GitHub Instructions

Theo Smith

August 2025

1 Git Overview

Git is a version control system that tracks changes in code or other files over time. It allows you to save different versions of a project, revert to earlier states, and develop new features in parallel using branches. Git is distributed: every copy of a repository contains the complete project history, so you can work fully offline.

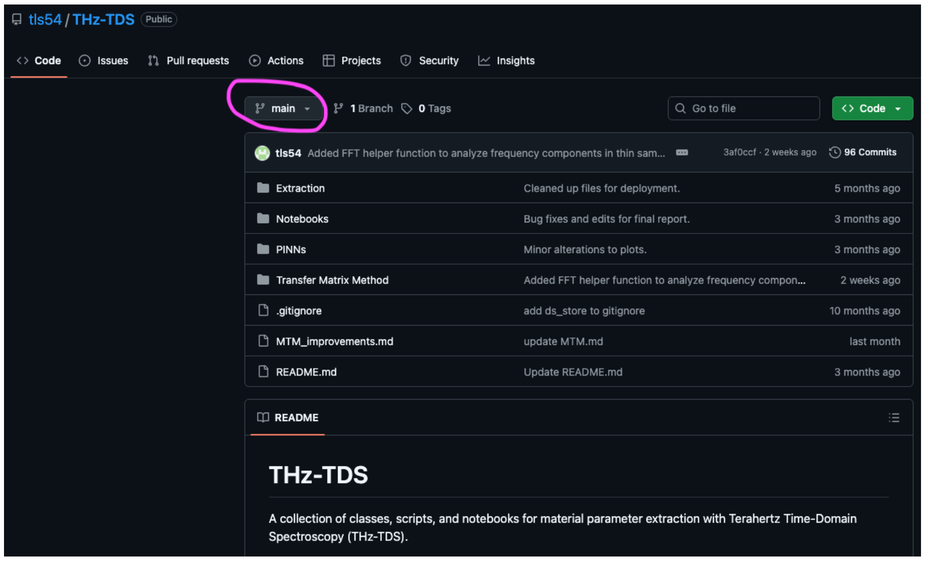
GitHub is an online platform that hosts Git repositories and makes collaboration easier. By pushing your local Git repository to GitHub, you create a remote backup, can share with teammates, and make use of tools such as pull requests, issues, and code reviews. Git itself works entirely on your computer; GitHub simply provides a central place for collaboration.

We have our projects stored remotely at the following two urls:
`https://github.com/tls54/THz-TDS`.
`https://github.com/tls54/THz-TD-CNN`.

There are local versions on the on the modelling computer in the THz lab.

1.1 retrieving and updating code

The most up-to-date stable code is kept in the main branch of the GitHub repository (sometimes historically called master). Figure 1 shows where you can check which branch you are currently on. Branches allow different people to safely make modifications in parallel. These can later be merged, where Git will automatically check for conflicts.



tls54 Added FFT helper function to analyze frequency components in thin sam… 3af0ccf · 2 weeks ago 96 Commits

Extraction	Cleaned up files for deployment.	5 months ago
Notebooks	Bug fixes and edits for final report.	3 months ago
PINNs	Minor alterations to plots.	3 months ago
Transfer Matrix Method	Added FFT helper function to analyze frequency compon…	2 weeks ago
.gitignore	add ds_store to gitignore	10 months ago
MTM_improvements.md	update MTM.md	last month
README.md	Update README.md	3 months ago

THz-TDS

A collection of classes, scripts, and notebooks for material parameter extraction with Terahertz Time-Domain Spectroscopy (THz-TDS).

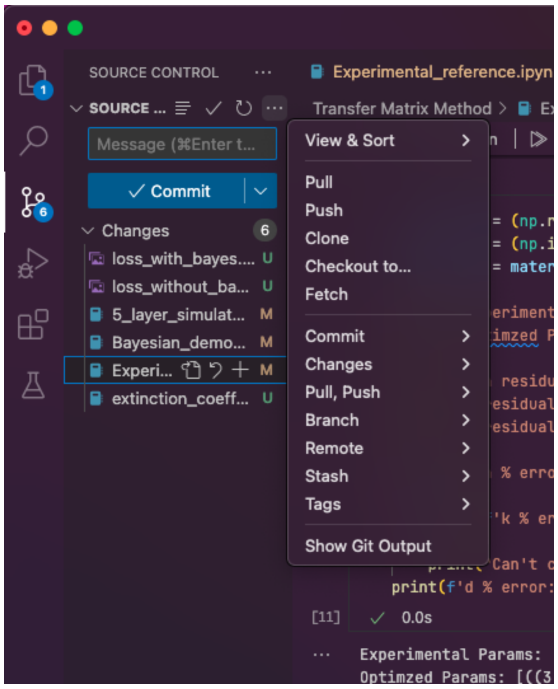
Figure 1: Pink circle shows the branch you are looking at in the Github repository.

To copy (clone) a repository to your computer, run the command below in the terminal (after using cd to move into the folder where you want the project stored). This sets up a local Git repository with the full project history:

```
git clone https://github.com/tls54/THz-TD-CNN.git
```

Listing 1: git clone

If changes are made on one device (for example, Theo's Mac) and pushed to GitHub, another device with the same repository can be updated by running git pull in the terminal (or using VSCode's interface as shown in Figure 2). This fetches and applies the latest version from the remote repository.



```
print(f'd % error:
[11] ✓ 0.0s
Experimental Params:
Optimzed Params: [((3
```

Figure 2: Git pull command in VSCode

1.2 Pushing and Committing code

All edits are first made in your local copy of the repository (on your computer). To record these changes, you use commits. A commit takes a snapshot of your current work along with a short descriptive message.

Once committed, the changes can be pushed to GitHub, which updates the remote copy of your branch. In general, it is best practice to make significant changes on a new branch rather than directly on main. This prevents accidental breakages and allows testing before merging.

You can create a new branch in VSCode (or the terminal). VSCode will then prompt you to "publish" the branch, which creates the same branch on GitHub. After making your changes, you commit them locally, and then push them to GitHub. VSCode simplifies this workflow: the blue Commit button stages and commits your changes locally, then offers to sync them with the remote branch (which runs git push in the background).

1.3 Merging

Once a feature has been added and tested, its branch can be merged into the main branch. The safest way to do this is via a pull request on GitHub. GitHub will automatically check for compatibility (merge conflicts) and only allow the merge if they can be resolved.

After the merge, your local copy of main will be behind the remote. To update, simply pull the new main branch. At this point you can also delete the old feature branch locally and on GitHub. In the rare case of serious problems, you can always re-clone the repository from GitHub.

2 Bayesian & Gradient Extractors

The extractors are held as classes (makes the extractor an object). Gradient extractor is held in AdamExtractor.py and the Bayesian is held in BayesianExtractor.py. These are both held in the folder Matrix_methods, this also contains all of the other code that these objects rely on to be used. Full working examples are held in the notebooks within the THz-TDS Github Repository at `https://github.com/tls54/THz-TDS`.

To use the objects have the Matrix_methods folder on the same level as the notebook you are working in. Import the extractor objects:

```python
import torch
from Matrix_methods.BayesianExtractor import
    BayesianLayeredExtractor
from Matrix_methods.AdamExtractor import LayeredExtractor
from Matrix_methods.Simulate import simulate_parallel
# Optional for when the refrence needs to be simulated
from Matrix_methods.Simulate import simulate_reference
```

Listing 2: Importing extractors

These objects require data to be held in PyTorch tensors. Arrays and lists can be converted to a tensor using the following line:

```python
# Convert refence pulse to tensor
reference_pulse = torch.tensor(reference_pulse, dtype=torch.
    float32)
```

Listing 3: Converting an array or list to a tensor

In order to be able to plot tensors they must be detached and moved back to the CPU. This can be done in the plotting line to make the move temporary.

```
plt.figure(figsize=(12,4))
plt.plot(t_axis, reference_pulse.detach().cpu().numoy(),
    label='ref pulse')
plt.xlabel('Time, t [ps]')
plt.legend()
plt.show()
```

Listing 4: Converting an array or list to a tensor

This plots the reference pulse below:

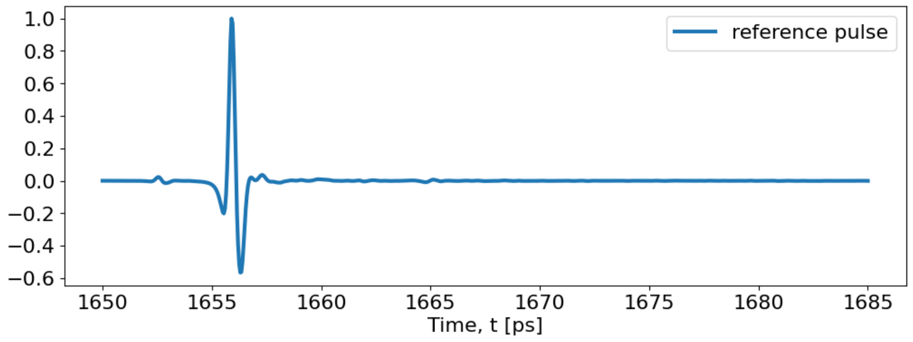


Figure 3: Reference Pulse plotted from Torch tensor using code in Listing 3

This also needs to be done for the sample pulse. This can be done in the same way as the refrence pulse.

Once you have these two core pieces of data, the extraction can be performed. The core simulation engine behind this method is Jeff's transfer matrix method code, in order to made a forward pass (prediction) we need the following information:

- Material parameters: (initial value provided by you and then provided by the model after).
- deltat: the time resolution of your reference and sample pulse (these should match).
- optimization mask: (optional) allows you to select which material parameters are being changed and what are set values.
- optimization boundaries: (only for bayes) defines a strict search space for the Bayesian extraction object.

```python
# set initial guesses for Bayes optimization
layers_init = [(3.17-0.0015j, 95e-6)]
optimization_bounds = [0.3, 0.01, 9.5e-6]

# Set optimization mask
optimization_mask = [(True, True, True)]
# Initialize extractor for the data
Bayes_extractor = BayesianLayeredExtractor(reference_pulse,
    experimental_pulse, deltat, layers_init,
    optimization_mask, optimization_bounds=
    optimization_bounds)

# Run the extraction loop and returns parameters
bayes_params = Bayes_extractor.bayesian_optimization(n_calls
    =50)
```

Listing 5: Code to initialize and run bayesian extractor

These parameters can be passed to the simulate_parallel function from Simulate.py (The forward pass code) to plot the pulse from the new parameters.

```python
1  # Run forward pass with Jeff's model
2  bayes_pulse = simulate_parallel(reference_pulse,
       bayes_params, deltat, 0)[1].detach().cpu().numpy()[:L]
3  # Print the parameters
4  print(bayes_params)
5  # plot the pulse
6  plt.figure(figsize=(12,4))
7  plt.plot(t_axis, bayes_pulse, label='Bayes pulse', linestyle
       ='--')
8  plt.xlabel('Time [ps]')
9  plt.xlim([1655, 1675])
10 plt.legend()
11 plt.show()
```

Listing 6: Code to reconstruct and plot pulse form Bayesian parameters

The reconstructed pulses aren't used for anything so I tend to detach them and move them to the CPU permanently. Because of upscaling in the frequency domain during the forward pass, we tend to truncate the pulse back down to its original size. This is done with the '[:L]' at the end of line 2 in Listing 6.

The results from here can be passed to the gradient (adam) based optimizer. This runs much faster.

```python
# Initialize adam optimizer object
grad_optimizer = LayeredExtractor(reference_pulse,
    experimental_pulse, deltat, bayes_params,
    optimization_mask, lr=0.0001)
# Run optimization for set iterations, ‘updates’ provides a
    printout of progress
optim_params = grad_optimizer.optimize(num_iterations=300,
    updates=50)

optim_pulse = simulate_parallel(reference_pulse,
    optim_params, deltat, 0)[1].detach().cpu().numpy()[:L]

print(optim_params)
plt.figure(figsize=(12,4))
plt.plot(t_axis, reference_pulse, label='Reference Pulse',
    alpha=0.5)
plt.plot(t_axis, experimental_pulse.detach().cpu().numpy(),
    label='Experimental Pulse')
plt.plot(t_axis, optim_pulse, label='Fine Tuned Pulse',
    alpha=0.7, linestyle='--')
plt.xlabel('Time, t [ps]')
plt.legend()
plt.show()
```

Listing 7: Code to run adam optimizer then reconstruct and plot pulse form Bayesian parameters

2.1 Material parameter storage

The material parameters are held in the following way:

[[(n1+1j*k1), d1], [(n2+1j*k2), d2], ...]

This is a nested set of lists with a complex float type for the refractive index and a float stored in meters for the thickness. n1, k1, d1.....(ect) are the free parameters that are being modified by the optimizers. 1j is the python equivalent of ‘i’ for complex numbers.

3 CNNs

The machine learning models are designed to work as trained models and fit into material classification workflows. It is rare that highly accurate measurements are achievable from machine learning alone. For example, it is unlikely that a neural network will ever be as precise as the gradient optimizer at calculating a material parameter. What they will excel at is being able to quickly make predictions on non-ideal pulses and identifying candidate values for material parameters allowing for better convergence of the iterative solvers. The foundational models are trained with synthetic datasets created using Simulate.py (Jeff's model). The generation script creates a dataset of N pulses each of which is 1024 points in length. Currently we are exploring what models are capable of, once the proof of concept (PoC) is completed for a task, more complex datasets can be created and used and trained on to make the model fit for real world application.

It is important to note that the datasets are usually 100-600Mb meaning they aren't uploaded to GitHub. All files ending in .pt (the file format we use for datasets) are in the .gitignore file, this means the git tracking ignores them when a commit is made. The File to generate data is included and datasets should be created for each new device being used.

3.1 Classification models

The Classifier models comprise a CNN processing network that is attached to a shallow classification head. The CNN learns to extract features of the pulse while the classification head takes these features and learns how they relate to the classes (in this case the number of layers in a sample). We have a model trained to identify up to 3 layers and does so with roughly 96% accuracy. We have shown also that both training on clean pulses and fine-tuning on noisy ones works as well as training on data with a mix of noise levels. This task is relatively simple and has been achieved with a light-weight model without the need for a complicated architecture. Its current drawback is that it is trained on pulses generated using a single reference and only takes the sample pulse. This means the model will likely fail on samples produced by a different reference.

There are two solutions to this issue, depending on the intended application:

- Train a model with a single reference pulse but with the pulses augmented to account for normal changes in environment and laser drift.
- Incorporate the reference pulse data into the training:
  - A 2-d CNN that passes over both the sample and the reference.
  - A Siamese architecture with two CNN nets one for the reference and one for the sample. These could then be connected by the classification head that learns to relate the two before making a classification.

Option one would be ideal for a turn-key system using where the pulse profile is pretty consistent. A set of models could be trained on a defined set of references such as air, purged box ...(ect) with augmentations made for minor changes in the system. These can also be fine-tuned on the noise profile of the system as well.

Option two would be much more suitable for a wide scale deployment, being able to learn the relationship between the reference and the sample and extract the layer numbers form this.

The model also needs to be tested on a higher number of layers, perhaps up to 5 layers.

3.2 Regression models

The regression models aim to take a sample pulse for a known number of layers and then predict the material parameters of the pulse. The main point of these models will be to provide a quality initial value for either the Bayesian optimization routine or the gradient optimization routine from no prior knowledge of the sample. This is a more complicated task and requires a larger network and a more careful training routine. This is because of how small the changes some material parameters have on the output pulse. This is currently causing a lack of precision in the current models. Ensuring good feature extraction is also tricky, using CNNs we need filters that can extract features at varying scales. We are currently working on a proof of concept model with pulses from 3 layers. We have an output of 9 parameters. Due to the difference in scales, we normalise the material parameters within their own domains. So n, k and d are separately scaled between 0 and 1 giving $\tilde{n_i}, \tilde{k_i} and \tilde{d_i}$.

$$[\tilde{n_1}, \tilde{k_1}, \tilde{d_1}, \tilde{n_2}, \tilde{k_2}, \tilde{d_2}, \tilde{n_3}, \tilde{k_3}, \tilde{d_3}] \tag{1}$$

The models are currently being trained with a supervised approach, this does not allow the model to see or be aware of the underlying physics of the system. Training a physics informed system by using the reconstruction error from the predicted parameters could be a solution. Our physical model (Jeff's code) is fully differentiable (see the gradient optimizer). This can either be used as the sole loss function or a hybrid approach with a weighting. For a 3 layered sample we have 9 parameters 'u', meaning the network outputs a 9 dimensional vector '$\hat{u}$'.

$$\mathcal{L} = \frac{1}{9}\sum_{i=1}^{9}(\hat{u}_i - u_i)^2 + \frac{1}{N}\lambda\sum_{j=1}^{N}(\hat{y_j}(\hat{u}) - y_j)^2 \tag{2}$$

This is a hybrid loss function for training using both a supervised loss contribution and a physics informed loss. The physics informed loss takes the parameters from the model, $\hat{u}$ and makes a prediction pulse using the differentiable transfer matrix model $\hat{y_j}(\hat{u})$ loss is then computed with the real pulse y_j with N being the number of points in the time domain. λ is a scaling function as MSE of the scaled vectors from the NN should range from 1 (worst case) to 10^{-4} for a later stages of training. The Physics informed loss (from a normalised reference) will be 10^{-2} to 10^{-6}. λ can be used to balance the weighting of these contributions.

Improved accuracy can also be found by separating the output heads by material parameter type. This could be because the material parameters have different impacts on the pulse shape. Having separate prediction heads for each parameter type could allow for higher quality predictions and is worth exploring.

3.3 Alternate Architectures

The current models purely use CNNs for the pulse processing stage. Alternate architectures such as RNNs, ResNets, LSTM or Transformers may be better at extracting global level features for this task.