# SHOE management

Yunsheng Dong

FOOT analysis/software meeting

03/12/2024

# SHOE branches at present

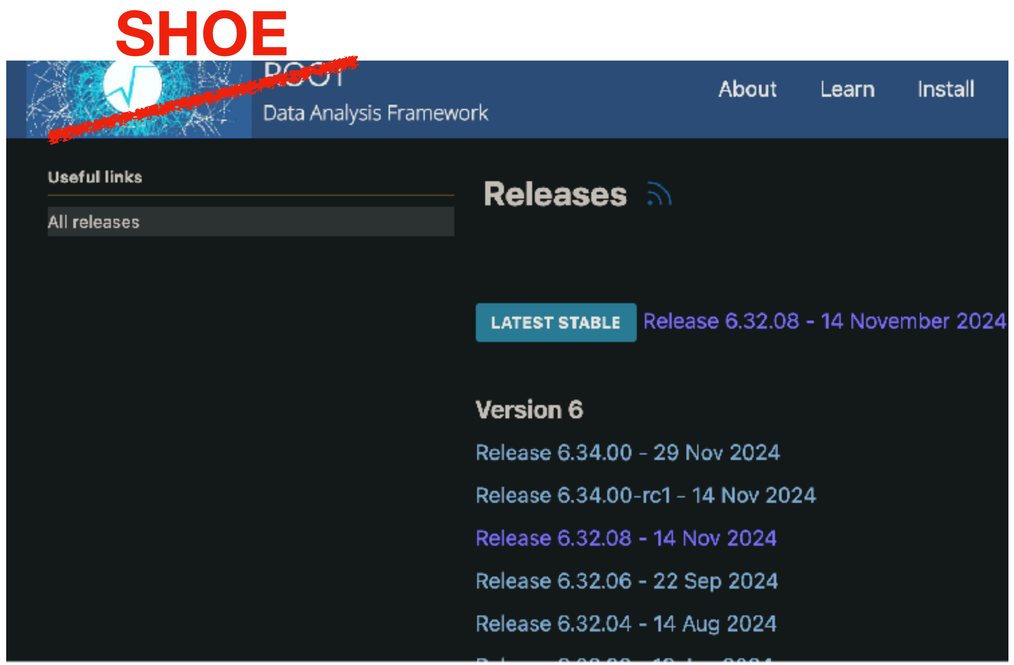

**SHOE branches**

- newgeom_v1.0: used as “development” branch where all the FOOT developers (~almost all the foot people) can push and update everything in the code
- Master: updated with newgeom_v1.0 after few months. Here only few people can push (protected branch were only 4-5 people can push)
- Other branches for dedicated studies
  (e.g.: GlobalGF_dev for global tracking development)

**Some drawbacks:**

- Difficulty to do long term analysis (e.g.: cross sections) staying updated with newgeom_v1.0
- Change of methods in some newgeom_v1.0 classes can have an impact on all the personal analysis macros etc., but the tracking of the changes is not easy only with “git log” or similar
- Master branch not used at all (at least for foot people, maybe some student used the master branch for a bachelor thesis)
- The FOOT.geo files can be a mess if whoever can modify them, we need to find a way to protect (and check) this kind of files

# Proposal of versioning




- Delete the master branch and use only the newgeom_v1.0, (or delete newgeom_v1.0 and use the master branch with the permission to push to all the foot developers)
- Use a release version of shoe.
  Maybe every ~6 months, with a release number based on the date, the development branch is copied and pushed as a separated branch of shoe
- Analysis people can refer to a specific version of the code
  If needed, they can pull only specific files for their analysis work.
- Add in the wiki a release wiki page in which all the main improvements/developments are listed

# Git hook

```
#!/bin/bash
# Define protected file patterns
PROTECTED_FILES="Reconstruction/geomaps/.*/FOOT.*geo"

# Define the list of allowed usernames
ALLOWED_USERS=("maintainer1" "maintainer2") # Replace with actual usernames

# Read incoming commits
while read oldrev newrev refname; do
  # Get the list of changed files
  CHANGED_FILES=$(git diff-tree --no-commit-id --name-only -r $newrev)

  # Get the committer's username
  COMMITTER_USERNAME=$(git show -s --format='%an' $newrev)

  for file in $CHANGED_FILES; do
    if [[ $file =~ $PROTECTED_FILES ]]; then
      echo "Protected file '$file' was modified."

      # Check if the committer is in the allowed list
      if [[ ! " ${ALLOWED_USERS[@]} " =~ " ${COMMITTER_USERNAME} " ]]; then
        echo "Error: User '$COMMITTER_USERNAME' is not authorized to modify '$file'."
        echo "Contact a maintainer for authorization."
        exit 1
      else
        echo "User '$COMMITTER_USERNAME' is authorized to modify '$file'."
      fi
    fi
  done
done

exit 0
```

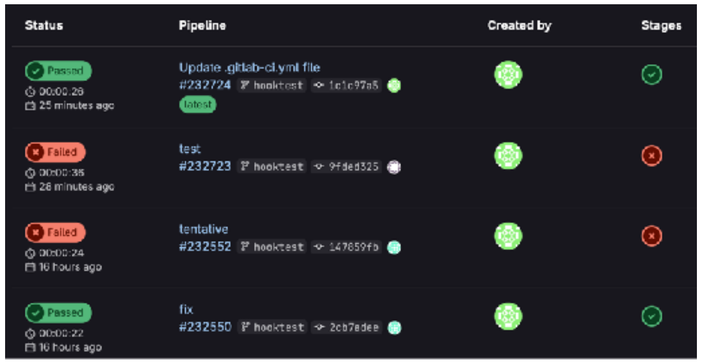
**Git does not provide a built-in mechanism to set different push permissions for specific files (e.g.: the foot.geo files)**

Different alternatives:
**Use of git hook pre-receive:**

- Set a pre-receive hook on baltig
- To be tested
- Prevent the push of given files from non authorised people
- Need to ask to info baltig service if they can allow us to add this hook
- We need to teach people what to do if they committed a change to a protected file and cannot push

# Git pipeline

```yaml
image: ubuntu:20.04

before_script:
  - hwclock --hctosys
  - apt-get update # Update the package list
  - apt-get install -y git util-linux

stages:          # List of stages for jobs, and their order of execution
  - check_files   # New stage to check for protected files
  - deploy

check_protected_files:
  stage: check_files
  script:
    - echo "Checking if protected files are modified..."
    - |
      # Get the list of files changed in the current commit
      CHANGED_FILES=$(git diff --name-only $CI_COMMIT_BEFORE_SHA $CI_COMMIT_SHA)

      # Define the protected files pattern (modify as per your requirement)
      PROTECTED_FILES="Reconstruction/geomaps/.*/FOOT.*geo"

      # Check if any of the changed files match the protected pattern
      for file in $CHANGED_FILES; do
        if [[ $file =~ $PROTECTED_FILES ]]; then
          # If the file is protected, check if the user is authorized
          echo "File $file is protected."

          # Define the list of allowed users
          ALLOWED_USERS=("yunsheng" "maintainer2")  # Replace with actual GitLab usernames

          # Check if the user is authorized (using CI_COMMITTER_USERNAME)
          if [[ ! " ${ALLOWED_USERS[@]} " =~ " ${CI_COMMITTER_USERNAME} " ]]; then
            echo "Error: User ${CI_COMMITTER_USERNAME} is not authorized to modify '$file'."
            exit 1
          else
            echo "User ${CI_COMMITTER_USERNAME} is authorized to modify '$file'."
          fi
        fi
      done
  only:
    - hooktest  # Restrict to the main branch or any other protected branch
```



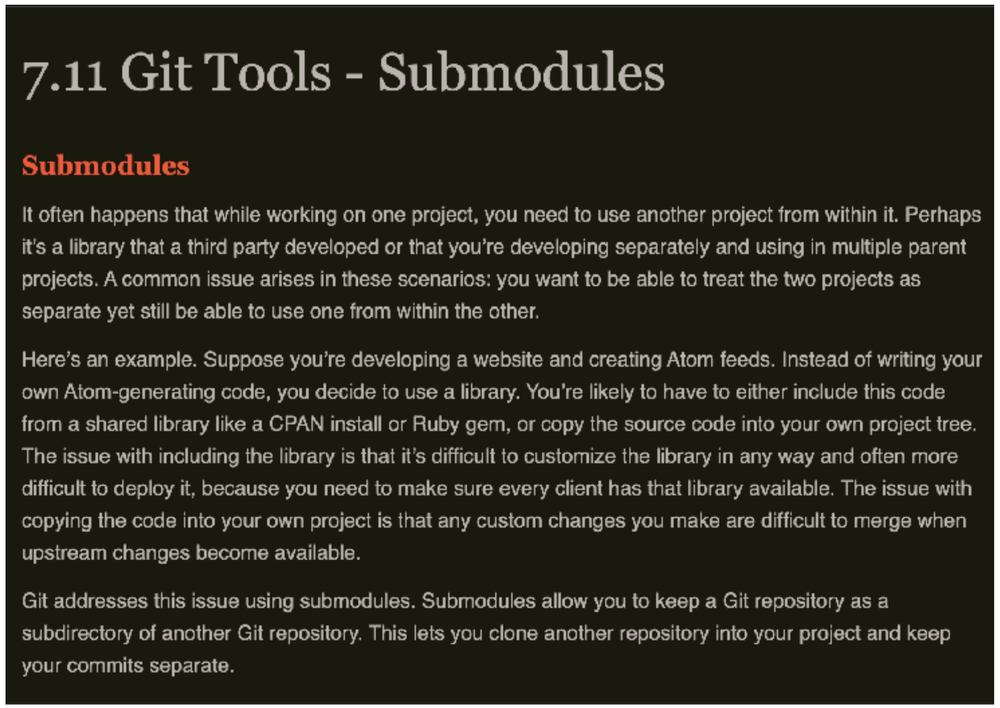

## What is a CICD pipeline?

A pipeline is the lead component of continuous integration, delivery, and deployment. It drives software development through building, testing and deploying code in stages. Pipelines are comprised of jobs, which define what will be done, such as compiling or testing code, as well as stages that spell out when to run the jobs. An example would be running tests after stages that compile the code.

A CI/CD pipeline automates steps in the SDLC such as builds, tests, and deployments. When a team takes advantage of automated pipelines, they simplify the handoff process and decrease the chance of human error, creating faster iterations and better quality code. Everyone can see where code is in the process and identify problems long before they make it to production.

- Possibility to add a pipeline to check each push and send a pipeline failed message when a non authorised person changed specific files
- The push is done, but we are aware of the push
- We can add and modify pipelines directly on gitlab, already tested

# Git submodules



## 7.11 Git Tools - Submodules

### Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here's an example. Suppose you're developing a website and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You're likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it's difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with copying the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

- Possibility to store all the geo files in a submodule, which is a different git repository added in shoe
- The new repository can be set as protected and only selected people can push there
- The submodule folder can be added somewhere inside shoe, but all the submodule file should be in the same folder/subfolder
- The usual commands as git clone, git pull need to be changed
- It's more suitable for something like genfit, or slippery

# TAGcluster:GetPosition and GetPositionG

```
protected:
   TVector3          fPosition1;  ///< position of the cluster in plane frame / measured position in FOOT frame
   TVector3          fPosError1;  ///< position's errors of the cluster in plane frame / measured position's error in FOOT frame
   TVector3          fPosition2;  ///< position of the clus in tracker frame / / fitted position in FOOT frame
   TVector3          fPosError2;  ///< position's errors of the clus in tracker frame /  fitted position's error in FOOT frame
```

```
//! Get position in local frame
virtual const TVector3&  GetPosition()      const { return fPosition1;     }
//! Get position error in local frame
virtual const TVector3&  GetPosError()      const { return fPosError1;     }
//! Get position in global tracker frame
virtual const TVector3&  GetPositionG()     const { return fPosition2;     }
//! Get position in global tracker frame
virtual const TVector3&  GetPosErrorG()     const { return fPosError2;     }
```

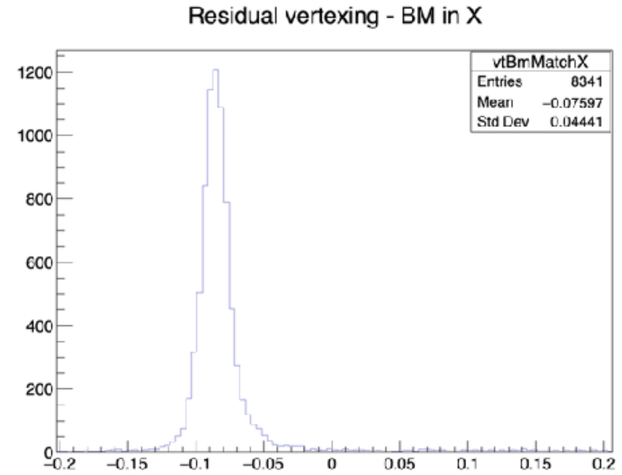
- The definition of fposition1 and fposition2 is not very clear and not unique
- If global tracking is not activated, fposition1 should be the position of the hit on the plane, fposition2 should be the position of the hit in the detector frame
- If global tracking is activated, the variables are overwritten
- Not sure if the convention if maintained for all the detectors

**Proposal: Add one/two more variables for the global tracking fitted position and change the variable names**

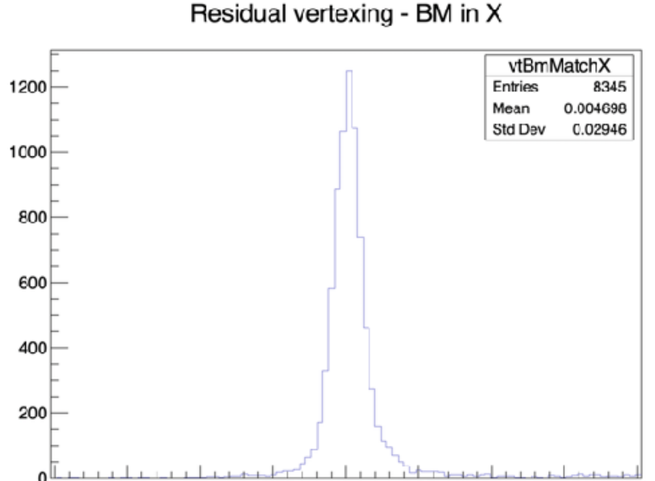
- We should change all the analysis/macros accordingly
- Probably the back compatibility is lost
  (but hey we have versioning!)

# TAGgeoTrafo:Intersection

```
void TAVTactBaseNtuTrack::FillBmHistogramm(TVector3 bmTrackPos)
{
   bmTrackPos  = fpFootGeo->FromBMLocalToGlobal(bmTrackPos);
   fpHisBmBeamProf->Fill(bmTrackPos.X(), bmTrackPos.Y());

   Float_t posZtg = fpFootGeo->FromTGLocalToGlobal(TVector3(0,0,0)).Z();
   posZtg = fpFootGeo->FromGlobalToVTLocal(TVector3(0, 0, posZtg)).Z();

   for (Int_t i = 0; i < GetTracksN(); ++i) {
    TAGbaseTrack* track  = GetTrack(i);
     TVector3      origin = track->Intersection(posZtg);

    origin  = fpFootGeo->FromVTLocalToGlobal(origin);


    TVector3 res = origin - bmTrackPos;
    fpHisVtxResX->Fill(origin.X(), bmTrackPos.X());
    fpHisVtxResY->Fill(origin.Y(), bmTrackPos.Y());
   }
```




```
void TAVTactBaseNtuTrack::FillBmHistogramm(TVector3 bmTrackPos)
{
   bmTrackPos  = fpFootGeo->FromBMLocalToGlobal(bmTrackPos);
   fpHisBmBeamProf->Fill(bmTrackPos.X(), bmTrackPos.Y());



   for (Int_t i = 0; i < GetTracksN(); ++i) {
    TAGbaseTrack* track  = GetTrack(i);

    TVector3      origin = fpFootGeo->Intersection(fpFootGeo->VecFromVTLocalToGlobal
    (track->GetSlopeZ(), fpFootGeo->FromVTLocalToGlobal(track->GetOrigin()),
    fpFootGeo->GetTGCenter().Z());
    TVector3 res = origin - bmTrackPos;
    fpHisVtxResX->Fill(origin.X(), bmTrackPos.X());
    fpHisVtxResY->Fill(origin.Y(), bmTrackPos.Y());
   }
}
```




- With the alignment procedure the detector sys of ref can be tilted!
- TAGgeoTrafo:Intersection should be used carefully: if the detector is rotated, the finalZ is not simply the detector center Z position, we need to use intersection changing the track slope, origin and the finalZ parameters in the final position sys of ref.
- Shoe is full of TAGgeoTrafo::Intersection calls. All the intersection calls has been checked and corrected in cnao24datataking few days ago, but maybe something is missed (variables filled with intersection haven't been checked everywhere)
- **Developers are invited to check the Intersection calls in their analysis/detector codes**
- **Be aware of detector rotations when you need to extrapolate with Intersection**
- **New method with Intersection(slope, origin, startingsysofref, finalz, finalsysofref) can added to easy the use of this method**

# Other

- Add the commit stash (and shoe branch/version) into runinfo

- Merge cnao24datataking into newgeom: if no objections, it will be done at the end of this meeting

- If we decide to change TAGcluster::fposition1/2 etc., the back compatibility of the code is probably compromised and we need to revise different part of SHOE, so this is a good occasion to introduce all the “heavy” changes
  eg.:
  -Should we change TABMtrack::GetSlope into GetSlopeZ?
  -Should we change TAMCntuRegion::fCharge from double to int?
  -In order to retrieve the particle block data from the crossings or TAMChit one need to add a -1
  (e.g.: mcNtuPart->GetTrack(cross->GetTrackIdx()-1) ),
  this is not valid if one need to retrieve the particle block data from reconstructed quantities
  (e.g.: TAMCpart* mcpart=mcNtuPart->GetTrack(vtcluster->GetMcTrackIdx(k)) )
  Should we change this?

- Other proposal/wishes?