FLASH quick start guide for experimentalists

Gabe Xu, University of Alabama in Huntsville
gabe.xu@uah.edu
(last update 8/18/25)

The follow guide is how I got FLASH to run on my laptop, using both Docker and a Linux distribution in Windows. It mostly follows the methods given in the FLASH users guide, with some notes and edits that are either particular to my system, or because that part of the user's guide is out of date. I also try and explain what's going on and the syntax, partly as a note to myself for later to remember what's what. Note this is just my experience, it may not work for everyone. But hopefully it's useful in some minor way.

The FLASH users guide gives two ways to install FLASH: the long complicated way that requires installing a half dozen libraries from different sources onto a Linux installation, or the short way using the Docker program that's intended for a single local user on their own computer. The latter is the easiest way to get FLASH running, but it will always run simulations slower than doing a Linux installation. For comparison, the basic 2D Sedov blast wave simulation took 44 seconds on a single core with Docker, and 17 second on a single core in the Window's Linux shell. If you go to a fully Linux boot partition, it'll probably run even faster since it's not fighting Windows for processing power.

First thing to do for either method: register to download FLASH

FLASH code is managed by the University of Rochester. You can request access to the code at the following page: https://flash.rochester.edu/site/flashcode.html. It takes about a day or two for the request to be processed, after which you'll get an email with username and password to download the program.

THE EASIER DOCKER METHOD

1. Getting the main programs

For this guide, I used six main programs:

1) Docker Desktop to create the Linux virtual environment (called a container) to run FLASH
2) Windows Subsystem for Linux (WSL) in order to run the Linux container in Docker
3) Visual Studio Code (VS Code) to give commands to the container and edit FLASH files
4) VisIt to visualize the FLASH outputs.
5) The FLASH code itself

Docker

Docker Desktop is a Windows program that allows you to create temporary virtual machines, or a small computer environment running in your computer. In Docker language, these are called containers, and containers are created from an image. The image is the permanent thing, while the container that is created from the image is temporary and usually deleted after you're done running code in it. In other words, the image is the original, and the container is a copied instance. The reason Docker is useful to us is because there is a FLASH Docker image that has a small Ubuntu OS and all the Linux libraries, dependencies, and components all built and linked (which what you'd need to do if you went the long way). Essentially, by running that FLASH Docker image, you create a virtual computer that is fully set to run a FLASH simulation, cool huh. However, the Docker image is at least 5 years old, and so doesn't have the most up to date programs and libraries. It's also slower to run because it's running a virtual computer within a program, which is then running in Windows.

You can follow the instructions to install docker from https://docs.docker.com/get-started/. Docker Desktop can be installed for windows either by downloading the install file (~600 MB), or installing it from the Windows Store. I tried both, and like using the install file because the windows store option doesn't give you indication if it's working or the progress, which is kind of annoying.

WSL

WSL is a built-in function in Windows, you just need to turn it on. Open a command line interface (CLI), like Windows Powershell or just search "CMD" in the start menu. In the terminal (black box with text input) that pops up, type

```
wsl --install
```

(Yes, two dashes) This takes care of a number of things to get WSL setup. For one, it turns on the two features in Windows needed, Virtual Machine and WSL (you could turn these one manually by going to "Turn Windows features on and off." That command also installed the Linux kernel, and the latest Ubuntu Linux OS (called a distribution). During the installation, it will ask you for a username and password. The username will create a directory within the Ubuntu OS, and the password is needed when you want to change things about the installation, so remember the password.

Once Ubuntu is installed, you'll want to do some basic updates by using the following two commands

```
sudo apt update
sudo apt upgrade
```

The first will update the repository of known libraries and programs that Ubuntu can install automatically, when you tell it to. The second upgrades all the libraries to their latest version.

VS Code
VS Code is an open source code editor made by Microsoft for Windows (it kind of look like Matlab). It can work with all sorts of languages by using extensions, which are downloaded from within the program. You can get VS Code for Windows from https://code.visualstudio.com/.

VisIT
VisIT is an open source visualization program made by LLNL which has built in recognition of FLASH output. You can get VisIT for windows from https://visit-dav.github.io/visit-website/, and then install it and normal. The users guide first says to get IDL, but that is a commercial program.

On the VisIT page, go to Downloads, the Releases. Scroll down a bit and there's a table with the latest release for different OS's. I used Windows 10, and the "use" link to download the install file. Then install the program. During the installation, it'll ask you if you want to pick a default database reader plugin. I selected FLASH since that's what I'd be using it for, and it'll save a click when looking at the results. But if you don't want to pick on, then select None.

Use the None for network configuration, and you can associate FLASH extensions with VisIT, but I'm not sure if it helps a whole lot.

FLASH code
FLASH code is managed by the University of Rochester. You can request access to the code at the following page: https://flash.rochester.edu/site/flashcode.html. It takes about a day or two for the request to be processed, after which you'll get an email with username and password to download the program. It'll be downloaded as a .tar.gz compressed file that you have to extract (like a zip file). Windows should be able to do it natively. Make a folder where you'll be doing the simulation runs and extract the files there. By default, the extraction will create a folder called FLASH4.8 with all the files inside, or whatever the version is currently. So you could just extract it onto your desktop.

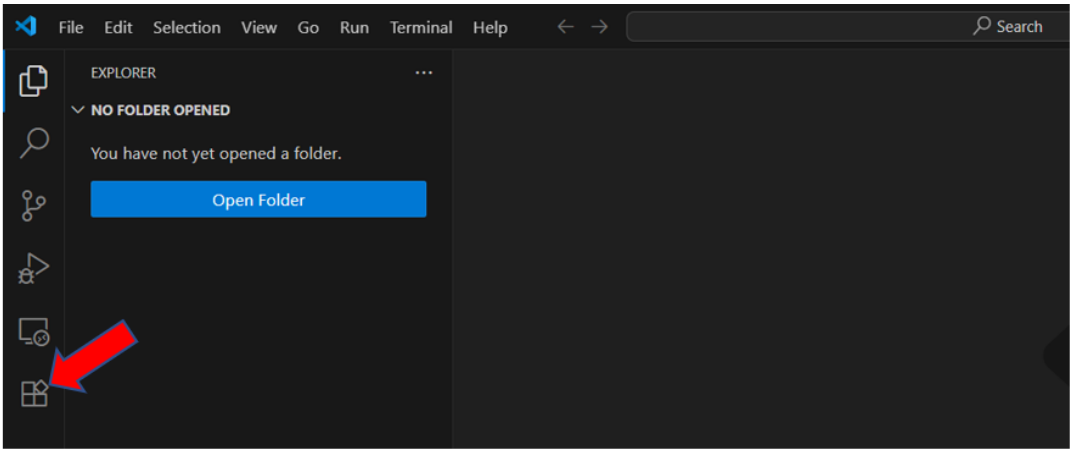
2. Setting up Docker

When you first run Docker, it will ask you to login or register. Docker Desktop is free to use for individuals, you just need to make an account (I did personal, not work. Not sure if it matters). Once Docker is running, you can do the basic tutorial with the welcome-to-docker image and container to see how it work. We'll come back to the FLASH container later.

Keep Docker running when you do the next step.

3. Setting up VS Code

Open VS Code and click on the Extensions icon on the left side (looks like 4 blocks).

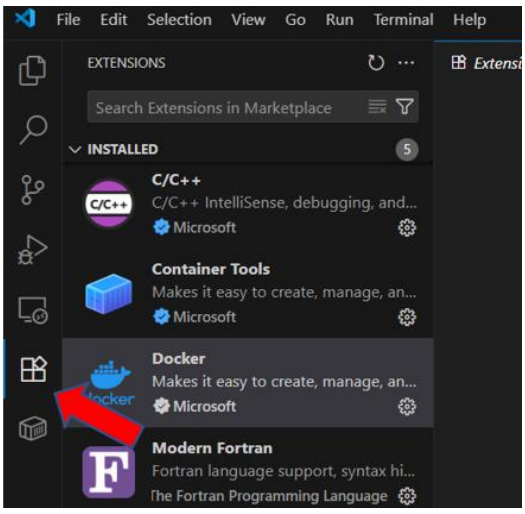


In the search bar, look for "Modern Fortran" and install that. When I did it, there was a warning that said "this is not signed by the extension." So I had to click the gear icon and force it to install and trust it.

- This extensive adds some conveniences like color coding and auto indents when you write fortran in the VS Code editor.
- It looks like Modern Fortran also installs a C/C++ extension. That's fine, we won't need it so best to leave it alone.

You'll also want to install the two Docker extensions "Docker", and "Dev Containers" (both by Microsoft). It'll put a containers icon on the left side of Docker (looks like a crate), under the Extension icon.

- This will also add the "Container Tools" extension.
- When I did this, the Dev Containers extension wouldn't load and kept giving an error that said can't find it. It went away when I reset the computer and then kept Docker running when I opened VS Code.



If you have Docker Desktop running, that tab will show you the existing containers. You can start it by right click and start. But, apparently the point of Docker is to create a new container each time from an image, kind of like resetting the computer each time. That's usually ok since all the files that are acted upon are in the FLASH folder on the computer, and it's shared with each new container instance.

4. Getting the FLASH image and container for Docker

Now that you have Docker and VS Code running, lets get the Docker container going for FLASH. The flash users guide says there's a Docker image called "icksa/flash4-deps", but that one is out of date and Docker won't download it. Instead, there is a newer one call "flashcenter/flash4-deps" that does work. In VS Code, you can open a terminal prompt from either the toolbar, or by hitting Ctrl + ` (that's the button next to the 1 key). This is similar to running "CMD" from the start menu or windows powershell.

In the terminal, type

```
docker pull flashcenter/flash4-deps
```

VS Code will make the word docker yellow since it recognizes as referring to that specific program/function (the whole color coding thing that's useful). This command will have Docker download the image "flash4-deps" from the Docker repository. Once that's done, if you go to the Docker icon in VS Code, it should show the flash4-deps image.

Now you can create a container from the image and link it to the folder where you extracted the FLASH code. I put my FLASH folder on the desktop, so my path to the folder is

```
C:\Users\gabe\Desktop\FLASH4.8
```

In VS Code terminal, then type (all one line, and replace the "gabe" part with your directory)

```
docker run -v C:/Users/gabe/Desktop/FLASH4.8:/mnt/flash -it flashcenter/flash4-deps
```

The first and last parts tell docker to run the flash4-deps image and create a container. The "-it" part tells docker to make the container interactive via an external terminal (like VS Code). The middle part tells the container to create a folder inside the container called "flash" under the /mnt directory, and link it to your FLASH4.8 folder on the desktop where the FLASH code sits. This is called a shared volume, which is the "-v" tag in the command.

- As a random fact that I learned and feel like sharing, the /mnt directory is a default one in Linux and stands for "mount". It's used as a temporary directory for any temporary file systems that are attached to the system, like a usb drive, or an external folder like we're doing.

If the command works, you'll see a new container pop up in the containers list in VS Code and in Docker. It'll have some random name like wizardly_lehmann. It'll also have a container ID that's some long random hex string. Your terminal prompt will also change to something like "root@b20a2e96902a:/#", where that "b20a2e…" part is part of the container ID. So it's saying that you're now talking to that container.

If you expand the container in the list on the left (click on the > sign), it'll expand the container. Then you can expand the files, and then expand mnt and you'll find the files in your FLASH4.8 folder on the desktop (or where ever you put it). This necessary because the Docker flash4-deps containers doesn't actually have the FLASH code, just all the Linux libraries and gfortran. At the time of writing, the flash4-deps image is using an old version of Ubuntu and gfortran. By creating the shared volume that links to the actual code on your desktop, you give the container environment access to the code.

Now you're ready to run FLASH. Huzzah!

To close a container, you can either type "exit" in the VS Code terminal, or go to the Docker Desktop program and hit the blue square stop button. If you then enter the "docker run" command again, it will create a brand new container with a new ID that links to the same FLASH code on your desktop. You can delete the old one by right clicking the container in the list in VS Code and select remove, or in Docker Desktop click the red trashcan.

5. Run the basics Sedov 2D blast wave simulation (and code fixes)

Now you can follow the FLASH user guide for running a simulation, but I'll also explain it here, along with how to fix a couple of errors I ran into.

Start up Docker, VS Code, and run a flash4-deps container that's linked to the FLASH4.8 folder on your computer. Then you'll need to get to the FLASH4.8 folder, which should be linked to the /mnt/flash folder inside the container. Enter the following in the terminal

```
cd /mnt/flash
```

That'll open the flash directory, then if you type "`dir`" and hit enter, it will show you the files and folders in that flash directory. One of them should be "setup". The setup file is how you start a FLASH simulation (there's 3 steps to prepare the program). To set up the Sedov 2D blast wave, type

```
./setup Sedov -auto
```

The ./ tells Linux to run a script or program in the current directory. The –auto part tells it to automatically set the units. You'll see a number of lines in the terminal as the code sets up the simulation. The "Sedov" part refers to the Sedov folder in your FLASH4.8 directory that's located in "FLASH4.8\source\Simulation\SimulationMain". If you look in SimulationMain, you'll find all the preconfigured simulations provided with FLASH. Not all of them work out of the box in my experience, but Sedov does.

If the setup work, then it'll say "SUCCESS" in the terminal and a new folder called "object" will appear in the FLASH4.8 directory. That's where the details of the current simulation that was just set up (or last set up) resides. Now you'll need to go into that directory and compile the FLASH code. So type the follow two commands in the terminal

```
cd object
```

```
make
```

The first command goes into the object folder. The second one tells Linux to compile the simulation and create an executable file. Make is a build-in function in Linux, and it depends on a Makefile, which was created in the object folder by the setup command. Makefile is a text file that gives instructions on how to compile the executable to run the simulation. Once you run the "make" command, a number of things will run in the terminal. You can actually run make with multiple processors using "`make -jX`", where X is the number of cores to use, e.g. `make -j3`. The first time you do this, it may give you and error that says

```
make: /usr/local/mpich2//bin/mpif90: Command not found
make: *** [Makefile:132: Burn_interface.o] Error 127
```

What that means is Linux tried to use the mpif90 program located in the "/usr/local/mpich2/bin/" directory in the container, but it wasn't there. The mpif90 function is in the flash4-deps container, it's just in a different place, so we have to change it. To find where it is, type

```
which mpif90
```

The "which" command will tell Linux to find that program, which should give back "/usr/local/bin/mpif90". So it's located in a slightly different place. The solution is to change where the code is looking for the mpif90 file. That will be in a Makefile, which is the set of text files that tell the code what to do and where to look for stuff. So lets find it and change it.

Go to the folder "/FLASH4.8/sites/Prototype/Linux" either on your computer on in VS Code. If in VS Code, then your FLASH4.8 folder is under "/mnt/flash". Open the Makefile.h you find there using either VS Code's editor or notepad in windows. At the top of the Makefile is the library path definition, which I've screenshotted below. You'll need to change the first one for "MPI_PATH" from "/usr/local/mpich2/" to "/usr/local" (yes, also remove the trailing /, that's apparently another oopsie in the code).

- The Prototype folder appears to be for the local computer. In the sites directory, you'll see bunch of folders that refer to various universities and labs. I believe those are predefined instructions for people at those places to use a central computer cluster or supercomputer for their FLASH simulations. Since we don't have access to the other sites, I deleted the rest except for Prototype to make getting to it faster. You can always recover those folders by extracting the FLASH4.8.tar.gz again if needed.

```
mnt > flash > sites > Prototypes > Linux > C Makefile.h
  # FLASH makefile definitions for x86-64 Linux (GNU compilers)
  #------------------------------------------------------------------------------
  # Set the HDF5/MPI library paths -- these need to be updated for your system
  #------------------------------------------------------------------------------

  MPI_PATH    = /usr/local/mpich2/
  HDF4_PATH   =
  HDF5_PATH   = /usr/local/hdf5
  HYPRE_PATH  = /usr/local/hypre

  ZLIB_PATH   =

  PAPI_PATH   =
  PAPI_FLAGS  =

  NCMPI_PATH  = /usr/local/netcdf
  MPE_PATH    =
```

Now if you run make again when in the "object" folder, it'll do a bunch of Linux commands to compile the simulation, but then it'll fatal error out with the following message

```
f951: Fatal Error: Reading module 'iso_c_binding' at line 1 column 1:
Unexpected EOF
```

If you go back to the Makefile in the "Prototype" folder, and scroll to the bottom, you'll find the following section

```
#------------------------------------------------------------------------------
# Fake existence of iso_c_bindings module to prevent unnecessary recompilations.
#------------------------------------------------------------------------------
ifeq ($(FLASHBINARY),true)
iso_c_binding.mod :
   touch $@
endif
```

According to the FLASH USERs mailing list, this is apparently an old workaround for old fortran compliers, but it no longer necessary for modern ones. So delete that section and save the Makefile. Now, if you try running make again, it'll still error out with the same one. That's because the old object configuration files are still there. To fix that, use the following command

```
make clean
```

That will remove all the object files and clean up a bit. Now you run make again and it should work. This will take a few minutes for Linux to compile the simulation. Go get some coffee.

It'll say "SUCCESS" when it's finished and worked, and will give you the terminal command prompt again. The complier will create a "flash4" file in the "object" folder. That's the thing

you'll execute to actually run the simulation. You can do it one of two ways depending on how much processing power you want your computer to use for it. The simplest is to just type

```
./flash4
```

This will run the simulation using a single core on your computer processor. You can use multiple processors by using mpi (message passing interface, which allows the processors to send stuff back and forth and share the load). To do that, use the follow command

```
mpirun -np N ./flash4
```

Where *N* is the number of processors you want to use.

- The flash users guide says to type "mpirun -np N flashX" where the X would be 4. However, this seems to cause a problem and throws an error "HYDU_create_process (lib/utils/launch.c:73): execvp error on file flash4 (No such file or directory)". This means for whatever reason it can't find the flash4 file. Adding the "./" tells it to look in the current directory, which is the "object" folder where it should be.

When the simulation runs, you'll see tables of numbers scroll as it does calculations and stuff.

Now, if you want to see how much time the simulation takes, you can use the time command in Linux. Simply put "time" in front of the command, for example

```
time ./flash4
time mpirun -np 3 ./flash4
```

This will cause the command to run as normal, but after it's done, you'll also get how long the process took. On my laptop, running the 1, 2, and 3 cores took 44, 25, and 23 seconds.

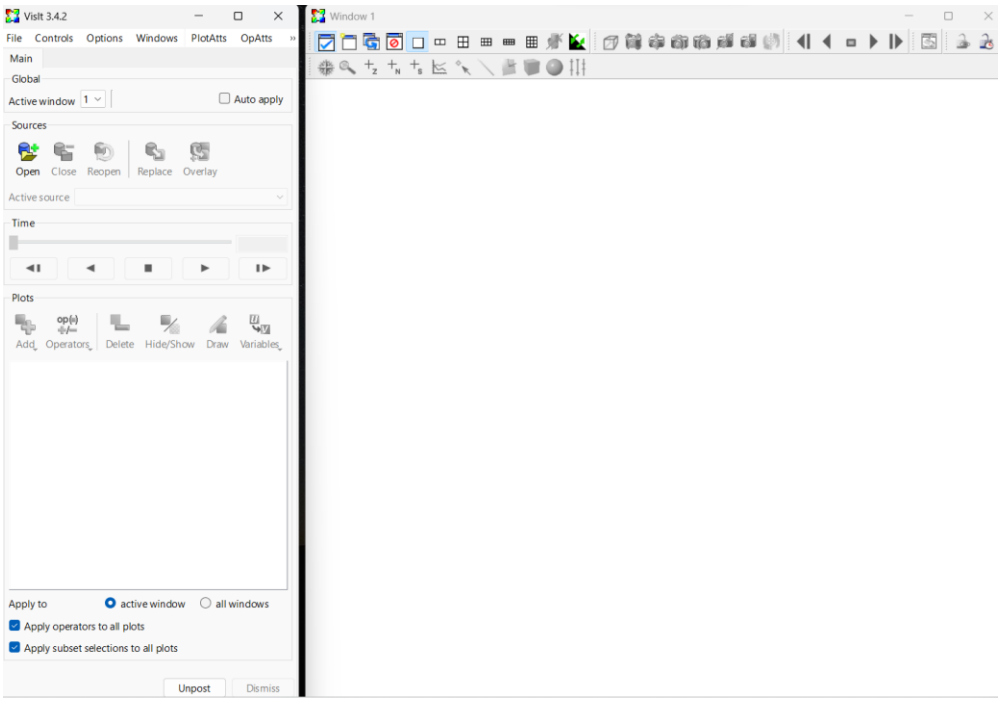
6. Plotting the results

If the simulation completes, it will create a new set of files in the "object" folder. It'll start with the name of the simulation, sedov in this case.

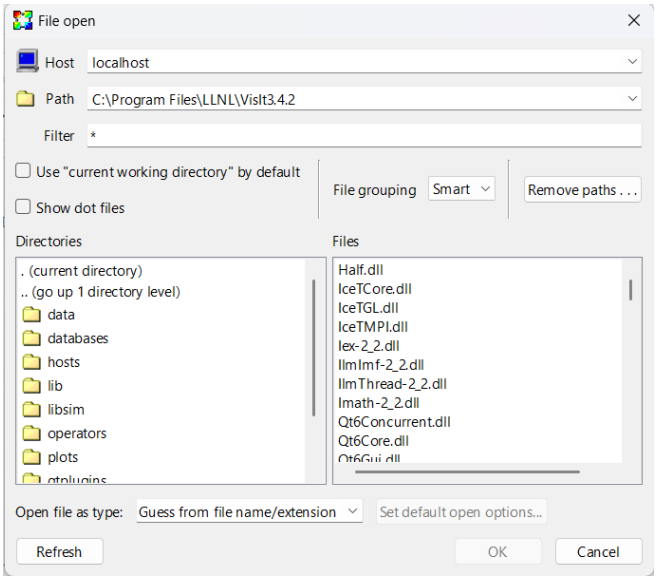
sedov.log
sedov.dat
sedov_hdf5_chk_000x (the x here is a number)
sedov_hdf5_plt_cnt_000x
sedov_forced_hdf5_plt_cnt_000x (this one doesn't show up for all simulation)

The users guide explains what they all are (expect the forced one). The ones we want are either the chk files or the plt files. The chk files, there will be 6 of them for the 2D Sedov simulation are at different time steps in the simulation. From those you can see the time evolution. The plt file is the final results at the end of the simulation (I believe).

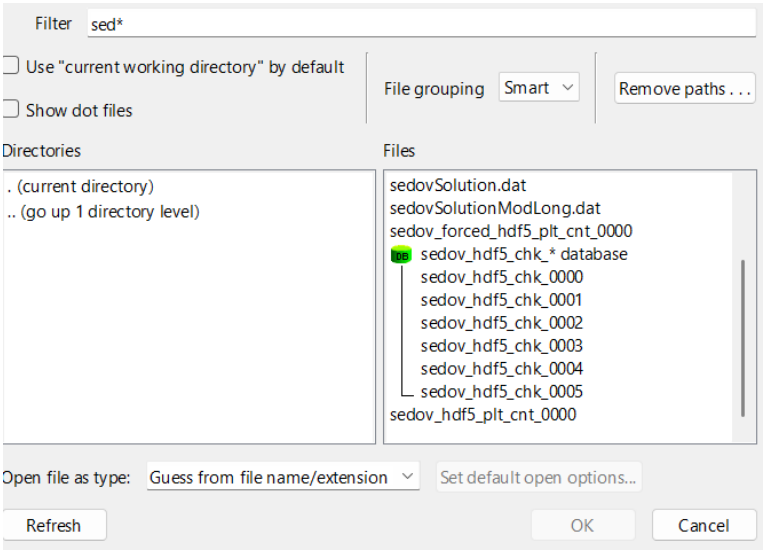
To look at the results, we'll use VisIT. Boot it up and you'll see two main windows as shown below. The left one is where you'll load files and pick parameters to plot. The right one is where the data will show up.



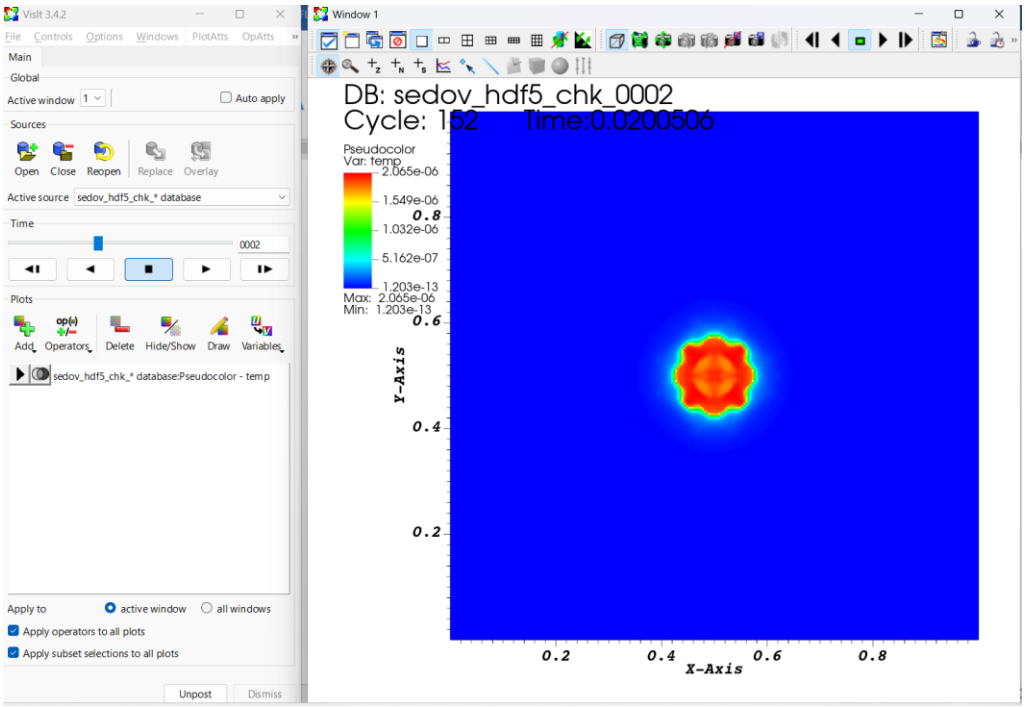
The “Open” button the left side lets us select a data file or data file set to load. It’ll pop up a new file selection window. The default path tends to be where VisIT was installed, which is kind of useless since our FLASH outputs are in a different folder. You can use the Directories windows on the left to get to the flash folder on your computer where the Sedov simulation results reside. The Path dropdown will remember all the places, and you can click the “Remove paths” button to prune the list down to just the one(s) you want to keep. To make sure VisIT remember that path for the future, hit cancel and close the window. Then go to the “Options” dropdown menu and select “save settings.” That will save the paths for the future so you don’t have to go find it every time you load VisIT.



When you load the object folder, there will be a ton of files. To filter that down, you can type "sed*" into the Filter textbox and hit enter. VisIT will group similar files, like the `sedov_hdf5_chk_000x` ones. If you select them, they'll all be highlighted, which is good since we want to load all of them as they are the simulation results at different timesteps.



Once it's loaded, the buttons under "Plots" should light up. Hit "Add" and you'll have a dropdown list of types of plots. Hover over one and it'll show you what you can plot that way. Once you select the plot, it goes into the list. To make it appear in the plot window on the right, you have to select the item from the list and hit "Draw". Below I plotted pseudocolor temp.



You can add more plots to the list, and use Draw to put it up. It seems you can only show one type of plot at a time, but you can show multiple types, like a pesudocolor and a contour. You "Hide/Show" to display different ones. It looks like only one plot can be show at any time.

The left and right arrows will play the data forward or backwards, or let you step through them one at a time.

To export or save the images, use "save movie" under the File menu. You can save it as a set of images, or as a mpeg video. I think mpeg video is less common these days, so saving images and turning them into an animated gif or a mp4 video may be better for presentations and such. Clipchamp is a free video editing program from Microsoft and can be used free, but it requires an internet connection (I think it sends your videos through Microsoft, so they're getting your data). I've heard DaVinci Resolve is offline and has a free version.

7. Conclusion

So we've gone through how to get FLASH set up using the Docker image, which should work for any computer that can install Docker. It's relatively easy since all the Linux libraries are preconfigured, but simulations run slower on Docker. If you just want to play around with FLASH a little bit, then it's probably sufficient. But, if you plan to do lots of simulations, then going the longer route and installing Ubuntu and the libraries is probably better. That's the next part. So onward.

I've found the Docker container version is useful for cross comparison with the self-installed version, especially if there's an error in the self version, I have been able to at time go to the Docker version and see what it did. The Docker version doesn't seem to have Chombo though, so that's not a useful comparison.

THE LONGER METHOD WITH LINUX
(FLASH via WSL, using Synaptic to install packages)

This is the method if you want to run FLASH "natively" on your computer through WSL (though it's not fully native, that would be if you install a fully Linux partition and boot into it, but that's too much work). It's also how you'd have to do it if you were setting up a remote computer cluster or server to do the simulation runs (which you probably would boot as a fully Linux machine). The benefit of this method is that you can use the most up to date version of the dependent libraries and programs like gfortran 13 instead of 8.5. The Docker flash4-deps image is a bit old, so uses older versions of stuff. The flash users guide lists out the libraries and programs you'll need, along with websites to get them. Unfortunately, some of the links are out of date and the actual files have moved. And most can be installed without downloading stuff from a webpage. I tried to give updated info here and faster ways that I did it. Running it directly through WSL is also faster than through Docker, which may be important if your simulations get big.

The Synaptic part refers to using a GUI package manager for Linux that lets you better pick and choose what versions of stuff to install and access more repositories than what comes native to Ubuntu. It's where you can find the latest pre-packaged releases of the libraries and programs below. The downside of this method is that you can't pick where the libraries get installed. So there's a lot more reconfiguring FLASH's code later.

1. Get all the stuff you'll need installed

The programs/libraries you need are:

1) A Linux OS, I used Windows Subsystem for Linux (WSL)
    a. GNU make
2) Visual Studio Code (VS Code) for a more streamlined work environment (my personal preference)
3) Fortran90 compiler
4) Python
5) Message Passing Interface (MPI) library
6) Hierarchical Data Structure (HDF5)
7) Parallel netCDF (PnetCDF)
8) HYPRE solver
9) Chombo for adaptive mesh refinement (this is optional for now)
10) VisIT for visualization of the results
11) The FLASH code itself

WSL

WSL is built into Windows, you just need to turn it on. Open a command line interface (CLI), like Windows Powershell or just search "CMD" in the start menu. In the terminal (black box with text input) that pops up, type

```
wsl --install
```

(Yes, two dashes) This takes care of a number of things to get WSL setup. For one, it turns on the two features in Windows needed, Virtual Machine and WSL (you could turn these one manually by going to “Turn Windows features on and off”). That command also installed the Linux kernel, and the latest Ubuntu Linux OS (called a distribution). During the installation, it will ask you for a username and password. The username will create a directory within the Ubuntu OS, and the password is needed when you want to change things about the installation, so remember the password.

Once Ubuntu is installed, you’ll want to do some basic updates by using the following two commands

```
sudo apt update
sudo apt upgrade
```

The first will update the list of known libraries and programs that Ubuntu can install automatically, when you tell it to. The second upgrades all the libraries to their latest version. From here on, I’ll refer to Linux, WSL, and Ubuntu interchangeably. But they all mean to work in or give commands to the Linux OS Ubuntu within WSL.

You’ll want to install a few libraries and packages upfront since the other components will need them to install. These are m4, make, and subversion. Use the following commands one at a time to install them. Some will says “XXX MB of additional disk space will be used” and ask if you want to continue [Y/N]. Type “y” and hit enter to install it.

```
sudo apt install make
sudo apt install subversion
sudo apt install m4
```

Note: if you have both Docker Desktop and WLS Ubuntu installed, when you run WSL by typing “wsl” into the terminal, it will start up whichever ones is set as default, probably what you installed first. To check what’s set as default, enter the command

```
wsl –l –v
```

That will show you the installed Linux OS’s and the one with the asterisk is the default. To change the default, use

```
wsl --set-default distribution_name
```

(No space between set and the -default)

Note 2: If you ever need to reset Ubuntu and delete everything (like if you massively screwed up the dependency libraries or something), you can go to a CLI and us e

```
wsl --unregister Ubuntu
```

That will delete the Ubuntu instance and all the files. You can then reinstall it with

```
wsl --install Ubuntu
```

VS Code

At this point, you can continue using WSL via CMD or powershell. But a few things are easier to do if you have a file manager and directory you can use to edit files directly. VS Code is an open source code editor made by Microsoft for Windows (it kind of look like Matlab). It can work with all sorts of languages by using extensions, which are downloaded from within the program. You can get VS Code for Windows from https://code.visualstudio.com/.

To use VS Code to control WSL/Ubuntu and view and edit the files, you'll need the WSL extension which can be easily found using the extension icon. Once it's installed, if you click the blue double arrow () square at the bottom left, it'll pop up the search bar and you can look for "Connect to WSL". Select that and VS Code will connect to your WSL distribution (may take a bit as it needs to do some stuff). Once it's connected, the bottom left should say "WSL:Ubuntu". If you click the Explorer icon on the left bar, it'll say "no folders opened". Click the open folders button and it'll go to the search bar and you can find the folders in the WSL distribution. Generally, you just want to mess with your /home/username directory, and that's where we'll put our files usually. Now you can see the folders and files in your /home/username directory, open them in the VS Code editor, and if you hit Ctrl + ` , you can get a terminal to do your commands.

From here on you can work in VS Code if you want, and give WSL commands in the VS Code terminal once you're connected to WSL.

Gfortran and python

FLASH requires fortran and python compilers. A compiler is a program that converts the text based input files we write into machine language that the computer can run. Gfortran is an open-source compiler that's used pretty widely. In WSL, this is easy since there are known repositories it can pull from. Just type into the terminal

```
sudo apt install gfortran
```

When you installed Ubuntu, it comes with python3. You can check it's installed correctly using

```
gfortran --verision
python3 --version
```

Using Synaptic

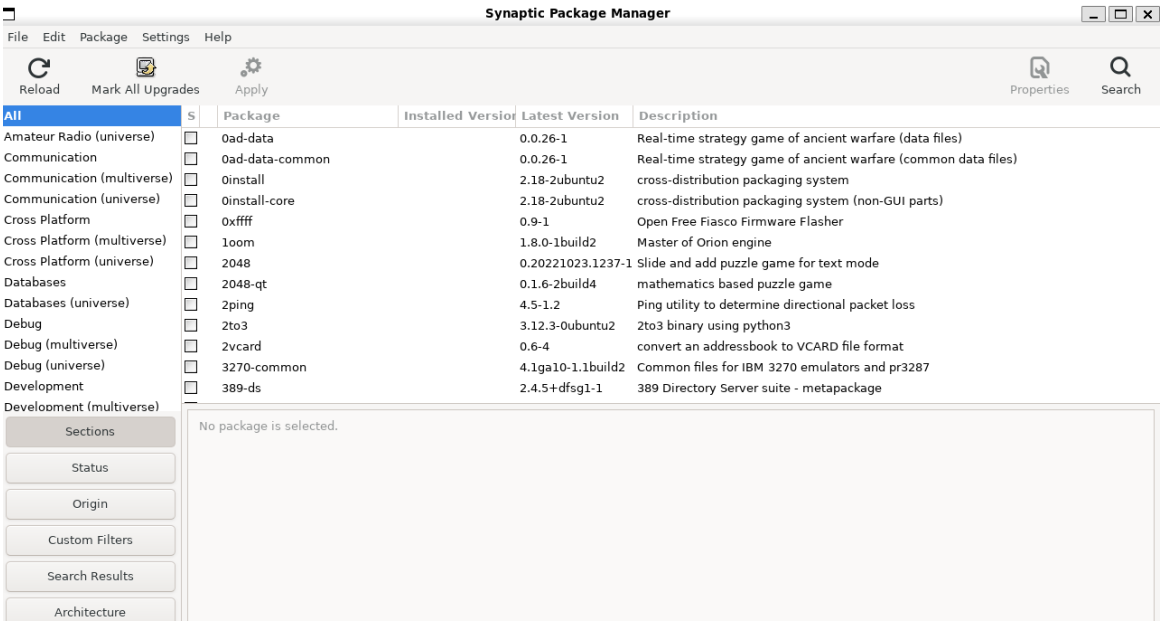
Now we get to the annoying libraries and dependent programs for FLASH. I used Synaptic for this. You can install Synaptic from WSL terminal with

```
sudo apt install synaptic
```

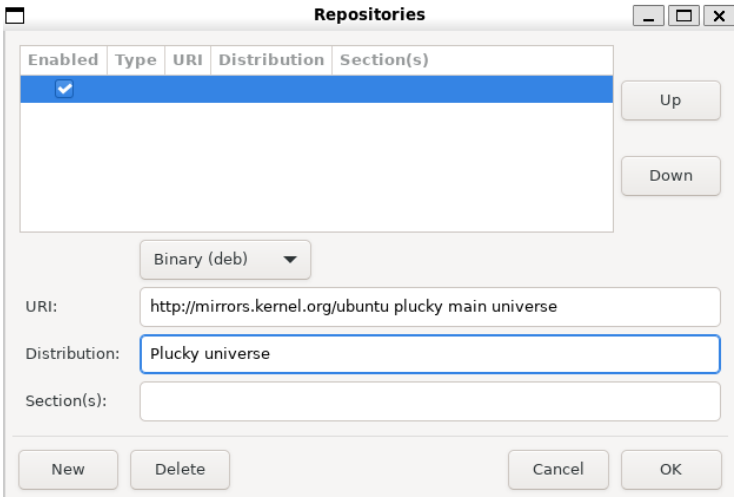
Then run it with

```
sudo synaptic
```

A window will pop up (see below) that you can click on and interact with like a Windows program. First, we'll want to point it to additional repository mirrors where updated libraries can be found. There's a list of packages for Ubuntu at https://packages.ubuntu.com/ that you can look through and do searches.



We'll add the "plucky universe" and "questing universe" repositories with the mirrors.kernel.org/ubuntu mirror. To add that to Synaptic, go to the Settings->Repositories menu, click New, and enter "http://mirrors.kernel.org/ubuntu plucky main universe" as shown below, and Plucky universe for the distribution. It'll tell you to reload, so click the reload button. An error might pop up that says it couldn't load some stuff, and that's fine. Close that warning window. Then do that again and add "http://mirrors.kernel.org/ubuntu questing main universe" and Questing universe as distribution. The questing repository has a couple of things Plucky doesn't. (FYI, the name plucky and questing refer to certain versions of Ubuntu).



Back on the main window, click search and type “hdf5”. Scroll down and you should see libhdf5-dev with the latest version of 1.14.5. That means it works and you can get the latest version of stuff. If you had just used sudo apt install, you’d get the 1.10.10 version.

MPICH

MPI enable parallel processing and the mpirun command, which is used to run parallel FLASH simulations with multiple processors. It’s kind of required to do most stuff in FLASH.

The flash users guide says to get a version of the MPI library called MPICH from Argonne National Laboratory (new webpage at https://www.mpich.org/). That site does maintain the latest version, which is 4.3.1 at the last check. But, a version of MPICH is also available from the regular Ubuntu apt repository, it’s just an older version (4.2.0 at last check). The questing repository through Synaptic has the 4.3.1. So lets use Synaptic.

In Synaptic, search for “mpich” and scroll down until you find mpich with 4.3.1 version. Click the checkbox and select “mark for installation”. A window might pop up that says it also needs to install or update a number of other files and if you want to mark additional changes. Click “mark.” To actually install, you have to hit the “Apply” button at the top. It’ll give a summary of what it’s going to do, so just hit Apply and wait.

When it’s done, a window that says changes applied will appear. You can scroll down the mpich list and see the checkbox next to mpich and a couple of other things are green, indicating it’s been installed. Now, I’m paranoid (since Linux is weird) so I like to check occasionally that the thing actually got installed. So exit Synaptic, and go back to the terminal and type

```
mpirun –version
```

If it’s there, it’ll show a bunch of stuff and the version, which indicates mpich is good.

HDF

HDF is a data structure that is used to organize and store large and complex data sets, like FLASH simulations and outputs.

The user guide says to get HDF5 serial version from their webpage (https://www.hdfgroup.org/). A version of HDF5 serial exists in Ubuntu’s repositories. Using Synaptic, and search for “hdf5” will give a whole list of stuff. The two to mark are “libhdf5-dev” and “libhdf5-mpich-dev”. They’ll ask to also mark and install other dependent libraries. The first one is the serial HDF5, and the second is the parallel with MPICH (or MPI). Both can/are used by FLASH.

You can check if this install correctly by going to terminal and using

```
h5dump –showconfig
```

That’ll show the details if it’s there.

PnetCDF
The users guide said to get PnetCDF from Argonne National Laboratory (the page has moved to https://parallel-netcdf.github.io/). A version exists in the Ubuntu repositories. Search for "pnetcdf" on Synaptic and there's only a few. I used "libpnetcdf-dev". The -dev portion means it has additional information and functions if you want to develop new stuff for/with the library. But it also has all the standard files. I think if you just want the library files without the development things, then it's the "pnetcdf-bin" version. I installed both.

You can now check if everything went well by doing

```
which ncmpidump
which ncmpidiff
```

If it all install correctly and the PATH was added, it'll return where those files are located, i.e. /usr/bin/ncmpidiff. You can also check the version and that it matches what you installed.

```
ncmpidump -version
```

It seems that you only get ncmpidump and ncmpidiff if you install the pnetcdf-bin version, and not the -dev version.

HYPRE
The webpage is at https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods/software. But the actual files are in GitHub (https://github.com/hypre-space/hypre).

You can also find them on Synaptic. The latest version at last check is 2.33.0. I used "libhypre-2.32.0".

Since it's a library, can't do a version check in terminal to make sure it installed correctly. You'll just have to trust the green box in Synpatic.

Chombo (haven't gotten this one to work yet, but it seems optional for if you want to use Chombo for adaptive mesh refinement. The Docker image doesn't seem to have it either)
This unfortunately has no pre made package installer than I could find. So have to download this and install manually. The webpage is at https://commons.lbl.gov/spaces/chombo/pages/78753529/Chombo+Download+Page. You'll have to register with them in order to download the files. You actually download the files from WSL using the subversion (svn) command.

Then go into your /home/username directory and download Chombo with the following command, where instead of "username" you put what you registered with (remove the ""). It'll ask you for the password once the command runs.

```
svn --username "username" co https://anag-
      repo.lbl.gov/svn/Chombo/release/3.2 Chombo-3.2
```

This will put a “Chombo-3.2” directory with all the files in your /home/username directory.

(I haven’t gotten this to install correct and it throws errors.)

VisIT

VisIT is an open source visualization program made by LLNL which has built in recognition of FLASH output. You can get VisIT for windows from https://visit-dav.github.io/visit-website/, and then install it and normal. The users guide first says to get IDL, but that is a commercial program.

On the VisIT page, go to Downloads, the Releases. Scroll down a bit and there’s a table with the latest release for different OS’s. I used Windows 10, and the “use” link to download the install file. Then install the program. During the installation, it’ll ask you if you want to pick a default database reader plugin. I selected FLASH since that’s what I’d be using it for, and it’ll save a click when looking at the results. But if you don’t want to pick on, then select None.

Use the None for network configuration, and you can associate FLASH extensions with VisIT, but I’m not sure if it helps a whole lot.

FLASH code

FLASH code is managed by the University of Rochester. You can request access to the code at the following page: https://flash.rochester.edu/site/flashcode.html. It takes about a day or two for the request to be processed, after which you’ll get an email with username and password to download the program.

It’ll be downloaded as a .tar.gz compressed file that you have to extract (like a zip file). Windows should be able to do it natively, but since we’re build a WSL/Linux environment, then it’s easier to extract it in WSL. After you download the FLASH4.8.tar.gz file to windows, open the Linux Ubuntu folder in the windows files explorer, go to /home/username, and copy the compressed .tar.gz file into there. Then go to the /home/username directory in WSL, check that the FLASH4.8.tar.gz file is there using the “dir” command, and then using the following command to extract all the files into a folder called FLASH4.8 in the /home/username directory.

```
tar -xvzf FLASH4.8.tar.gz
```

You can then go into that directory with “`cd FLASH4.8`”, and yes capitalization matters here.

Sometimes you might get a permission denied error when you try the tar command. That can happen when you use Windows file explorer to copy a file into WSL. You can give yourself permission to execute it with

```
chmod +x filename
```

You can check permissions with

```
ls -l filename
```

- When you copy a file from Windows into WSL via the file explorer, Windows generates a file with the extension .Zone.Identifier. This apparently stores the metadata information about where the file came from for Windows security purposes. But WSL's file system doesn't function the same way as Windows, so to preserve that info, Windows creates this extra file. You can delete it without issue since it doesn't affect Linux.

2. Run the basics Sedov 2D blast wave simulation (and code fixes)

Now you can follow the FLASH user guide for running a simulation, but I'll also explain it here with some error fixes that I ran into. While I could have given the instructions here to fix all the errors at once, I think it's more instructive if you run into the errors as your go and understand what the fixes are doing.

Start up VS Code and remote connect to WSL. Then open the folder /home/username so you can find and more edit open and edit files. Enter the following in the terminal

```
cd /home/username/FLASH4.8
```

That'll open the flash directory, then if you type "dir" and hit enter, it will show you the files and folders in that flash directory. One of them should be "setup". The setup file is how you start a FLASH simulation (there's 3 steps to prepare the program). To set up the Sedov 2D blast wave, type

```
./setup Sedov -auto
```

The ./ tells Linux to run a script or program in the current directory. The –auto part tells it to automatically set the units. You'll see a number of lines in the terminal as the code sets up the simulation. The "Sedov" part refers to the Sedov folder in your FLASH4.8 directory that's located in FLASH4.8\source\Simulation\SimulationMain. If you look in SimulationMain, you'll find all the preconfigured simulations provided with FLASH. Now, not all of them work out of the box in my experience, but Sedov does.

If the setup work, then it'll say "SUCCESS" in the terminal and a new folder called "object" will appear in the FLASH4.8 directory. That's where the details of the current simulation that was just set up (or last set up) resides. But, if this is the first time you're running FLASH, you'll get a number of error to fix. Let do it.

Error 1: You may see a number of SyntaxWarnings like below:

```
/home/username/FLASH4.8/bin/libCfg.py:110: SyntaxWarning: invalid escape
sequence '\S'
  self.initParser('LIBRARY', {},
'LIBRARY\\s+(\S+)\\s*((?:[^\[])*)\\s*(?:[\[](\S*)[\]])?\\s*$')
```

This is one of the things the modern version of python does that older ones, like the one in Docker, doesn't. If you go to that file, libCfg.py, in the WSL file explorer and open it up, you'll find line

110 is that LIBRARY thing. Apparently after some version of python, it issues syntax warning when there's something it doesn't like, but it keeps compiling. In the future though, it may just stop the program altogether. So you can fix it. My fix is kind of dumb, and that's to just add another \ to the current ones in that line, and line 117. See the snippet of code below that shows it. That should get rid of the syntax warning next time you run ./setup.

```
    def parseLIBRARY(self, line):
        self.initParser('LIBRARY', {},
'LIBRARY\\s+(\\S+)\\s*((?:[^\\[])*)\\s*(?:[\\[](\\S*)[\\]])?\\s*$')
        libmatch = self.match('LIBRARY',line)
        libname = libmatch.group(1).lower()
        libargs = " ".join(libmatch.group(2).split()) # trims and removes
multiple spaces
        self['LIBRARY'][libname] = libargs

    def parseTYPE(self,line):
        self.initParser('TYPE', "EXTERNAL",'TYPE\\s+(INTERNAL|EXTERNAL)\\s*$')
        self['TYPE']= self.match('TYPE',line).group(1)
```

Even with the syntax warnings, it'll typically still finish setup and say SUCCESS. Now you'll need to go into the object directory and compile the FLASH code. Type the follow two commands in the terminal

```
cd object
make
```

The first command goes into the object folder. The second one tells Linux to compile the simulation and create an executable file. Make is a build-in function in Linux, and it depends on a Makefile, which was created in the object folder by the setup command. Makefile is a text file that gives instructions on how to compile the executable to run the simulation. Once you run the "make" command, a number of things will run in the terminal, and probably throw errors. You can actually run make with multiple processors using "`make -jX`", where X is the number of cores to use, e.g. `make -j3`.

Error 2: The first time you run make, it may give you an error that says

```
make: /usr/local/mpich2//bin/mpif90: Command not found
make: *** [Makefile:132: Burn_interface.o] Error 127
```

What that means is Linux tried to use the mpif90 program located in the "/usr/local/mpich2/bin/" directory in the container, but it wasn't there. The mpif90 function is there, it's just in a different place, so we have to change it. To find where it is, type

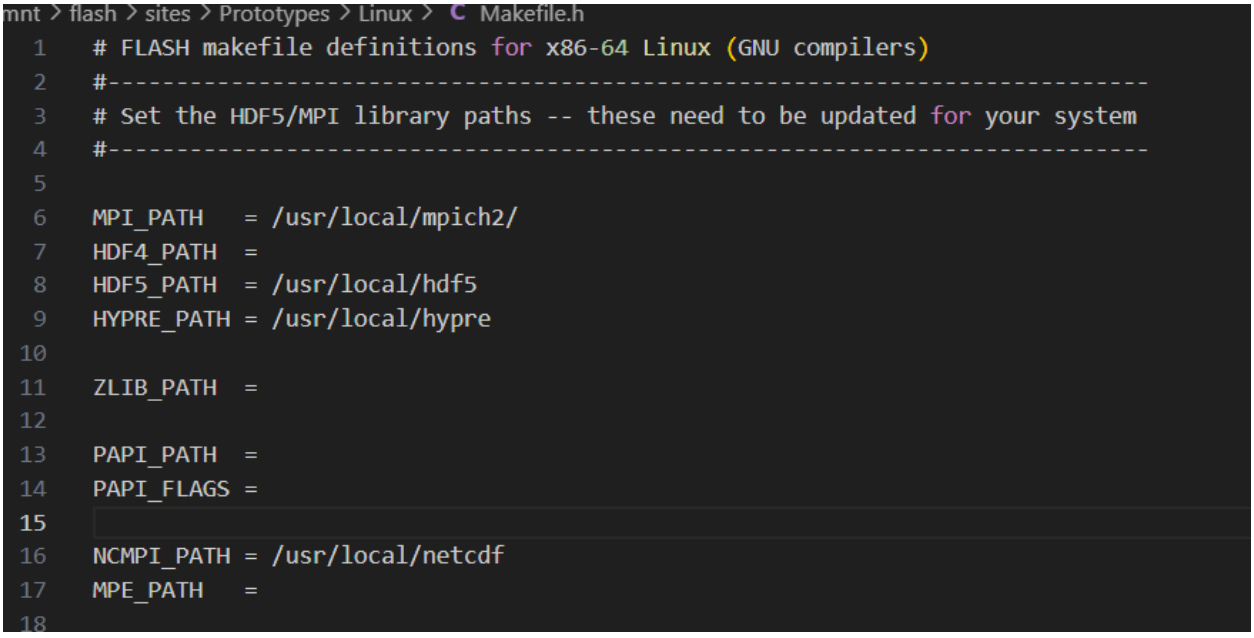
```
which mpif90
```

The "which" command will tell WSL to find that program, which for me gave back "/usr/bin/mpif90". So it's located in a slightly different place. The solution is to change where the

code is looking for the mpif90 file. That will be in a Makefile, which is the set of text files that tell the code what to do and where to look for stuff. So lets find it and change it.

Go to the folder "/FLASH4.8/sites/Prototype/Linux" either on your computer on in VS Code. If in VS Code, then your FLASH4.8 folder is under "/mnt/flash". Open the Makefile.h you find there using either VS Code's editor or notepad in windows. At the top of the Makefile is the library path definition, which I've screenshotted below. You'll need to change the first one for "MPI_PATH" from "/usr/local/mpich2/" to "/usr" (yes, also remove the trailing /, that's apparently another oopsie in the code).

- The Prototype folder appears to be for the local computer. In the sites directory, you'll see bunch of folders that refer to various universities and labs. I believe those are predefined instructions for people at those places to use a central computer cluster or supercomputer for their FLASH simulations. Since we don't have access to the other sites, I deleted the rest except for Prototype to make getting to it faster. You can always recover those folders by extracting the FLASH4.8.tar.gz again if needed.
- The Makefile automatically add a /bin to that path. That's in lines 29 and 32. I think you can change that and delete the bin there and add bin to line 6 instead. I didn't try though.



Now if you run make again when in the "object" folder, it'll do a bunch of Linux commands to compile the simulation, but you might get errors still.

Error 3: I got a message of

```
f951: Fatal Error: Reading module 'iso_c_binding' at line 1 column 1:
Unexpected EOF
```

If you go back to the Makefile in the "Prototype" folder, and scroll to the bottom, you'll find the following section

```
#-------------------------------------------------------------------------
# Fake existence of iso_c_bindings module to prevent unnecessary recompilations.
#-------------------------------------------------------------------------
ifeq ($(FLASHBINARY),true)
iso_c_binding.mod :
   touch $@
endif
```

According to the FLASH USERs mailing list, this is apparently an old workaround for old fortran compliers, but it no longer necessary for modern ones. So, delete that section and save the Makefile. Now, if you try running make again, it'll still error out with the same one. That's because the old object configuration files are still there. To fix that, use the following command

```
make clean
```

That will remove all the files created during the last make command. Now you run make again and it should work, or most likely through an error after a few minutes.

Error 4: The next error I got said: /usr/bin/env: 'python': No such file or directory

I think this is because FLASH is looking for python, but instead modern python is python3. So instead of trying to figure out where in the massive FLASH code it's calling for it, I added a link within Ubuntu that says whenever something asks for python, point it to python 3. Use following command.

```
sudo ln -s /usr/bin/python3 /usr/bin/python
```

Error 5: The next error was a fatal error:

```
/usr/bin/mpicc -I/usr/local/hdf5/include -DH5_USE_16_API -ggdb -c -O2 -
Wuninitialized -D_FORTIFY_SOURCE=2 -DMAXBLOCKS=1000 -DNXB=8 -DNYB=8 -
DNZB=1 -DN_DIM=2 io_attribute.c
In file included from io_h5_attribute.h:6,
                 from io_attribute.h:13,
                 from io_attribute.c:2:
hdf5_flash.h:34:10: fatal error: hdf5.h: No such file or directory
   34 | #include "hdf5.h"
```

So this one is another one of those cases where it's looking for a file in a place but can't find it. This time it's looking for the hdf5.h file that was installed with the hdf5 libraries, and looking for it in "/usr/include/hdf5/include". If you open the Makefile.h in the /sites/Prototypes/Linux folder, line 8 has "HDF5_PATH = /usr/local/hdf5". But my hdf5 folder was in the "/usr/include" directory. And I had two of them, mpich and serial, since I installed both. Sedov only uses the serial one I believe (I read in one FLASH-USER thread that the mpich version is only needed if you get to like 10,000 cpu hours).

Secondly, you'll note that the first line in the error has "local/hdf5/include". That last include is added in line 88 of the Makefile.h.

```
CFLAGS_HDF5 = -I${HDF5_PATH}/include -DH5_USE_16_API
```

I change the path and deleted the "include" so it now points to where the hdf5 serial files are located on my WSL verison

```
HDF5_PATH = /usr/include/hdf5/serial
CFLAGS_HDF5 = -I${HDF5_PATH}/ -DH5_USE_16_API
```

Error 6: Another error where the file is in different place

```
-L /usr/include/hdf5/serial/ -lhdf5 -lz
/usr/bin/ld: cannot find -lhdf5: No such file or directory
collect2: error: ld returned 1 exit status
make: *** [Makefile:384: flash4] Error 1
```

The "ld" here in the /usr/bin/ld refers to a linking utility in Linux. The file it's looking for is a library called "libhdf5.so" (the -lhdf5 appears to be shorthand for libhdf5.so). Line 119 in Makefile.h is where it tells the compiler where to look for that file, and it's based on the HDF5_PATH defined earlier.

```
LIB_HDF5  = -L ${HDF5_PATH}/lib -lhdf5 -lz
```

But, my install of hdf5 obviously didn't go what FLASH thought it was going to do. Instead, it didn't something random (or seems random to me). It put the majority of the hdf5 files into the aforementioned "/usr/include/hdft/serial" folder. But, the libhadf files are in "usr\lib\x86_64-linux-gnu\hdf5\serial" (Why…just why Linux). So, I replaced the address in line 119 with the following

```
LIB_HDF5  = -L /usr/lib/x86_64-linux-gnu/hdf5/serial -lhdf5 -lz
```

And finally make finishes and it says "SUCCESS" when it's finished and worked, and will give you the terminal command prompt again. The complier will create a "flash4" file in the "object" folder. That's the thing you'll execute to actually run the simulation. You can do it one of two ways depending on how much processing power you want your computer to use for it. The simplest is to just type

```
./flash4
```

This will run the simulation using a single core on your computer processor. You can use multiple processors by using mpi (message passing interface, which allows the processors to send stuff back and forth and share the load). To do that, use the follow command

```
mpirun -np N ./flash4
```

Where *N* is the number of processors you want to use.

- The flash users guide says to type “mpirun -np N flashX” where the X would be 4. However, this seems to cause a problem and throws an error “HYDU_create_process (lib/utils/launch.c:73): execvp error on file flash4 (No such file or directory)”. This means for whatever reason it can’t find the flash4 file. Adding the “./” tells it to look in the current directory, which is the “object” folder where it should be.

When the simulation runs, you’ll see tables of numbers scroll as it does calculations and stuff.

Now, if you want to see how much time the simulation takes, you can use the time command in Linux. Simply put “time” in front of the command, for example

```
time ./flash4
time mpirun -np 3 ./flash4
```

This will cause the command to run as normal, but after it’s done, you’ll also get how long the process took. On my laptop, running with 1, 2, and 3 cores took 12, 7, and 8 seconds. Noticeably faster than the Docker version.

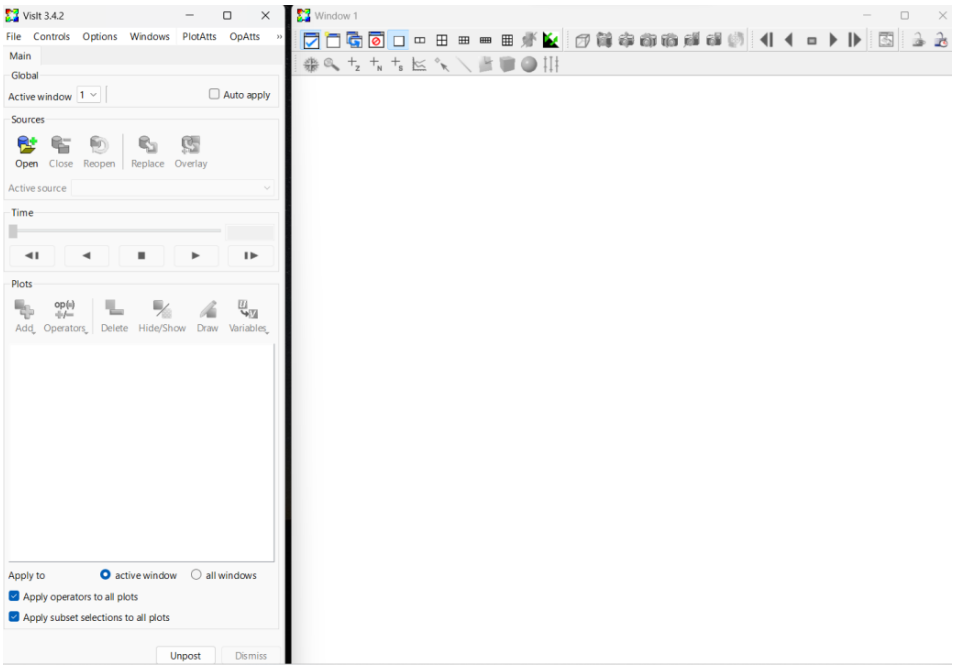
3. Plotting the results (same as the Docker section)

If the simulation completes, it will create a new set of files in the “object” folder. It’ll start with the name of the simulation, sedov in this case.

sedov.log
sedov.dat
sedov_hdf5_chk_000x (the x here is a number)
sedov_hdf5_plt_cnt_000x
sedov_forced_hdf5_plt_cnt_000x (this one doesn’t show up for all simulation)

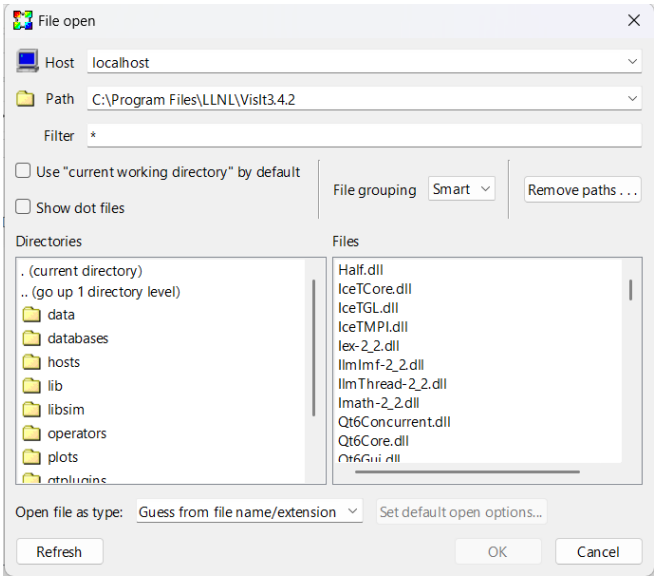
The users guide explains what they all are (expect the forced one). The ones we want are either the chk files or the plt files. The chk files, there will be 6 of them for the 2D Sedov simulation are at different time steps in the simulation. From those you can see the time evolution. The plt file is the final results at the end of the simulation (I believe).

To look at the results, we’ll use VisIT. Boot it up and you’ll see two main windows as shown below. The left one is where you’ll load files and pick parameters to plot. The right one is where the data will show up.

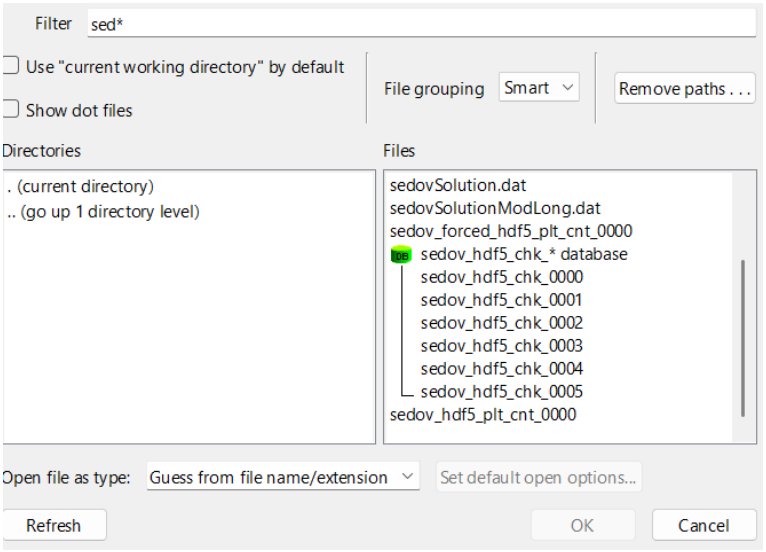


The “Open” button the left side lets us select a data file or data file set to load. It’ll pop up a new file selection window. The default path tends to be where VisIT was installed, which is kind of useless since our FLASH outputs are in a different folder. The path to your Ubuntu install in WSL isn’t easily accessible from the directories window. So you can more easily open a Windows file explorer, go to your FLASH4.8 folder in Ubuntu, copy the path and paste it into the path window VisIT. The Path dropdown will remember all the places, and you can click the “Remove paths” button to prune the list down to just the one(s) you want to keep.

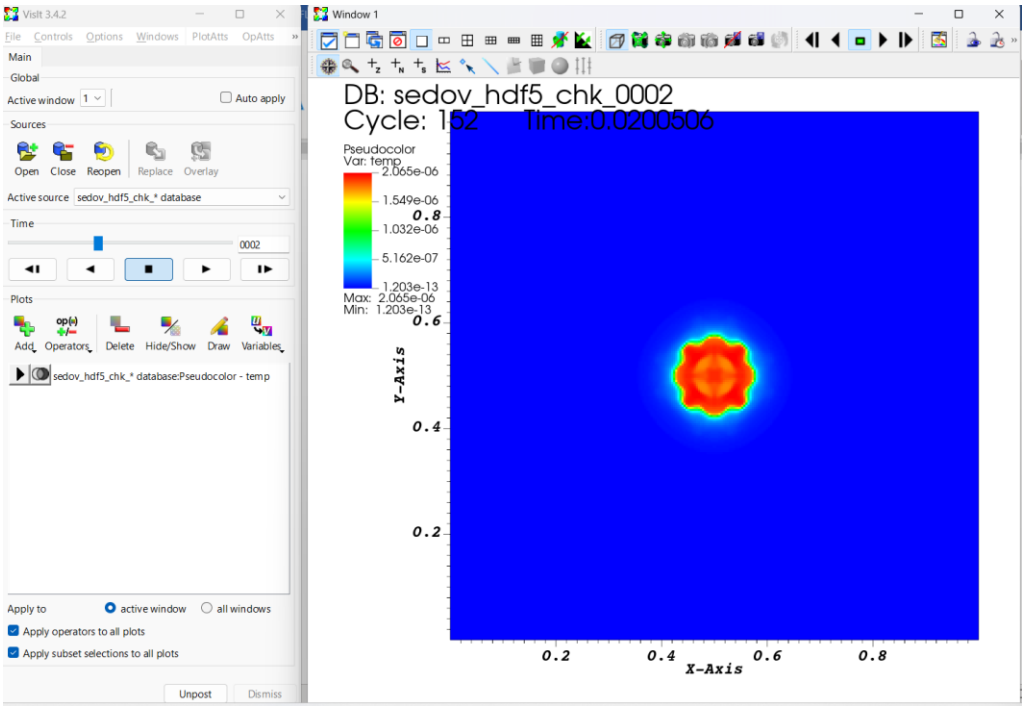
To make sure VisIT remember that path for the future, hit cancel and close the window. Then go to the “Options” dropdown menu and select “save settings.” That will save the paths for the future so you don’t have to go find it every time you load VisIT.



When you load the object folder, there will be a ton of files. To filter that down, you can type "sed*" into the Filter textbox and hit enter. VisIT will group similar files, like the `sedov_hdf5_chk_000x` ones. If you select them, they'll all be highlighted, which is good since we want to load all of them as they are the simulation results at different timesteps.



Once it's loaded, the buttons under "Plots" should light up. Hit "Add" and you'll have a dropdown list of types of plots. Hover over one and it'll show you what you can plot that way. Once you select the plot, it goes into the list. To make it appear in the plot window on the right, you have to select the item from the list and hit "Draw". Below I plotted pseudocolor temp.



You can add more plots to the list, and use Draw to put it up. It seems you can only show one type of plot at a time, but you can show multiple types, like a pesudocolor and a contour. You "Hide/Show" to display different ones. It looks like only one plot can be show at any time.

The left and right arrows will play the data forward or backwards, or let you step through them one at a time.

To export or save the images, use "save movie" under the File menu. You can save it as a set of images, or as a mpeg video. I think mpeg video is less common these days, so saving images and turning them into an animated gif or a mp4 video may be better for presentations and such. Clipchamp is a free video editing program from Microsoft and can be used free, but it requires an internet connection (I think it sends your videos through Microsoft, so they're getting your data). I've heard DaVinci Resolve is offline and has a free version.

4. Conclusion

So, we've gone through how to get FLASH set up using WSL which will run faster. It's not bad, at least if you aren't me trying to figure out this all for the first time solo.

MAKE AND EDITING SIMULATIONS

This a list of notes for what the code means and make new simulation/editing existing ones. This is a slow work in progress with my students and I. I do know the "Setting Up New Problems" section in the flash users guide doesn't work out the box as written, or at least it didn't for me.

- FLASH uses both fortran and python coding language. Earlier in the user guide it said that python is needed to run the setup script. Looking through the various files, it appears the Config and *.par files are in python, and the *.F90 and *.h files are in fortran. I'm judging this mainly by what's used as a comment marker in the code. Python uses # for comment, and fortran uses !. So that's confusing.
- Looks like you need minimum 6 files to run a simulation: Config, flash.par, Makefile, Simulation_data.F90, Simulation_init.F90, and Simulation_initBlock.F90
- The nend = 1000 line in flash.par tells the max number of iterations to run per time step. To see it's effect, you have to rerun ./setup and then make again. 1000 seems to be the default in the provided simulations I've looked at.
- The tmax = 0.2 sets the max simulation time.
- The checkpointFileIntervalTime = 0.1 sets the time steps between each data dump. Same for plotfileIntervalTime = 0.1.
- The lrefine_max variable in flash.par sets how much mesh refinement the code will do to improve the accuracy of the simulation. A higher the number, like >5, seems to start having an exponential increase in computational time. But that could just be my computer.