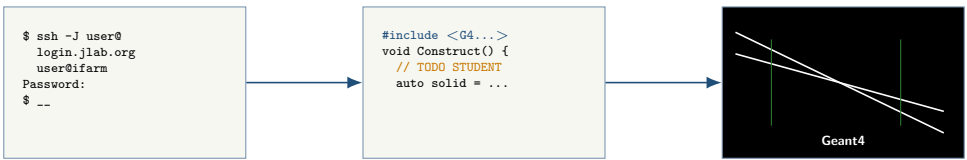
JLab Scientific Computing Hands-On Workshop

A four-part curriculum: from CLI to a personal cosmic muon telescope



Access `ifarm` | Edit & build | Simulate, analyze, push

Cameron Clarke
cameronc@jlab.org
Radiation Detector & Imaging Group, Jefferson Lab

What you will do today

- Log in to JLab's `ifarm` from your own laptop.
- Locate and build Geant4 example repository (B1)
- Compile and run a Geant4 simulation of a cosmic muon telescope.
- Choose your own central-detector material and shape; rerun the simulation.
- Analyze the output `TTree` two ways: a PyROOT script and a compiled C++ executable.
- Commit and push everything to your personal `code.jlab.org` repository.
- Open and merge a pull request in your own fork.

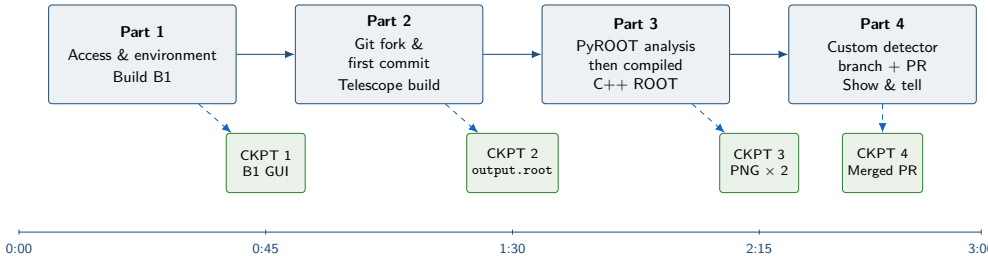
Key idea

By noon today you will own a personal git repository with a plot of *your* detector's muon-tagging efficiency vs zenith angle. No two attendees' plots will look the same.

Quick audience check — show of hands

1. Who has used a Linux or Mac **terminal** for more than 5 minutes?
2. Who has used `git` or `github` for any project?
3. Who has used `ROOT` or `Geant4` before today?
4. Who has used `ssh` to log into a remote computer?

The four-part map



Green boxes are the checkpoints we'll wait for everyone to pass. Allocate 3 hours.

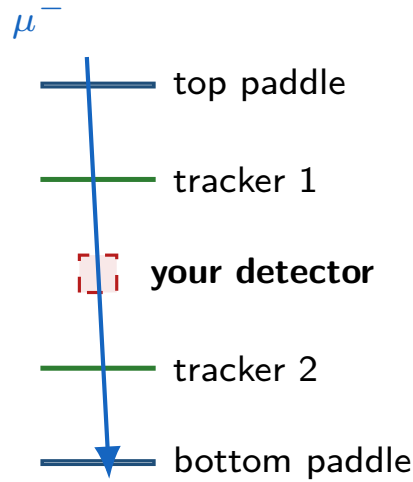
The physics we will simulate today

A **cosmic muon telescope**: a stack of detectors that lets you tag and reconstruct atmospheric muons one by one.

- Top & bottom paddles fire a trigger when a muon passes through both.
- Two or more tracking detector planes reconstruct the track.
- In the middle: **your** detector. You pick the material and the shape.
- We measure: directional and integral efficiency of proposed detector materials and geometries based on known cosmic muon tracks and angular spectrum.

Why this slide is shaped like this

Everyone picks a different detector implementation and gets a different, physically meaningful answer - the power of scientific computing.



A note on bash vs. tcsh

Jefferson Lab's default login shell can be changed for `ifarm`. Some prefer `tcsh` as the CLI interpreter, others `bash`.

Typically `bash` is used for repeated task scripting

- Scripts start with `#!/bin/bash` so they work regardless of your login shell.
- Check which default shell is set for your profile with `echo $SHELL`.
- If you would like to switch your default shell - send an email request to `helpdesk@jlab.org`.

Part 1
Access, environment, B1

The four directories you actually need today

Path	Use it for
`/home/<user>`	Your personal config files. **Small** (a few GB).
`/work/<group>`	Your active project code and small data. **Where we'll work today.**
`/group/<group>`	Shared group software and reference data.
`/volatile` or `/scratch`	Big simulation outputs you don't need to keep.

Today, everything we do lives under:

```
/work/<your-group>/<your-username>/20260514-GSPDA-Computing-Bootcamp/
```

Potential hurdle

If `/work/<your-group>` doesn't exist for you yet, use your /home/<username> folder for now, and reach out to your compute coordinator for help

Step 1: Access ifarm

```
Connect via VSCode
```

Or: Open a terminal on your laptop now (macOS Terminal, Linux terminal, or Windows Terminal / WSL) and type:

```
ssh -J <user>@login.jlab.org <user>@ifarm
```

At the two password prompts:

- For `login.jlab.org`: your MFA passcode = PIN + 6-digit MobilePASS code, typed on one line.
- For `ifarm`: your CUE password.

When you see a prompt ending in `ifarm2401$` or `ifarm2402$`, you're in.

Step 2: switch to bash, find your work directory

Once you see the `ifarm` prompt:

Type this exact command

```
bash
hostname # ifarm2401 or ifarm2402
mkdir -p /work/<your-group>/$USER
cd /work/<your-group>/$USER
pwd
```

Your group name should be whichever hall or main project you work on - if none exists stick to using your /home/<username> folder for this workshop

Step 3: load Geant4 and ROOT from CVMFS

CVMFS is a global read-only filesystem mounted on every JLab compute node. Everything we need to compile and run is already there.

Type this exact command

```
module use /cvmfs/oasis.opensciencegrid.org/jlab/scicomp/sw/el9/modulefiles
module use /cvmfs/oasis.opensciencegrid.org/jlab/geant4/modules

module load root
module load geant4/11.3.2

which root && root --version
which geant4-config && geant4-config --version
```

Expected output (versions may differ slightly): `ROOT 6.30.x`, `Geant4 11.3.2`, both living under `/cvmfs/....`

Why CVMFS? Why not install Geant4 yourself?

- CERN built CVMFS to deliver physics software to thousands of compute nodes worldwide without local installation.
- JLab mirrors a subset under `/cvmfs/oasis.opensciencegrid.org/jlab/`.
- You get *the same* ROOT and Geant4 builds as everyone else.
- Updates happen centrally; your simulation results don't quietly diverge from the rest of the collaboration's.
- conda-forge packaged ROOT and Geant4 can be used instead, but CVMFS are explicitly supported for ifarm

Why this slide is shaped like this

Installing Geant4 from source takes hours and frequently breaks on system-library mismatches.
We aren't going to do it today; you would not get to the physics.
CVMFS gives us the same Geant4 the production farm uses, in one second.

Geant4 example B1: the simplest meaningful sim

The Geant4 distribution ships with about a dozen "basic" examples, labeled B1–B5. B1 is the simplest one.

What it simulates:

- A 6 MeV gamma source fires at a small water box.
- Inside the box: a cone of tissue, a trapezoid of bone.
- The simulation tracks each gamma through the materials and reports the dose deposited.

Why we compile it first: to prove the toolchain works *before* we touch our own code.

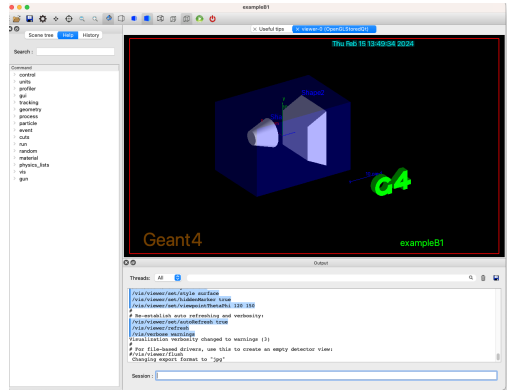


Figure: B1 Example Geant4 GUI.

Step 4: build the unmodified B1

Type this exact command

```
# Copy the example source to your work directory
cp -r $G4INSTALL/data/Geant4/examples/basic/B1 ./B1
# Build out-of-source (standard CMake practice)
mkdir B1-build && cd B1-build
cmake ../B1
make -j4
# Run it -- this opens the OpenGL visualizer
./exampleB1
```

What's happening:

- `cmake` reads `CMakeLists.txt` and writes a `Makefile` appropriate for this machine.
- `make -j4` runs the compiler in parallel on 4 cores.
- `./exampleB1` launches the sim. **WARNING**: Uncomment threads in init_vis.mac!

This is what success looks like

You should see a window like the one on the right.

In the Geant4 prompt at the bottom of the window, type:

```
/run/beamOn 100
```

You will see 100 gammas fired, drawn as colored tracks scattering off the materials.

If you can do this, your Geant4 toolchain works.

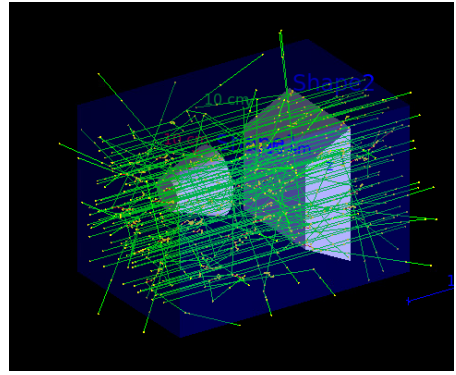


Figure: B1 Example Geant4 GUI with tracks.

Checkpoint 1 of 4

Checkpoint

Everyone show me the B1 GUI with at least 10 tracks visible.
Wait for anyone whose build failed or whose X-forwarding isn't working.

Part 2
Git fork, telescope build

What is git

- Git is a **versioned save-button** with history. Every `commit` is a save point you can return to.
- A **remote** (here: `code.jlab.org`) is a copy of your repo on a server, so your work survives your laptop dying.
- A **fork** is your own personal copy of someone else's repository, where you can experiment freely without affecting theirs.

Key idea

By the end of Part 2 every change you make today will be backed up to `code.jlab.org`. If your laptop catches fire, you can replay your work from any other computer in three commands.

Step 5: fork the baseline repository

In your browser, go to: code.jlab.org and log in with CUE credentials

`code.jlab.org/cameronc/20260514-GSPDA-Computing-Bootcamp`

Click the **Fork** button (top right). Pick yourself as the destination namespace. You now have your own copy at:

`code.jlab.org/<your-username>/20260514-GSPDA-Computing-Bootcamp`

Step 6: clone, edit, commit, push

Type this exact command

```
# Configure git once (use your real name and email)
git config --global user.name "Your Name"
git config --global user.email "your-email@jlab.org"

cd /work/<your-group>/$USER
git clone git@code.jlab.org:<your-username>/20260514-GSPDA-Computing-
    Bootcamp.git
cd 20260514-GSPDA-Computing-Bootcamp
# Make a first save point that's yours
echo "$USER, first commit on $(date)" >> notes.md
git add notes.md
git commit -m "Initial notes from setup"
git push
```

Refresh fork's webpage at `code.jlab.org`. You should see your commit appear within seconds.

Mid-Part-2 quick check

Checkpoint

Everyone: refresh your fork's web page and show me your `Initial notes from setup` commit.

You should see it as the most recent activity in the repository.

Tour: what's in the baseline repo?

```
20260514-GSPDA-Computing-Bootcamp/
|-- src/ <-- Geant4 source code
| |-- DetectorConstruction.cc <-- TODO STUDENT (Part 4)
| |-- PrimaryGeneratorAction.cc <-- cosmic muons
| `-- ... (rest is boilerplate, you will not touch it)
|-- macros/
| |-- run_cosmic.mac <-- batch run macro
| `-- vis.mac <-- GUI settings
|-- analysis/
| |-- efficiency.py <-- PyROOT, TODO STUDENT (Part 3)
| `-- efficiency.cc <-- compiled, TODO STUDENT + 1 bug
|-- docs/
| |-- detector_menu.md <-- suggested central detectors
| `-- troubleshooting.md <-- if you get stuck, look here
|-- CMakeLists.txt <-- build recipe (you will not edit)
`-- setup.sh <-- sources the CVMFS modules
```

Step 7: build the muon telescope

Type this exact command

```
source setup.sh # loads ROOT and Geant4 modules

mkdir build && cd build
cmake ..
make -j4

# Interactive (GUI) mode first
./muon_telescope
```

In the Geant4 prompt:

```
/run/beamOn 100
```

You should see 100 muon tracks falling through the telescope. Some will pass through both trigger paddles; some will not.

Step 8: produce a ROOT file in batch mode

Now exit the GUI (`exit` at the Geant4 prompt) and run in batch mode to produce the data we'll analyze in Part 3:

Type this exact command

```
./muon_telescope macros/run_cosmic.mac
ls -lsatrh output.root
```

This runs 10^5 events in batch mode (no GUI), producing `output.root`. Expect ∼60 seconds runtime.

Quick inspection:

```
root -l output.root
> .ls # list contents
> events->Print() # see the TTree branches
> .q
```

Checkpoint 2 of 4

Checkpoint

Everyone: show me an `output.root` of more than 1 MB in your build directory.

And your first commit is visible on `code.jlab.org`.

Potential hurdle

For those done early: start reading through `src/DetectorConstruction.cc` in your editor and read the `TODO STUDENT` blocks to get a head start on Part 4.
For those still working: let me know if you need assistance.

Part 3
ROOT analysis: interpreted then compiled

What is a ROOT TTree?

A TTree is a **NTuple - columns of data**:

- One row per "booked" entry.
- Each column ("branch") holds one variable for that step, hit, event, or tracked data storing unit (can be an angle, a logical flag, an energy, or even a vector of quantities).
- Reading is fast because you can ask for only the columns you need.

Our events tree has 36 branches, including:
pri_theta, pri_phi,
trig_top, trig_bot,
trk_nhits_0, trk_nhits_3,
cen_edep_sum.

(one row = one muon event)

pri_theta	trig_top	cen_edep_sum
0.21	1	4.34
0.55	1	0
0.78	0	0
0.42	1	2.123
⋮	⋮	⋮

Test data sanity with a ROOT draw command:

```
events->Draw("cen_edep_sum:pri_theta",
"pri_theta<0.45 && cen_edep_sum >
1","*")
```

What we are about to compute: efficiency vs zenith

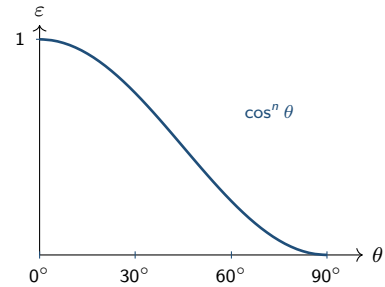
For each event in the tree, ask:

1. Did both trigger paddles fire?
   (= a muon passed through the telescope)
2. Did the central detector also fire?
   (= the muon was tagged by your detector)

$$\varepsilon(\theta) = \frac{N_{\text{trig AND central}}(\theta)}{N_{\text{trig}}(\theta)}$$

Fit with $\cos^n\theta$ over the range where statistics are decent ($\theta < 60°$).

NOTE: This cosine fit only works for the numerator or denominator - the ratio efficiency is a geometry/physics function with no known "best" fit function (but this can be used still)



Roughly. Your detector's shape will perturb this.

Step 9: complete simple analysis

Open `analysis/efficiency_simple.py`. The file is mostly written; you can improve at the four `TODO STUDENT` blocks (edep > 0.05 and "E1" draw are useful)

```
import ROOT
c = ROOT.TCanvas("c", "Canvas", 800, 600)
f = ROOT.TFile("../build/output.root")
events = f.Get("events")
h_denom = ROOT.TH1F("denom", ";theta [rad];events", 30, 0, 1.57)
h_numer = ROOT.TH1F("numer", ";theta [rad];events", 30, 0, 1.57)

# TODO STUDENT (1): loop over events
for ev in events:
    if ev.trig_top and ev.trig_bot: # TODO (2): trigger gate
        h_denom.Fill(ev.pri_theta)
        if ev.cen_edep_sum > 0: # TODO (3): central gate
            h_numer.Fill(ev.pri_theta)

h_numer.Divide(h_denom) # TODO (4): efficiency
h_numer.Draw()
fit = ROOT.TF1("f", "[0]*cos(x)^[1]", 0, 1.0)
fit.SetParameters(1.0, 2.0)
h_numer.Fit(fit, "R")
c.SaveAs("efficiency_simple.png")
```

Step 10: run it, save a PNG, commit

Type this exact command

```
cd analysis
python3 efficiency_simple.py
ls efficiency_simple.png
display efficiency_simple.png
git add efficiency_simple.py
git commit -m "PyROOT efficiency analysis"
git push
```

Refresh your fork's webpage on `code.jlab.org`: you should see `efficiency_simple.py` in the file listing.

Next: try the pre-written efficiency.py example (more robust algorithm, or debug why efficiency_simple.py is inadequate)

Now the same analysis in compiled C++

Why bother?

- Compiled C++ ROOT is what you normally would like to submit to the JLab batch farm to run billions of events with more efficiency.
- It runs $10\times$ faster than the interpreted Python (usually).
- The vast majority of legacy nuclear-physics analysis code at JLab looks like this.

What changes:

- Same algorithm.
- Different syntax (C++ instead of Python).
- Build step before running.
- One file: `analysis/efficiency.cc`.

Key idea

The lesson here is not C++ syntax. It is the recognition that the *same algorithm* runs as a script and as a compiled binary. The binary form is what scales.

Step 11: compile `efficiency.cc`

Type this exact command

```
cd ../analysis # (you should already be here)
bash
g++ -o efficiency efficiency.cc $(root-config --cflags --libs)
./efficiency
display efficiency_cpp.png
```

Potential hurdle

Heads up: the supplied `efficiency.cc` contains *exactly one* compile error that we deliberately left in. Read the compiler's output carefully — it will tell you the file and line number. Fix it before continuing.

This is on purpose. It is the first compiler error you will encounter today, and we want you to encounter it with helpers in the room.

Another note: switching `main()` to a "named macro" approach lets you run the .cc script using "root -l -b -q efficiency.cc"

Checkpoint 3 of 4

Checkpoint

Show me both PNGs:
`efficiency.png` (from PyROOT)
`efficiency_cpp.png` (from compiled C++)
They should look identical.

Bonus question to ask yourself: *Why* do they look identical? What guarantees that? (Answer: same input file, same algorithm, same fitter. Different language doesn't change the math.)

Part 4
Your detector, branch, PR

Pick your central detector

Open `docs/detector_menu.md` for the recommended list. A few to think about:

Material/shape	What you'll see
NaI cube, 5 cm	Decent scintillator, moderate efficiency
LYSO cube, 3 cm	Imaging-group flavor; high density
Plastic scintillator, 8 cm	Big, low-Z, high efficiency
Lead absorber, 1 cm slab	Bad detector! (Good physics lesson.)
Liquid Ar, 8 cm cube	Noble-liquid TPC starter

Or invent your own as long as it fits in a 10 cm cube envelope.

Step 12: branch, edit, rebuild

Type this exact command

```
cd /work/<group>/$USER/20260514-GSPDA-Computing-Bootcamp
git checkout -b my-detector
# Edit src/DetectorConstruction.cc -- two TODO STUDENT blocks
# (use your favorite editor: nano, vim, or VSCode remote)

# Rebuild
cd build && make -j4 && cd ..
# Rerun the simulation
./build/muon_telescope macros/run_cosmic.mac
# Rerun the analysis
cd analysis
python3 efficiency.py
./efficiency
```

Your new `efficiency.png` now reflects *your* detector.

Step 13: open a pull request, review, merge

Commit and push your changes:

Type this exact command

```
git add -A
git commit -m "Central detector: <your material> <your shape>"
git push -u origin my-detector
```

On `code.jlab.org` in your fork:

1. Click the prompt to "Create merge request" from your branch.
2. Set the target to `main` in your own fork.
3. Read the diff. Confirm your changes are sensible.
4. Approve and merge.

Refresh `main` on your fork. Your changes are now there.

Checkpoint 4 of 4 — the final one

Checkpoint

Everyone: show me a green-merged PR in your fork.
The `main` branch of your fork should contain your `efficiency.png` from *your* detector.
You are now ready for show-and-tell.

Practice and show/tell

Where to go from here

For ROOT depth: The ROOT Primer, `root.cern/primer/`, covers everything in Part 3 plus much more.

For Geant4 depth: The Book for Application Developers, `geant4.web.cern.ch`.

For git depth: *Pro Git* (free online), `git-scm.com/book/en/v2`.

For batch submission (slurm, swif2): Covered in tomorrow's parallel session.

For help when you're stuck: `helpdesk@jlab.org`; the SciComp portal at `scicomp.jlab.org`; me at `cameronc@jlab.org`.

Key idea

Everything you did today is reproducible from the commits in your fork. Open a new terminal tomorrow, `git clone` your fork on a different machine, and you can pick up exactly where you left off.

Thank you

Questions? cameronc@jlab.org

Radiation Detector & Imaging Group

Jefferson Lab