An Informal Guide to DASVader.jl and the Julia Language

Hello there!

Welcome to DASVader.jl!

As I worked with Julia, I started writing a bunch of code to handle the Distributed Acoustic Sensing (DAS) data from the FIMOPTIC project at IPGP. After a few months of coding and testing, I realized I had enough tools to perform the basic steps one would typically follow in SAC when working with seismic data: preprocessing, filtering, plotting, and saving. So, I decided to bundle everything into this package so that everyone can benefit from it!

About This Guide

This PDF is an informal and informative guide. It's not part of the formal documentation that will eventually accompany DASVader.jl. Instead, it's meant to help anyone get started with the package quickly.

I also place special emphasis on using Julia, as I know it's not the most well-known computational language out there (yet!).

What to Expect

We'll start by installing Julia on your computer and setting up DASVader.jl. After that, I'll give you a quick tour of the Julia environment (don't worry—I'll also share links to better tutorials than my own!). Finally, we'll go through the examples included with the package so you can follow along and start working with your own data.

Let's dive in!

1. Installing Julia Language.

Julia is a relatively new computational language designed specifically for science! It was created by scientists for scientists. When it first launched, it carried the slogan: "Julia: come for the syntax, stay for the speed." The idea behind Julia is to combine the ease of use and learnability of "modern" computational languages like Python and MATLAB with the speed, power, and flexibility of "classic" languages like C, C++, and Fortran. Julia excels at parallel computing, GPU programming, and machine learning, making it an incredibly versatile tool.

If this sparks your curiosity, now might be the perfect time to give it a try!

To get started, let's install Julia on your computer. Visit the official Julia downloads page and select the appropriate installer for your operating system.

https://julialang.org/downloads/#current_stable_release

I recommend you install version 11.1.

If you have **Linux** you should be able to use this command to get it immediately:

curl -fsSL https://install.julialang.org | sh

For **macOS**, you'll need to choose the installer that matches your machine's architecture:

- If your Mac uses an Intel processor, select macOS x86 (Intel or Rosetta).
- If your Mac has an M1 or M2 chip, choose macOS (Apple Silicon).

Once downloaded, open the .dmg file and drag the Julia app into your Applications folder. To use Julia from your terminal, you'll need to add it to your system's PATH. The exact steps depend on your macOS setup, but the path you're looking for is likely:

/Applications/Julia-1.11.app/Contents/Resources/julia/bin

Here's how to add it to your PATH:

Open your terminal and check which shell you're using by running:

echo $SHELL

If you're using zsh (the default in recent macOS versions), you'll need to edit the .zshrc file.
For older systems using bash, edit the .bash_profile file instead. Open the configuration file with a text editor, for example:

```
nano ~/.zshrc
```

Add the following line at the end of the file:

```
export PATH="/Applications/Julia-1.x.app/Contents/Resources/julia/bin:$PATH"
```

Save and close the file (in nano, press CTRL+O, then CTRL+X).

Refresh your shell:

```
source ~/.zshrc
```

Now, you should be able to launch Julia by simply typing julia in your terminal!

For **Windows**, there are installers available on the Julia Downloads page. Unfortunately, I haven't personally tried them, so I can't provide detailed steps. But don't worry! The installation process should be straightforward—just follow the instructions on the page.

A Note About JuliaUp

If you think Julia might be the right tool for you, consider checking out JuliaUp. It's the official way to install and manage Julia versions, making updates and version switching easier. You can learn more about it here.

https://github.com/JuliaLang/juliaup

However, fair warning: JuliaUp is a bit more hands-on and technical. That's why I've stuck with the "standard installation" instructions for this guide.

2. Getting started with Julia Language.

Once you have installed julia and added it to your path you should be able to launch julia from your terminal. Open a terminal and write:

```
julia
```

and press return or enter:

You should see something like this

```
[andurin@MacBookAir ~ ] julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.11.1 (2024-10-16)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

You can do some basic math and create some simple arrays (hit return after each line and after the # the explanation of each line):

- A=2 # Variable A equals 2 (this is an Int64)
- B=2.0 # Variable B equals 4.0 (this is an Float64)
- C=A+B # Add A and B to get the answer (it is Float64)
- V1=[2 3 4 5 6] # Create a 1 row matrix (if you separate the number with ";" it is a vector)
- V2=1:5 # Create a range from 1 to 5
- M=V1.*V2 # vector multiplication. Notice we use a dot .* without the dot, give it a try without
- M2=pi*M # multiply M by pu
- CM=cosd.(M2) # get the cosine for each value in M2

Here a view of the same inside Julia

```
julia> A=2
2

julia> B=2.0
2.0

julia> C=A+B
4.0

julia> V=[2 4 5 6 7 8]
1×6 Matrix{Int64}:
 2  4  5  6  7  8

julia> W=1:5
1:5

julia> M=V.*W
5×6 Matrix{Int64}:
  2   4   5   6   7   8
  4   8  10  12  14  16
  6  12  15  18  21  24
  8  16  20  24  28  32
 10  20  25  30  35  40

julia> M2=pi*M
5×6 Matrix{Float64}:
  6.28319  12.5664  15.708   18.8496   21.9911   25.1327
 12.5664   25.1327  31.4159  37.6991   43.9823   50.2655
 18.8496   37.6991  47.1239  56.5487   65.9734   75.3982
 25.1327   50.2655  62.8319  75.3982   87.9646  100.531
 31.4159   62.8319  78.5398  94.2478  109.956   125.664

julia> CM=cosd.(M2)
5×6 Matrix{Float64}:
 0.993993  0.976045  0.962654   0.94637     0.927242    0.905326
 0.976045  0.905326  0.853406   0.791233    0.719554    0.639231
 0.94637   0.791233  0.680415   0.551228    0.40716     0.252099
 0.905326  0.639231  0.456603   0.252099    0.0355171  -0.182767
 0.853406  0.456603  0.198687  -0.0740698  -0.341294   -0.583027
```

I know this is very very basic, but you can see that working in julia is not very different from Python, R or Matlab (in which you might have some experience). If you want to dive more check out this links:

https://www.datacamp.com/tutorial/julia-programming-tutorial-for-beginners

For a more formal version that dives deeper into things try this PDF:

https://www.sas.upenn.edu/~jesusfv/Chapter_HPC_8_Julia.pdf

For a formal introduction to the language I have to recommend the tutorial from JuliaAcademy. You can install the full GitHub project in your computer:

https://github.com/JuliaAcademy/JuliaTutorials

Or you can go inside the introductory course and see the Jupyter notebooks (.ipynb one by one).

https://github.com/JuliaAcademy/JuliaTutorials/tree/main/introductory-tutorials/intro-to-julia

Installing packages

In order to do more you will need to install some packages. For this you will need to find in your keyboard the symbol "]". When you press this key into julia the repl (the Julia terminal changes) goes into package mode:

```
(@v1.11) pkg>
```

Here you can install any package. try installing a plotting library:

```
add Plots
```

You should see something like this but longer:

```
(@v1.11) pkg> add Plots
   Resolving package versions...
   Installed Xorg_xcb_util_image_jll ─────── v0.4.0+1
   Installed x265_jll ────────────────────── v3.5.0+0
   Installed GR_jll ──────────────────────── v0.73.8+0
   Installed Xorg_xcb_util_wm_jll ────────── v0.4.1+1
   Installed Measures ────────────────────── v0.3.2
   Installed ConcurrentUtilities ─────────── v2.4.2
   Installed LoggingExtras ───────────────── v1.1.0
   Installed FFMPEG ──────────────────────── v0.4.2
   Installed RecipesPipeline ─────────────── v0.6.12
   Installed OpenSSL ─────────────────────── v1.4.3
   Installed Xorg_libSM_jll ──────────────── v1.2.4+0
   Installed Xorg_xcb_util_keysyms_jll ───── v0.4.0+1
   Installed Xorg_xcb_util_jll ───────────── v0.4.0+1
   Installed HTTP ────────────────────────── v1.10.12
   Installed SimpleBufferStream ──────────── v1.2.0
```

Once its finished you should have Plots.jl in your system and you can do your first plot! just try

```
heatmap(CM)
```

A little window should open and show you this:

A word of warning Plots.jl is very powerful, maybe you can dive into this website if you are interested

https://docs.juliaplots.org/stable/

Using Shell mode!

Yes, Julia can give use access you a standard linux shell, you can change path, make dir, ls etc. For this just press ";" and the terminal will change:

```
shell> pwd
/Users/andurin

shell> mkdir Hello

shell> ls
09_05                          Untitled1.ipynb
DASVader_V0                    Untitled2.ipynb
DASVader_V0.1.zip              Voids_Model_equi_SlowLen_2.txt
```

If ever in doubt press "CONTROL and C" to kill a line and "CONTROL and D" to kill Julia

3. Install DASVader

Now you are ready to install DASVader.jl!

Now, before anything, you can visit the project's GITHUB page here:

https://github.com/marianoarnaiz/DASvader.jl

There you can copy the following line if it does not work from this PDF. Go to Installation and copy the like in point 3. Just click on the 2 little squares to copy the line and paste is in julia.

Installation

At present, **DASVader** is unregistered, and both it and its dependencies must be installed manually. Follow the steps below to get started:

1. Launch Julia from a terminal or your favorite IDE.
2. Enter **Pkg mode** by pressing `]` in the Julia REPL.
3. Run the following command to add DASVader and its required dependencies:

```
(v1.11) pkg> add https://github.com/marianoarnaiz/DASvader.jl https://github.com/anowacki/
```

4. Once the installation is complete, you can start using DASVader by loading it into your Julia session:

```
julia> using DASVader
```

If not, open the package manager (using "]") and paste the next instruction.

add https://github.com/marianoarnaiz/DASvader.jl https://github.com/anowacki/Geodesics.jl https://github.com/anowacki/Seis.jl

Notice that it is much more complicated that installing Plots.jl. The reason is that DASVader as well as Geodetics.jl and Seis.jl are not registered packages yet.

Once installed write:

```
using DASVader
```

To check if it is working we will ask julia for some help! for this in julia type "?" and the repl will tun into the help center. You can write the name on a function a get information (Docstrings). Try bpdas (band pass das, one of the functions we will be using. You should see:

```
help?> bpdas
search: bpdas bpdas! xbpdas hpdas lpdas ppdas xbpdas! absdas xhpdas xppdas lpdas! hpdas! xlpdas ppdas! fkdas

  bpdas: Band Pass dDAS data for each channel.

  Input:

    • dDAS: DAS data structure

    • f1: lower frequency of the filter. In Hz.

    • f2: upper frequency of the filter. In Hz.

    • poles: poles of the filter.

  Outputs:

    • bpDAS: band passed DAS data

  Notes:

    • No figure is created.

  Example: Save one channel and its time vector to a file.
  ========================================================

  julia> bpDAS = bpdas(dDAS; f1=50, f2=500, poles=4)

julia>
```

OK. This should get you started. Now you can go to the examples I have provided.

BEFORE YOU START TO GO THROUGH THE EXAMPLES PLEASE GET SOME DATA! I HAVE UPLOADED SOME FILES TO DROPBOX. Here are the links

A Noisy blast file:

https://www.dropbox.com/scl/fi/c6ui9cxcb1gxawm0qqkcp/SR_DS_2023-08-24_14-06-17_UTC_Noisy_Blast.h5?rlkey=e3yn1likn3mkrhesyx85pcpw8&st=6zh6w1i0&dl=0

A file with a micro event:

https://www.dropbox.com/scl/fi/xxrd8rlw8kwthmfgwamwx/SR_DS_2023-10-30_12-01-40_UTC_Microevent.h5?rlkey=3zjvn706s46grco4gzhzrqmu5&st=0di6y3x8&dl=0

A file with a big blast:

https://www.dropbox.com/scl/fi/abcb1zphevctfkzetn7ql/SR_DS_2024-10-22_14-08-02_UTC_Big_Blast.h5?rlkey=mi9mvj3ynzptgxj3e1khkre8l&st=46eo4s9h&dl=0

A file with something that might be an event:

https://www.dropbox.com/scl/fi/n5czzuez7lq2yt3j2s5k5/SR_DS_2024-10-22_21-27-02_UTC_Hidden_Event.h5?rlkey=fsb9tq7wuxgbek5av3uc8rpzd&st=ncrqbwhx&dl=0

I suggest you get a few of these, put them in a directory and open each example in a text editor. You can begin just by copying and pasting lines from each example file into Julia. If you use VSCODE, VIM or EMACS you should really check this links out:

https://www.julia-vscode.org/

https://github.com/JuliaEditorSupport/julia-vim

https://github.com/JuliaEditorSupport/julia-emacs

4. Example 1.

Notice that there are more comments in the Example1.jl file!

For Example1.jl you will need to have "SR_DS_2023-10-30_12-01-40_UTC_Microevent.h5" in your computer. This is a file that contains a micro event recorded in the Garpenberg mine. First start julia as you have before and ask it to use DASVader by using this line:

```
using DASVader
```

Now, we will read the file into a variable we will call dDAS (for DAS data). The command is rdas (read DAS data) and currently only supports data from the FEBUS A1 DAS. Remember to put the name in "" so Julia understands it is a String. The name could also have the absolute path, if you are working in a different directory.

```
dDAS = rdas("SR_DS_2023-10-30_12-01-40_UTC_Microevent.h5")
```

Alternatively, you can save the name to another variable and call it inside rdas. For example:

```
filename="SR_DS_2023-10-30_12-01-40_UTC_Microevent.h5"

dDAS = rdas(filename)
```

Now, lets plot the data. We will use viewdas. It requires a few inputs: (a) the variable with the data, (b) a colormap, (c) a limit for the color scale. Try this command and you will see the picture bellow.

```
fig = viewdas(dDAS; cm=:RdBu_9, climit=10000)
```

Depending on your screen, operating system and GPU this picture will be slightly different because viewdas (as well as all the plotting tools included will try to make plots that are adequate for your machine to handle).

The function viewdas will produce a colormap (or heatmap) of the data. The X axis corresponds to the time and the Y axis to the distance along the fiber (offset). This is a convention that would be used all along the package. You can try different colormaps (cm=:viridis or cm=:turbo) and limits to (climit=1000) to make the figure to your liking.

The figure will open in a new interactive window, meaning you can zoom-scroll, window-zoom. Try this:

Zoom in and out: Use your mouse wheel/track pad like scrolling on a document up and down to zoom in an out. The mouse position on the figure marks the center of the zooming.

Window zoom: Right click the figure and hold, you will be able to draw a rectangle and zoom in into it.

Restricted window zoom: If you hold the X or Y key on your keyboard you will be able to zoom in into the entire data while keeping the whole axis steady. Hold the key, right click and drag the mouse up/down/right or left.

Reset your view: At any time you can reset your view by holding CONTROL and right click in the figure.

In the figure we have made there is a vertical line. This is actually a micro event. Try to zoom to it an get a nice picture of it. Like this one:

If you really like your figure you can save it to PDF using the savefig function. This is a very useful one, just give the name of the figure variable and the PDF mane you want.

```
savefig(fig, "MicroEvent.pdf")
```

That is a nice event right there! Now lets try to select a section of the event (cut, if you will). If yo zoom in you will be able to see what is the time and offset range where the event is recorded. Maybe one of the black squares (where the fiber goes in and out of a borehole) or the orange square (where the fiber is inside a trench) catches your eye for this. Here you will need slicedas, which requires the input variable to cut (dDAS in this case) and the time range (given as tmin and tmax) and the offset range (as xmin and xmax). Just look at the borders of your figure to get these values. For me, these values work to make the next figure. We will save the output to a new variable called cDAS (for cut DAS)

```
cDAS = slicedas(dDAS; tmin=4.65, tmax=4.85, xmin=1250, xmax=1400)
```

And you can plot and save a figure of cDAS as well:

```
fig2 = viewdas(cDAS; cm=:RdBu_9, climit=10000)

savefig(fig2, "MicroEvent_Zoom.pdf")
```

Notice that when plotting this time I have changed the output. We have fig and fig2 in memory. This takes a lot of space in memory, but you can recall either by typing the name and pressing return. Give it a go!.

Now, this is not the best way to see the waveforms of the data. For this one needs a record section! Fortunately you can plot one by using recordsection. This function is quite simple it requires two inputs: (a) the data to plot, (b) a scaling factor (think that it is a fraction of the offset). I will try a scale=5.0 this time and I will save the section (rs) to a PDF as well. One thing to note is that savefig will save the original configuration of the plot. Zoom in, for example, has no effect on the written file.

```
rs= recordsection(cDAS, scale=5.0)

savefig(rs, "Record_Section.pdf")
```

You can inspect this figure much like you did the ones before.

Now, maybe is a good time to study one of this waveforms with a bit more detail. Maybe the one with the earliest arrival. For this you will need to isolate the channel. There are a few ways to do this, but lets use viewchannel first. This function will allow you to select one channel by its distance in the fiber, plot it on its own window and save the data to its own variable. Additionally you will get an quick-and-dirty frequency spectrum and spectogram!. Additionally the figure will also be saved to a variable. You will need: (a) the data from where to extract the channel and (b) the distance of the channel in the fiber. Notice that you only need to give an approximated distance. Zoom in into the record section to find the distance to the channel you prefer, for me it is 1326.5.

```
channel1, spectrum1, spectogram1, f_Channel1 = viewchannel(dDAS;
x=1326.5)
```

Now, this function is in active development to improve the spectrum that is shown and the spectogram too.

You can independently zoom in to any panel of the figure and save it if you wish.

The function have created 3 independent variables for you: channel1 (a vector with the channel), spectrum1 (the frequency spectrum of the channel) and spectogram1(the spectogram of the channel). The index of the channel is also printed to the repl (julia terminal) and to the title of the figure!. This means that we can use this information inside julia in many ways! Remember that DASVader is not an isolated environment and that you could use it to achieve many things inside Julia. Let's spend some time doing so.

Lets start by making a plot of the channel itself. For the sake of simplicity we will a function that is part of DASVader. The function ilines makes line graphics. We have the amplitude of our channel (or trace) in memory, but we need the time! The time is stored in the dDAS variable (there is also the cut time in cDAS). You can check it out by calling

```
dDAS.time
```

Now we can make a quick plot like this:

```
ilines(dDAS.time, channel1)
```

That should make figure you can inspect as before. And maybe we would like to overlay the data from the next channel and the previous channel to the same plot, to see the difference. We can do this with the index of the channel. You can check that

```
dDAS.data
```

hold the entire DAS data matrix. Each row is a time sample, while each column is a channel. By calling

```
dDAS.data[:,830]
```

you will get the same data saved in the variable channel1. Therefore the previous and next channel are in index 829 and 831. To add them to the plot we will use ilines!. Notice that there is an extra "!". This exclamation mark is not for making things more exiting, it is a julia convention. Functions with "!" at the end are called in-place functions/operations, meaning the usually overwrite something from the memory. In this case we will draw over our previous plot like this:

```
ilines!(dDAS.time, dDAS.data[:,829])

ilines!(dDAS.time, dDAS.data[:,831])
```

With a bit of zoom it, you should be able to see 3 traces in your plot:

As a very last step maybe it is useful to save some of our work. We have a range of channels with very useful data that we can write to individual SAC files. This can be done with das2sac by giving: (a) the data and (b) a range of channels (only one channel is accepted as well).

```
das2sac(dDAS; x=1310:1340)
```

This will create in your path several .SAC files with the data of each Channel in the current working path.

The final step could be to save the event file to a new format. This way if you need to work on this event, and in the channel range selected you do not need to load the entire raw file again. For this, DASVader uses its own format. This format is lighter than the raw Febus file and holds all the information you need. Try this line:

```
writejdl(cDAS, "MicroEvent")
```

If you get an error, maybe you need to load this library and install JLD2 by hand (this happens often and I am tryin to work around it):

```
import Pkg; Pkg.add("CodecZlib")
using JLD2
```

To read this file to memory just do:

```
@load "MicroEvent.jld2"
```

OK. This will be the end of this example. You can try to do something similar with the rest of the files. I think that is a lot of ground we have covered regarding handling data. The next example covers more on how to process information.