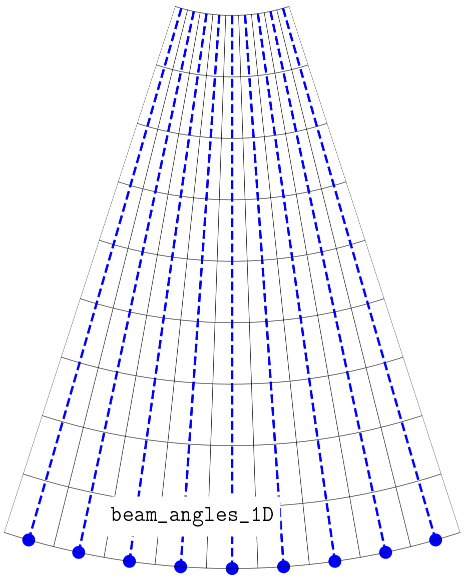
   To compute r_i for the samples in between, insert these values into NumPy's `linspace` function along with `sample_count` from step 3. Assign this result to

   `range_values_1D`

   We now have the r_i values for all the samples in the ping.



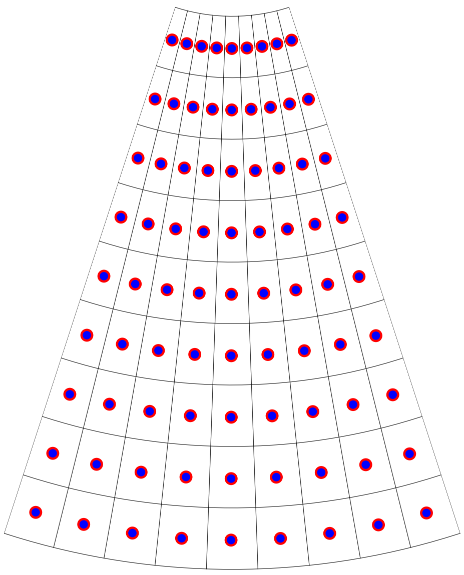
6. Use the `beam_angle` attribute (refer to the Using the Code operator online Help page) to obtain the angle at the center of each beam.



This is in degrees. Convert it using NumPy's `radians` function (for the trigonometric computations later on) and assign the result to

`beam_angles_1D`

7. Finally, use `meshgrid` with `range_values_1D` and `beam_angles_1D` to create a 2 D lattice corresponding to (r_i, θ_i). Utilize Python's multiple assignment feature to designate the output to `sample_ranges` and `sample_angles`.



Each point represents the (r_i, θ_i) coordinates of the vector $\vec{r_i}$ (Figure 22).

Testing if the sample is within the circular mask

8. Compute $|\vec{r_0} - \vec{r_i}|$ which is the distance between the center of the circular mask (vector $\vec{r_0}$) and center of each ping sample (vector $\vec{r_i}$).

   An easy way is to convert the coordinates of the vectors from polar (r, θ) to cartesian (x, y) coordinates and subtract the x and y components:

   $$(\vec{r_0} - \vec{r_i})_x = |r_0| \cos \phi - |r_i| \cos \theta_i$$
   $$(\vec{r_0} - \vec{r_i})_y = |r_0| \sin \phi - |r_i| \sin \theta_i$$

   then

   $$|\vec{r_0} - \vec{r_i}| = \sqrt{(\vec{r_0} - \vec{r_i})_x{}^2 + (\vec{r_0} - \vec{r_i})_y{}^2}$$

   Assign the result to

   `distance_2D`

9. Finally use NumPy's `where` method (appendix D) to `return` the result of the test $|\vec{r_0} - \vec{r_i}| < R$.

   a. If the condition is satisfied, set (r_i, θ_i) to NumPy's `nan`—which is interpreted as no data in Echoview.

   b. If the condition is not satisfied, communicate the unfiltered ping sample to Echoview.

You may verify your answers against `multibeam-circle-filter-solution.py`.

18.4 Executing the Echoview Python source file

Back in Echoview double-click on Exercise 3 - Multibeam circle filter to execute the Echoview Python source file `multibeam-circle-filter-exercise.py` and open the echogram. Navigate through the pings of the echogram to see the mask of no data regions in all the pings as shown in Figure 23.

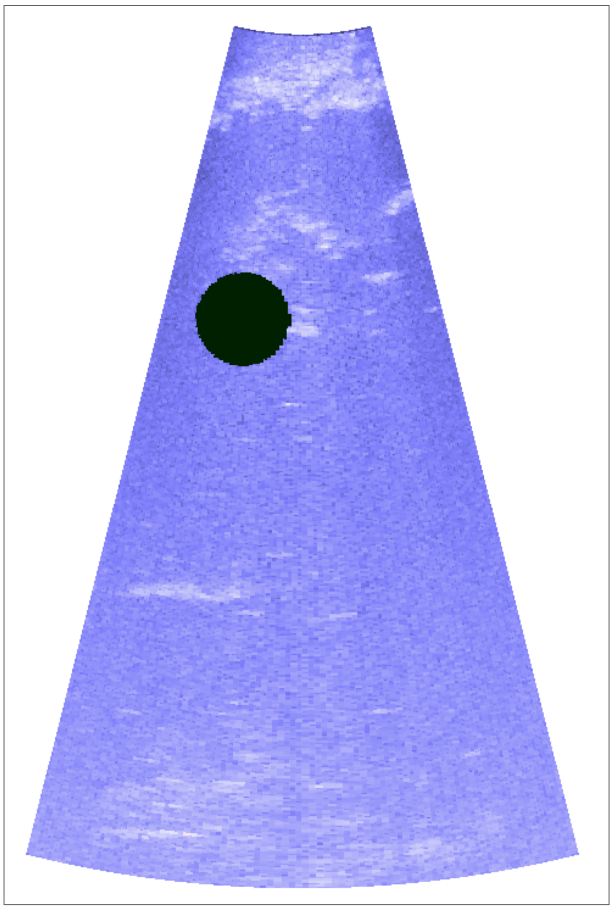


Figure 23

Part IV

Appendix

A Complete Echoview Python source files

Here are the complete versions of the practice Echoview Python source file programs from this tutorial.

Fonts in green (such as `def`) are reserved keywords, meaning we cannot use them as variable names. `self` (appendix B.3) is an exception and is technically not a Python keyword, however, it is accepted as one in practice.

A.1 About the commands in the default Echoview Python source file (section 2.1.2)

```
from typing import List
import echoview as ev
import numpy as np

class Operator(ev.OperatorBase):

    def eval(self, inputs: List[ev.OperandInput]):
        first_input = inputs[0]
        return first_input.measurement.data
```

A.2 Practice—Loading additional operands into the Code operator (section 5)

```
from typing import List
import echoview as ev
import numpy as np

class Operator(ev.OperatorBase):

    def result_type(self, input_types: List[ev.MeasurementType]):
        return input_types[1]

    def eval(self, inputs: List[ev.OperandInput]):
        first_input = inputs[0]
        second_input = inputs[1]
        return second_input.measurement.data
```

A.3 Practice—Working with ping samples from different operands (section 7)

```python
from typing import List
import echoview as ev
import numpy as np

class Operator(ev.OperatorBase):

    def result_type(self, input_types: List[ev.MeasurementType]):
        return ev.MeasurementType.SINGLE_BEAM_LINEAR

    def eval(self, inputs: List[ev.OperandInput]):
        first_input = inputs[0]
        second_input = inputs[1]
        targets_per_unit_volume_log = first_input.measurement.data - second_input.measurement.data
        targets_per_unit_volume_linear = 10**(targets_per_unit_volume_log/10)
        return targets_per_unit_volume_linear
```

A.4 Practice—Outputting messages to a log (section 11)

```python
from typing import List
import echoview as ev
import numpy as np

class Operator(ev.OperatorBase):

    def __init__(self):
        self.logfile = "C:\\path_to\\code-operator-window-edges\\window-edges.log"

    def eval(self, inputs: List[ev.OperandInput]):
        first_input = inputs[0]

        current_window_length = len(first_input.window_measurements)
        with open(self.logfile, 'a') as log:
            log.writelines(
                f"Matched ping: {first_input.measurement.index}\n "
                f"Current window length: {current_window_length}/{self.window_size}\n "
                f"Pings in window:\n ")
            for ping in first_input.window_measurements:
                log.writelines(f"\t Ping {ping.index}\n ")
            log.writelines(f"\n ")
        return first_input.measurement.data
```

B The Code operator and object-oriented programming

The Code operator is developed according to a paradigm in computer science known as object-oriented programming (OOP). In this section, we will introduce some OOP concepts as they apply to the Code operator.

B.1 What is an “object”?

Refer to the default Echoview Python source file (appendix A.1). The code from the `class` keyword up to the colon informs you of the object type. This and the indented code aligned to it is a blueprint of the object. When we instantiate[26] it—in our case by creating a Code operator in Echoview—an instance of that blueprint manifests, i.e.,

Code $\Longleftrightarrow$ `class Operator(ev.OperatorBase):`

The Python syntax

```
Operator(ev.OperatorBase)
```

means we are defining a `class` called `Operator` that inherits all its attributes and methods from `OperatorBase`[27].

B.2 Attributes and methods

Objects have two features called attributes and methods (next section). These characterize the state of the object when it is instantiated and its behavior respectively. Both are defined within the `class` body.

B.2.1 Programming a method for the Code operator

You can program the `eval` and `result_type` methods (appendix C.1.1 and C.1.2), which Echoview calls on `Operator`.

In Python, you use the syntax for programming functions to create methods as well, for example,

```
class Operator(ev.OperatorBase):
    def eval(self, inputs: List[ev.OperandInput]):
        ...
        ...
```

This defines the `eval` method for the `Operator` `class`.

[26]The act of allocating computer memory to create an object.
[27]`OperatorBase` is an abstract base class from the `echoview` package (appendix C) which we `import` at the beginning of the Echoview Python source file.

The `return` keyword

If it is specified in the method, the `return` statement e.g.,

```
def result_type(self, input_types: List[ev.MeasurementType]):
    return input_types[0]
```

communicates the value that follows the `return` keyword back to the caller code and terminates the method. This statement is needed for both the `eval` and `result_type` methods.

B.2.2 Programming an attribute for the Code operator

You may define the `__init__` method under the `Operator class` in your Echoview Python source file to create customized attributes for your Code operator object.

```
def __init__(self):
```

Note that `Operator` only accepts the `self` (appendix B.3) parameter in `__init__`. This is because Echoview initializes the Code operator in a standardized way. Any custom parameters will be unknown to Echoview.

The `window_size` attribute

Note that Echoview creates the `window_size` attribute (section 9) you do not define it in `__init__` in the `Operator class`.

B.2.3 Calling an attribute or method

Python utilizes the dot notation to get or set an attribute, or call a method on an object. For example, refer to this code snippet defining the `result_type` method.

```
def result_type(self, input_types: List[ev.OperandInput]):
    if input_types[0].is_acoustic:
        return input_types[0]
    else:
        return ev.MeasurementType.UNDEFINED
```

In the line `if input_types[0].is_acoustic:` is calling the `is_acoustic` method on `input_types[0]`

```
input_types[0]
```

is an object of the `MeasurementType class` (appendix C.4) and has a method called

```
is_acoustic
```

which returns a Boolean.

Similarly, in the line

```
return ev.MeasurementType.UNDEFINED
```

we use the dot notation to set the `UNDEFINED` attribute of the `MeasurementType` `class`.

You need to prefix attributes of the `Operator` `class` with `self`.

B.3 About `self`

In Python `self` refers to the object of the `class` the method is called upon. It is like a placeholder that allows access to attributes and methods of the object.

This is a characteristic of how Python works, and you need to build `self` into `Operator` `class` to comply with the Python syntax requirements.

In the context of the Echoview Python source file, there are three instances when you need to specify `self`

- setting up either `eval` or `result_type` in the `Operator` `class`,
- utilizing the `window_size` attribute of `Operator` `class`, or
- programming and using any attribute in the `Operator` `class`.

Note that `self` is not a Python keyword. You may designate another name for this parameter, however, the canonical protocol is to use `self`.

C The `echoview` Python package

It is necessary to `import` the `echoview` Python package in your Python source file for the Code operator to work. The `echoview` package comes installed with Echoview.

Refer to the default Echoview Python source file (appendix A.1). The statement

```
import echoview as ev
```

imports the `echoview` Python package and gives the `Operator class` access to the following classes.

- `OperatorBase`
- `OperandInput`
- `Measurement`
- `MeasurementType`

We will briefly describe these classes here. However, refer to the Using the Code operator online Help page for more details about them.

C.1 `OperatorBase`

This is the parent for the `Operator class`, and `Operator` inherits the `eval` and `result_type` methods from `OperatorBase`.

C.1.1 `eval(self, inputs: List[ev.OperandInput])`

This method reads in pings from all the input operands, executes your Python commands on the pings and communicates the resultant ping samples to Echoview. It is necessary to declare the `eval` (abstract) method in your Python source file.

This method requires the `return` statement. The value is a NumPy array of the same shape as `inputs`.

The `eval` method has two parameters:

1. `self`

   When programming a method in Python, the first parameter fulfills a unique role. It governs which object you want the method to act on (appendix B.3).

2. `inputs: List[ev.OperandInput]`[28]

   This is a `list` of all the operands you specified on the Operands page of the Variable Properties dialog box (Figure 5).

   `inputs[0]` corresponds to Operand 1. `inputs`$[N-1]$ corresponds to Operand N. The operands must have the same ping geometry.

[28]This is defined according to Python's function annotation feature which serves to assist you by indicating the parameter's type. The text before the colon is the parameter's name, and the text after the colon is the parameter's type—in this case a Python `list`. While coding, reference this parameter with its name.

Each operand in `inputs` is an object of the `OperandInput` `class` (appendix C.2).

C.1.2 `result_type(self, input_types: List[ev.MeasurementType])`

This method sets the data type of the Code operator in Echoview.

Even if you do not declare it, Echoview automatically calls `result_type` (via `OperatorBase`) and sets the data type of the Code operator to the same as Operand 1 (i.e., `input_types[0]`).

You may change the data type to that of another Operand or to any other data type supported by the Code operator with the attributes of the `MeasurementType` `class` (appendix C.4).

This method requires the `return` statement. The value is a `MeasurementType` object.

The `result_type` method has two parameters:

1. `self`

   Refer to the equivalent description in `eval` in appendix C.1.1.

2. `input_types: List[ev.MeasurementType]`[28]

   This is a `list` of all the data types of the operands you specified on the Operands page of the Variable Properties dialog box (Figure 5).

   `input_types[0]` corresponds to the data type of Operand 1. `input_types[`$N-1$`]` corresponds to the data type of Operand N.

   Each data type in `input_types` is an object of the `MeasurementType` `class` (appendix C.4).

C.2 `OperandInput`

Each operand on the Operand page of the Variable Properties dialog box in Echoview is an object of this `class`.

The `inputs` parameter in `eval` (appendix C.1.1) is a `list` of objects from this `class`.

Refer to the Using the Code operator online Help page for more details on this `class`.

C.3 `Measurement`

Each object of this `class` represents a ping (in the window of measurements (section 9)).

Refer to the Using the Code operator online Help page for more details on this `class`.

C.4 `MeasurementType`

Echoview organizes ping data into categories based on their types. For example, Boolean, angular, Sv or TS. This mechanism is implemented in the Code operator via the

`MeasurementType` `class`.

The `input_types` parameter in `result_type` (appendix C.1.2) is a `list` of objects from this `class`.

Refer to the Using the Code operator online Help page for more details on this `class`.

D Examples using `numpy.where`

The `numpy.where` method is a useful and versatile tool that combines the functionality of an `if` statement and a `for` loop to iterate over NumPy arrays. Here is an overview to demonstrate how it works. For more details, refer to the documentation on NumPy's website (section 14.2).

```
numpy.where(condition, x, y)
```

Parameters:

condition:	A Python conditional statement. If it is satisfied, yield x, otherwise, yield y.
x, y:	The values to yield from the conditional statement.

The usual convention is to use the following statement to import NumPy

```
import numpy as np
```

Then, as an example, if we create a NumPy array

```
a = np.array([0.,1.,2.,3.,4.,5.,6.,7.,8.,9.])
```

with `np.where` we can filter elements in the array with value greater than 5.

```
mask = np.where(a > 5, a, np.nan)
```

`mask` would be `[nan nan nan nan nan nan 6. 7. 8. 9.]`, where `nan` stands for "not a number". An equivalent implementation using a `for` loop with an `if` statement would look like

```
mask = a
for i in range(len(a)):
    if i > 5:
        mask[i] = a[i]
    else:
        mask[i] = np.nan
```

However, `np.where` works on multi-dimensional arrays as well.

If we instead create

```
a = np.array([[0.,1.,2.,3.,4.,5.,6.,7.,8.,9.],
              [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.],
              [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.],
              [0.,1.,2.,3.,4.,5.,6.,7.,8.,9.]])
```

then `np.where(a > 5, a, np.nan)` would output

```
[[nan nan nan nan nan nan 6. 7. 8. 9.]
 [nan nan nan nan nan nan 6. 7. 8. 9.]
 [nan nan nan nan nan nan 6. 7. 8. 9.]
 [nan nan nan nan nan nan 6. 7. 8. 9.]]
```

and `np.where(a != 5, True, False)` would output

```
[[ True True True True True False True True True True]
 [ True True True True True False True True True True]
 [ True True True True True False True True True True]
 [ True True True True True False True True True True]]
```