TOMOPT: Differential Muon Tomography Optimisation

Tommaso Dorigo & Giles Strong

MODE Workshop on Differentiable Programming,

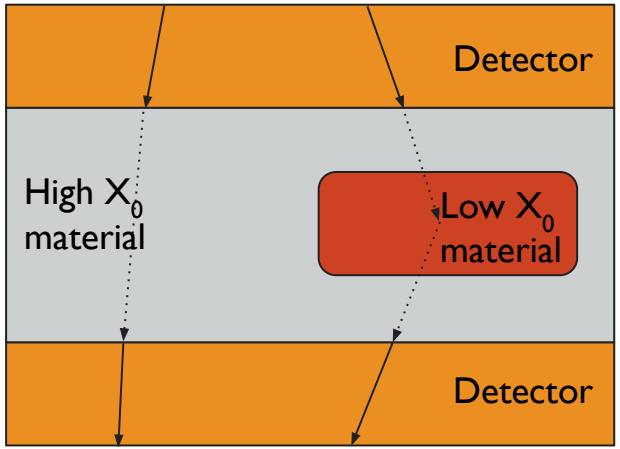
Online - 06/09/21

INTRODUCTION

Quick intro to tomography and motivation of detector optimisation

TOMOGRAPHY VIA MULTIPLE SCATTERING

- Consider a volume with unknown composition
  - E.g. Shipping container, archeological site, nuclear waste, industrial machinery
  - Want to build 3D map of elemental composition of volume
- Cosmic muons scattered by volume according to radiation-length (X_0 [m]) of elements in material
  - Measure muons above and below volume
  - Kinematic changes provide info on material composition
- Tomography by absorption/transmission possible (but not covered here)



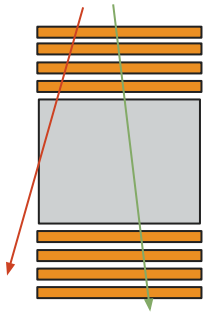
High X_0 = low scattering

Low X_0 = high scattering

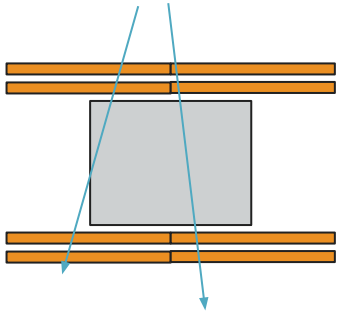
X_0 = average distance between scatterings

PROBLEM

- Each use-case likely to have a budget:
  - E.g. fiscal, heat, power, spatial, imaging time
- How should detectors be positioned to best function in each use case subject to constraints?
- Domain knowledge, experience, and intuition can help
  - But solutions likely to be based on heuristics and proxy objectives (e.g. lowest uncertainty on muon-path angles)



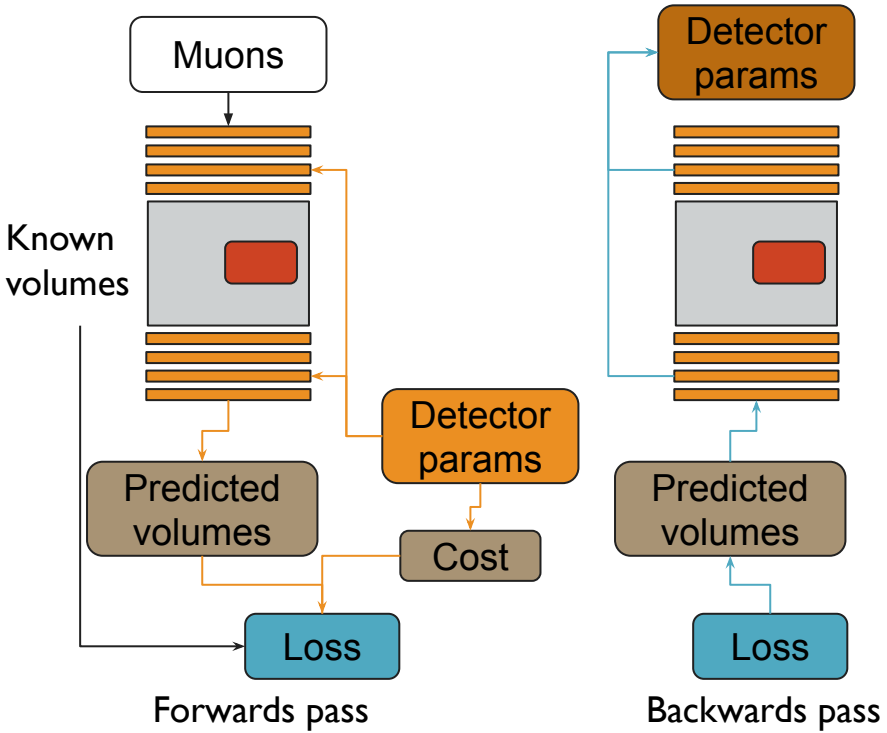
Example 1: Muons measured precisely but over a small area



Example 2: Muons measured less precisely but over a wider area

END-TO-END OPTIMISATION

- Consider instead simulating muon propagation and expressing the entire inference chain as a differentiable system
  - We can now compute the analytical effects of detector parameters (position, size, resolution, etc.) on system outputs
- Now express the desired task as a loss function
  - E.g. error on X_0 predictions, detector costs, time to achieve desired resolution
- We can now backpropagate the loss gradient to detector parameters and optimise via gradient descent
  - Just like a neural network



TOMOPT

Differential optimisation of muon-tomography detectors

TOMOPT

- Python package for differential optimisation of muon-tomography detectors
  - Modular design
  - PyTorch provides autodiff
- Currently developed within the MODE collaboration
  - First application of our optimisation pipeline proposed in MODE, 2021
  - Aim is an open-source package
  - Contributors and co-developers welcome (we aim to publish a paper, too!)
- Work so far is focussing on populating framework
  - Several simplifications made, and will be improved on later
  - Modular design means that each part can be changed independently
- Will provide complete simulation+optimisation cycle:
  - Muon generation & scattering simulation
  - Detector & volume simulation
  - Volume inference
  - Detector optimisation

SIMULATING MUON SCATTERING

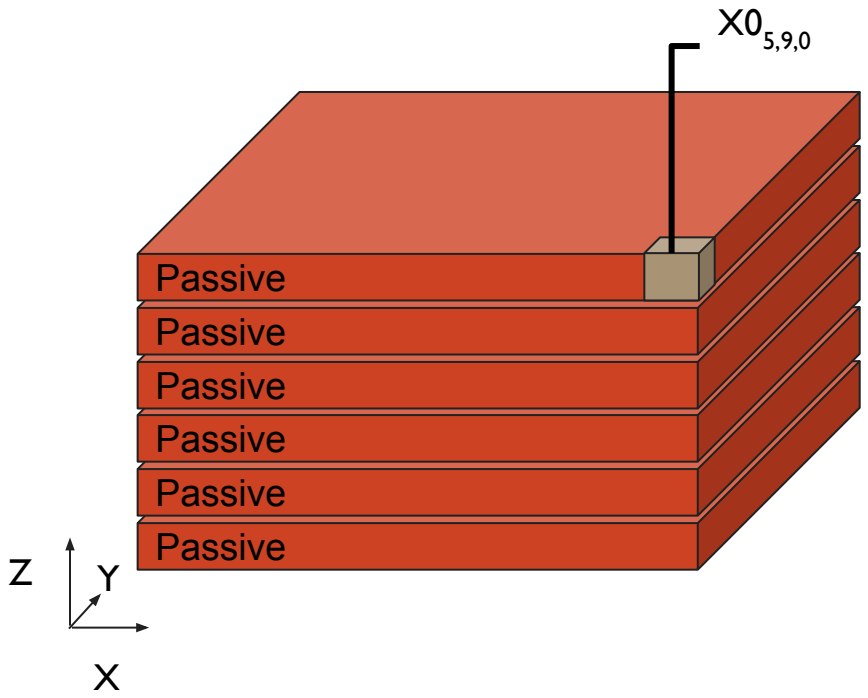
- Muons generated via simple model
- Simplified literature model for scattering
- Scattering = change in trajectory angle and xy displacement after propagating specified z distance in material
- Random terms mean scattering is non-deterministic

$$N_{\mathrm{rad-lengths}} = \delta z / \left(X_0 \times \cos\theta\right)$$
$$\theta_0 = \left(0.0136 / p_{\mathrm{muon}}\right) \times \sqrt{N_{\mathrm{rad-lengths}}}$$
$$\theta_{\mathrm{msc}} = \sqrt{2} \times \mathcal{N}(0,1) \times \theta_0$$
$$\phi_{\mathrm{msc}} = 2\pi \times \mathcal{U}(0,1)$$
$$\Delta h_{\mathrm{msc}} = \delta z \times \sin\theta_0 \times \left(\frac{\mathcal{N}(0,1)}{\sqrt{12}} + \frac{\mathcal{N}(0,1)}{2}\right)$$

$$\Delta x = \sqrt{2} \times \Delta h_{\mathrm{msc}} \times \cos\phi_{\mathrm{msc}} \times \cos\theta_x$$
$$\Delta y = \sqrt{2} \times \Delta h_{\mathrm{msc}} \times \sin\phi_{\mathrm{msc}} \times \cos\theta_y$$
$$\Delta x = \theta_{\mathrm{msc}} \times \cos\phi_{\mathrm{msc}}$$
$$\Delta y = \theta_{\mathrm{msc}} \times \sin\phi_{\mathrm{msc}}$$

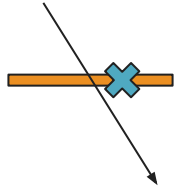
SIMULATING PASSIVE VOLUMES

- Discretise material to be imaged (passive volume) into layers in z
- Each layer further split into xy voxels
- Each voxel can be of different material
  - Inference task is to predict the X_0 of entire voxels

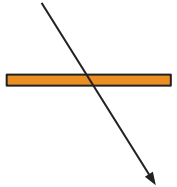


DETECTORS

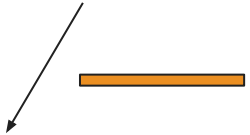
- Detectors record positions of muons:
  - With a certain probability (detector efficiency)
  - With a certain precision (detector resolution)
- In order to optimise the detectors, the output of the model must be differentiable w.r.t. the parameters
- This leads to several difficulties and restrictions
  - Primarily that detectors cannot be created/destroyed differentiably (discrete action)



Example 1: Muon passes through detector. Recorded hit located near muon's true location



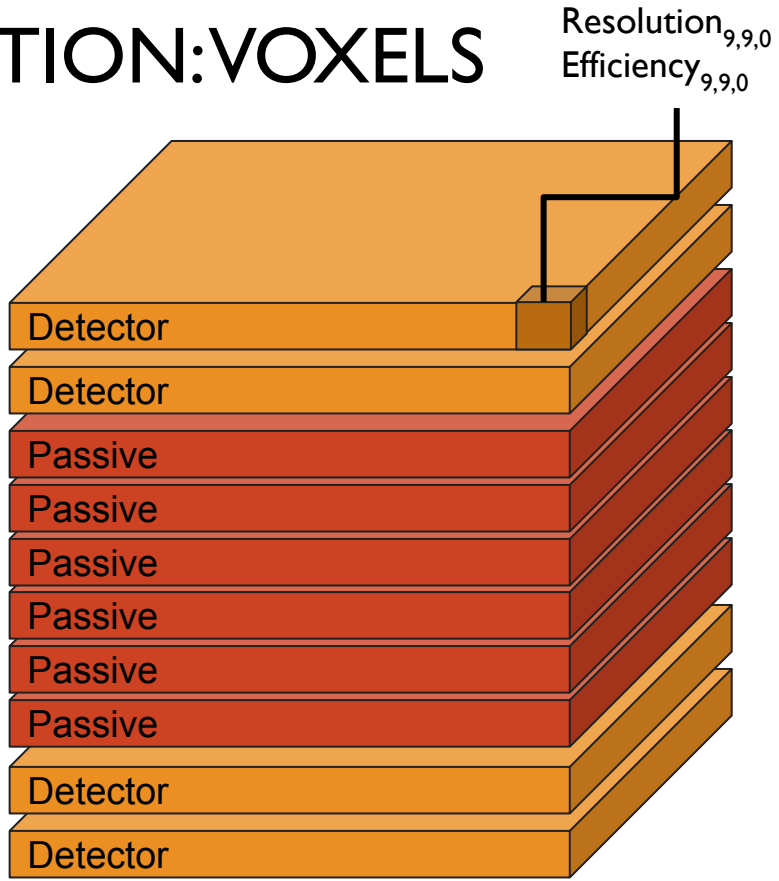
Example 2: Muon passes through detector. But no hit is recorded



Example 3: Muon doesn't pass through detector. No hit is recorded

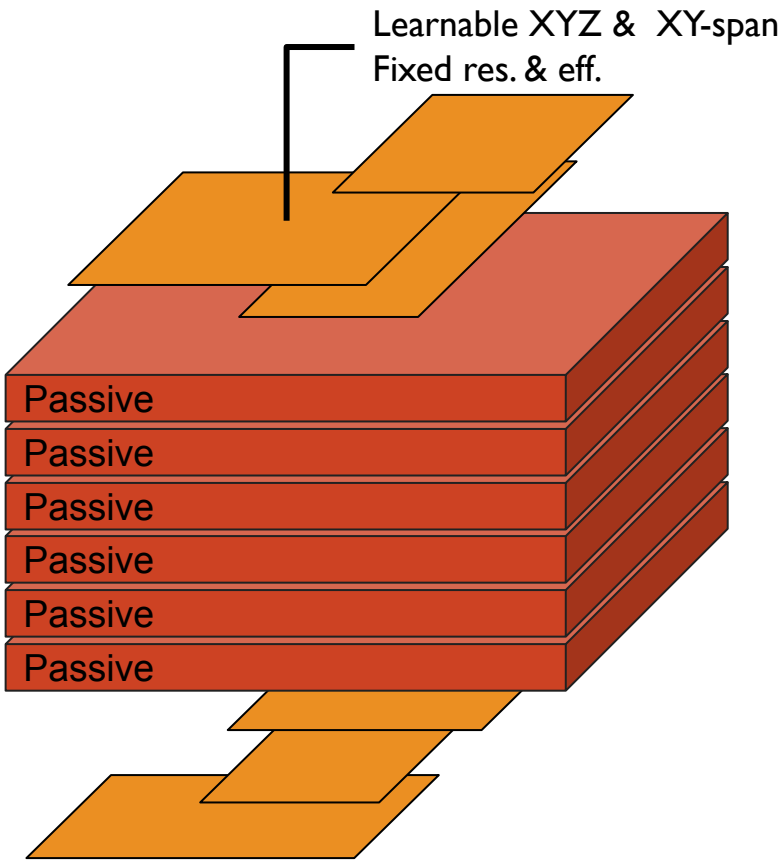
DETECTOR SIMULATION: VOXELS

- Detector layers are also voxelised
  - Each detector voxel *exists* and has resolution and efficiency parameters to be optimised
  - Cost of each detector voxel scales with its resolution and efficiency
- After optimisation, place lower bound on efficiency and only build detectors above the threshold
- Difficulties:
  - Detector types normally have fixed resolution, efficiency, and cost
  - Limited z positioning and optimisation capability
  - Likely to provide non-uniform detector layouts
  - Large difference between simulated and real detector
- Was used for early versions of TomOpt, but is now being replaced



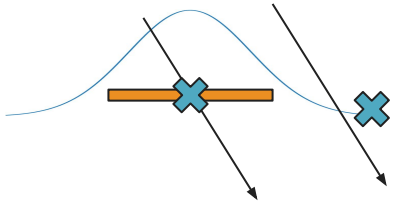
DETECTOR SIMULATION: PANELS

- Detector layers are regions of space containing a fixed number of *panels*
  - Each panel *exists* and has a fixed resolution and efficiency
  - The (x,y,z) position and xy-span are parameters to be optimised
  - Cost of each panel scales with its area
- Main difficulty: Muons either pass through detectors or miss them
  - How can we get gradients w.r.t. position and span?

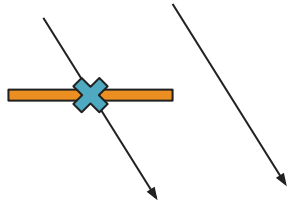


PANEL HITS

- Consider two modes:
  - Training: non-realistic simulation ok provided it is useful, loss gradients required
  - Evaluation: try to be as realistic as possible, loss gradients not required
- During training:
  - Panels have resolution & efficiency parameterised as rescaled Gaussians in xy:
    - Centres = at xy position of panel
    - Scales = panel span in xy
  - All muons have hits for every panel
    - Hits are diff. w.r.t. resolution and efficiency and therefore xy position and xy span
    - Z gradient comes later in scattering inference
  - N.B. Could keep resolution constant, but from optimisation point of view, the more effect a parameter has, the better
- During evaluation:
  - Hits outside the panels are ignored
  - Hits inside panel recorded with full resolution & efficiency



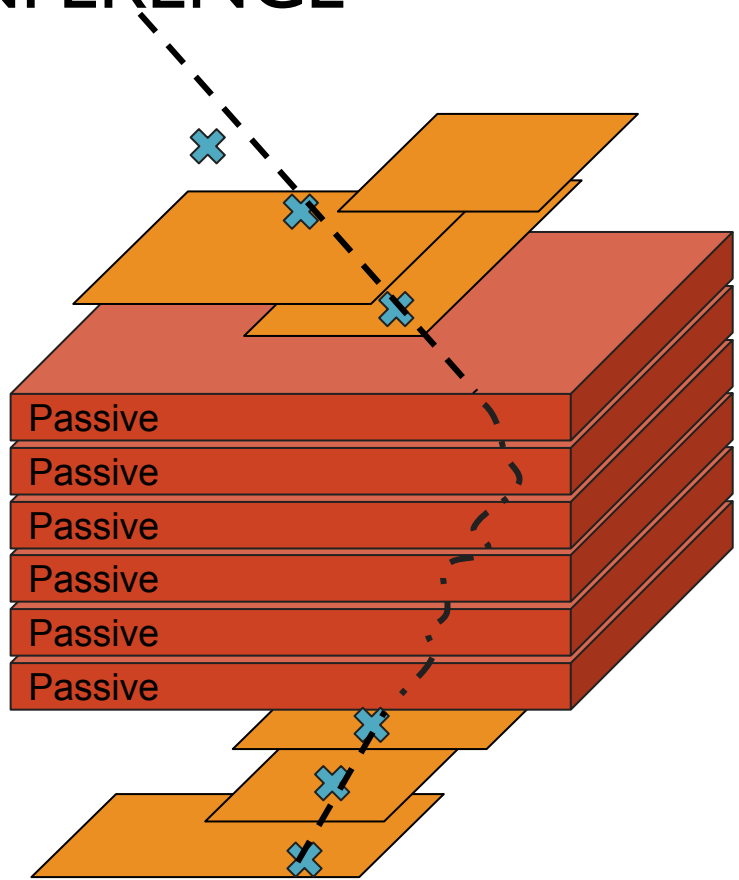
Training:
Both muons recorded, but with different resolutions



Evaluation:
Only the muon passing through the detector is considered

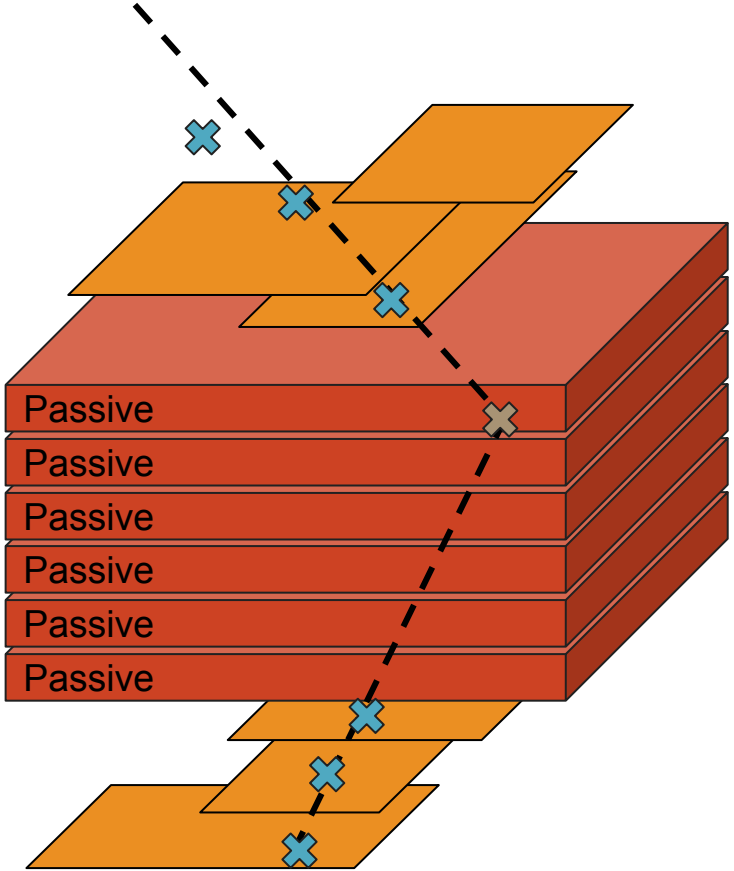
SCATTERING INFERENCE

- Once hits are recorded, need to reconstruct muon trajectory
- 1st step is to compute incoming and outgoing muon paths:
  - Fit straight lines to hits in detector layers above and below passive volume
  - Fits performed by considering the uncertainty (1/(resolution*efficiency)) on each hit
    - Trajectories diff. w.r.t. xyz position and xy-span of panels
- From this we have:
  - θ_{in}, θ_{out}, & $\Delta\theta$ and their uncertainties (thanks to autodiff)
- However, muon will scatter multiple times within the volume:
  - We cannot be certain of the true trajectory



SCATTERING INFERENCE: POCA

- Simplest inference method: Point Of Closest Approach (POCA)
  - Assumes that the muon only scatters once, and at the point where the extrapolated muon lines meet (are closest)
- From this, we now have the scatter location, the xy displacement, and their uncertainties (again, via autodiff)



X_0 INFERENCE

- Two complementary measurements of X_0 possible, via $\Delta\theta$ scattering and Δxy displacement (not currently implemented - doesn't carry much info.)
- POCA means only the X_0 of a single voxel is inferred (later we correct for this)
- Inference involves inverting the scattering formula and averaging over many muons
- Currently we assume knowledge of the muon momentum, p_{muon}

Extrapolate hits to measure $\Delta\theta_x$ & $\Delta\theta_y$

$$\Delta\theta^2 = \Delta\theta_x^2 + \Delta\theta_y^2 = \theta_{\mathrm{msc}}^2$$

$$\theta_0^2 = \frac{\Delta\theta^2}{2\mathcal{N}(0,1)^2}$$

$$N_{\mathrm{rad-lengths}} = \frac{\Delta\theta^2}{2\mathcal{N}(0,1)^2} \times \left(\frac{p_{\mathrm{muon}}}{0.0136}\right)^2$$

$$X_0 = \frac{\Delta\theta^2}{2\mathcal{N}(0,1)^2} \times \left(\frac{p_{\mathrm{muon}}}{0.0136}\right)^2 \times \frac{\cos\theta}{\delta z}$$

$$cos\theta = 0.5 \times (\cos\theta_{\mathrm{in}} + \cos\theta_{\mathrm{out}})$$

Average over muons, $\left\langle \mathcal{N}(0,1)^2 \right\rangle = 1$

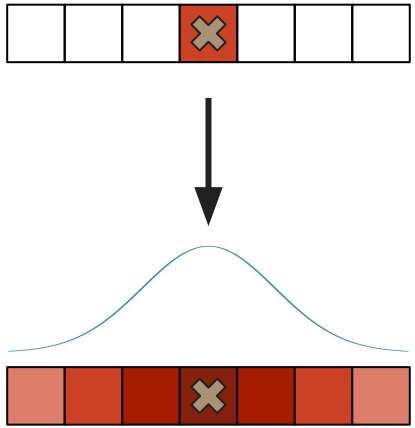
$$\langle X_0 \rangle = \left\langle \Delta\theta^2 \times \left(\frac{p_{\mathrm{muon}}}{0.0136}\right)^2 \times \frac{\cos\theta}{\delta z} \right\rangle$$

PREDICTION AVERAGING

- In the previous slide we derived $\langle X_0 \rangle$ by averaging over all muons with POCAs in the same voxel in order to set $\langle N(0,1)^2 \rangle = 1$
- However:
  - Not all muons are the same:
    - The uncertainties in their predictions can vary
    - Their efficiency can also vary
      - Due to book-keeping and simulation efficiency, we implement detection efficiency as a probability weight rather than a random pass/fail
    - Therefore weight muon prediction by efficiency and uncertainty
  - We want to try to improve the POCA method a bit to consider that the scattering does not take place at a single point
    - This improvement (discussed next) provides a 3rd weight

POCA IMPROVEMENT

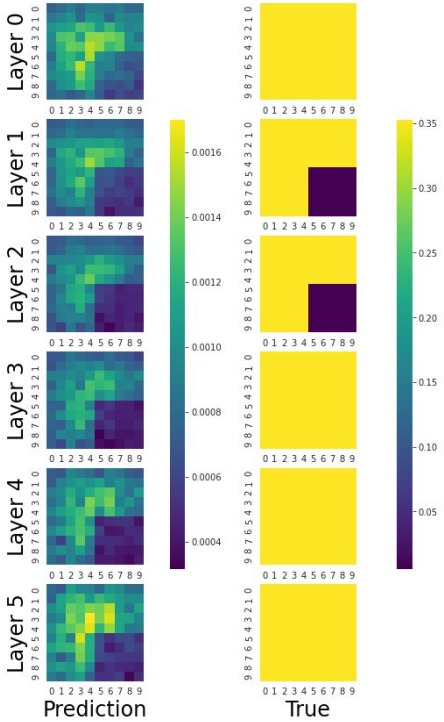
- Vanilla POCA assigns entirety of scattering to single point
  - But we have an uncertainty on where that point is
- Instead consider a 3D in xyz:
  - Centre = nominal scatter location
  - Scale = scatter location uncertainty in xyz
  - Integrating the Gaussian PDF over a voxel provides a probability of the scattering having occurred within that voxel
- Every muon now predicts the X_0 of every voxel, but predictions are now also weighted by scattering probability
  - Ideally, X_0 prediction should change per voxel according to $\Delta\theta$, but difficult to implement uncertainty calculation



Single muon goes from providing inference of single voxel, to providing Gaussian weighted predictions of all voxels (shown on in 1D here)

IMAGING

- Example volume:
  - Block of lead (X_0=0.005612m)
  - Surrounded by beryllium (X_0=0.3528m)
- Prediction based on 100k muons
  - ~2h computation time
  - Predictions highly biased to underestimate X_0
  - Lead block clearly visible, but high z uncertainty in scatter location causes ‘ghosting’ above and below
  - Predictions suffer edge-effects due to high probability for muons to transversely exit volume near edges



LOSS

- The X_0-prediction module provides the weighted average of predictions per voxel
  - Thanks to the Gaussian spreading, every voxel has an X_0 prediction
  - The sum of weights per voxel are also provided
- The loss of the system should contain two components:
  - The weighted error on predictions (diff. w.r.t. xyz position & xy-span)
    - Weight includes absolute value of efficiency
  - The cost of the detectors (diff. w.r.t xy-span)

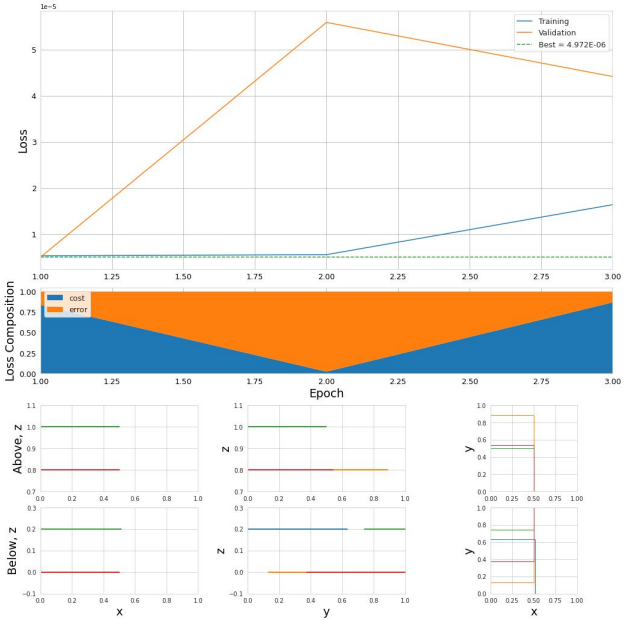
$$\mathcal{L}_{\text{Error}} = \frac{1}{N_{\text{voxels}}} \sum_{i=1}^{N_{\text{voxels}}} \frac{\left(X_{0,i,\text{True}} - X_{0,i,\text{Pred.}}\right)^2}{w_i}$$

$$\mathcal{L}_{\text{Cost}} = \sum_{i=1}^{N_{\text{panels}}} f_i\left(\text{span}_{x,i}, \text{span}_{y,i}\right)$$

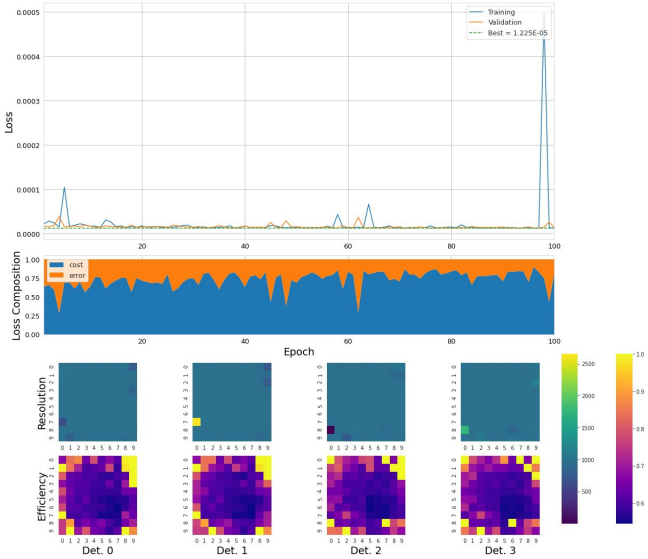
$$\mathcal{L} = \mathcal{L}_{\text{Error}} + \alpha \mathcal{L}_{\text{Cost}}$$

OPTIMISATION

- We can now backpropagate the loss gradient to the detector panel parameters
- Standard optimisers (SGD, Adam, etc.) can be used to update parameters
- TomOpt provides live feedback of optimisation
  - Both loss and detector states
- TomOpt also has a stateful callback system for easily augmenting training/prediction procedures



Panel-detector optimisation example (work in progress)



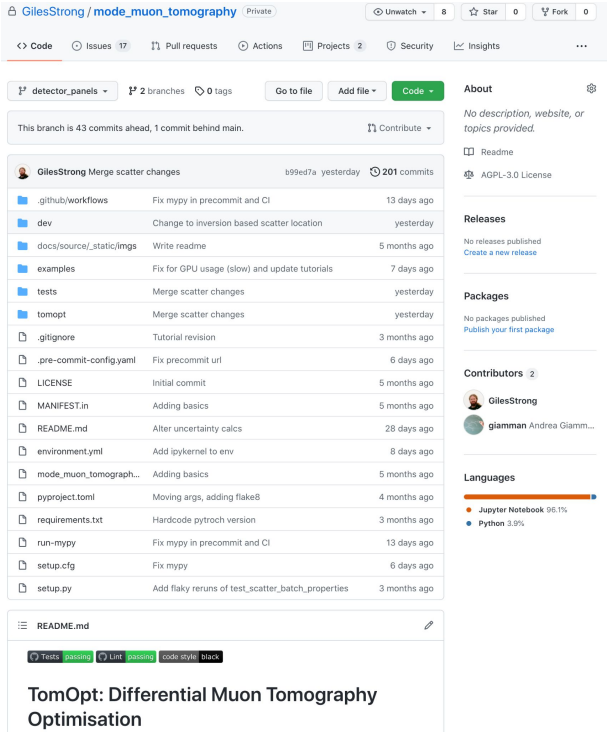
Voxel-detector optimisation

CODE DEVELOPMENT

What we're working on and how you can get involved

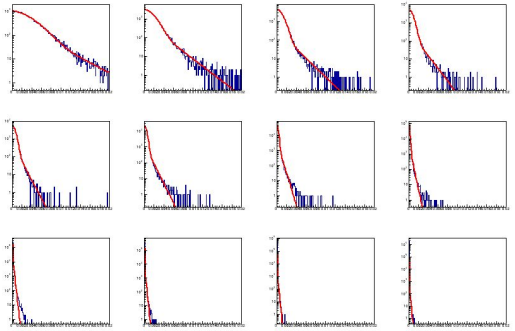
DEVELOPMENT

- Currently developed in private repo
  - Implicitly accessible for MODE members
  - Potential external contributors are welcome to contact us
- Will make repo public once publication plan is decided, or project is mature enough
  - Still lots of work tasks to carry out and improve on (see upcoming slides)
  - Interested members should start to get involved soon, so we can ensure authorship

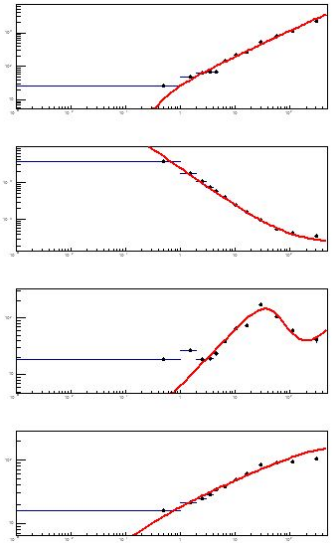


MUONS

- Kinematics of muons generated from a basic model with fixed momentum
  - Proper generation should be performed from a more accurate model
  - Generation should vary according to latitude & longitude of experiment
  - Muons should be generated with a variety of momenta
- Scattering of muons through the passive volume is performed using a simplified model
  - Dorigo & Kieseler are helping to improve the scattering model
  - Create model of $\Delta\theta$ PDF for Δz step in given X_0 parameterised by muon momentum



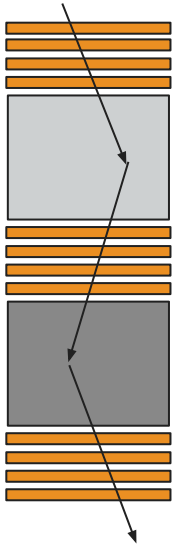
Fit to $\Delta\theta$ in bins of momentum



Fit to fit-parameters as function of momentum

INFERENCE

- X_0 inference assumes knowledge of muon momenta: need to also infer momenta prior to inferring X_0
  - Can be achieved with a second passive volume of known X_0 and a third detector layer
  - Solve for momenta via scattering in second volume, then solve for X_0 in first volume
  - Giammanco + Naples & Florence have interest, knowledge, & experience with this
- POCA+spread is still suboptimal and biased
  - Dorigo & Vischia are looking into a maximum likelihood fit approach
  - (Pretrained) DNN could apply residual corrections to predicted X_0
- Inference produces X_0 values (float), but really we want to know material class (category)
  - (Pretrained) U-net could provide class predictions per voxel

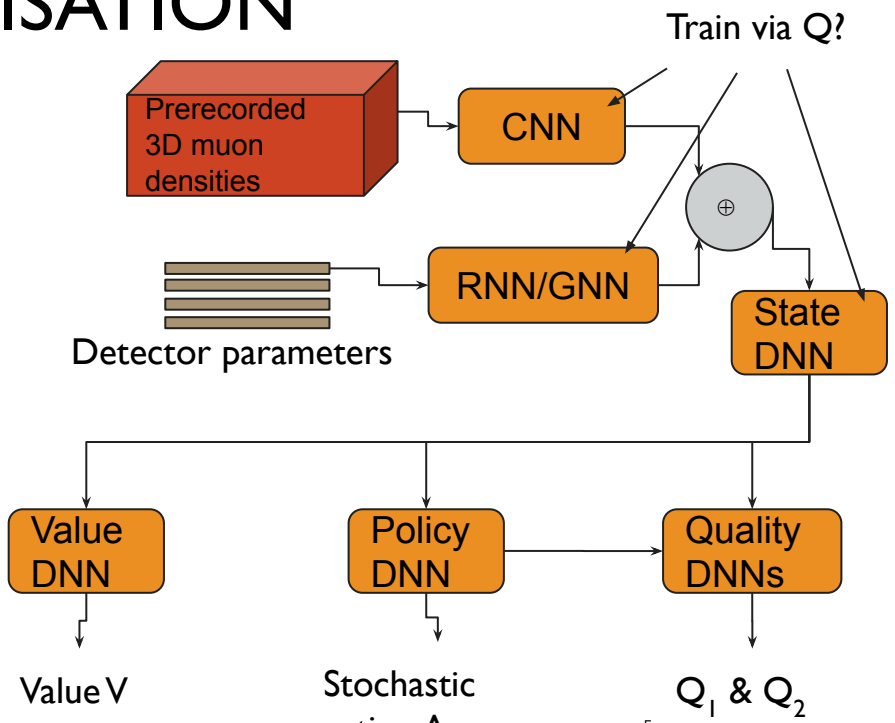


2. Measure X_0 of unknown material using momentum measurement

1. Measure momentum via deflection in material of known X_0

OPTIMISATION

- For a given batch of passive volumes, the average maps of true muon-trajectories are going to be the same, regardless of the detector configuration
  - How can the optimisation algorithm make use of this *off-policy* data?
- Current optimisation cannot create/destroy detectors
  - This either limits the optimisation potential, or requires us to perform separate optimisations for different panel multiplicities
- Can we take inspiration from reinforcement learning? E.g.:
  - State = current panels and trajectory maps (collected off-policy or computed prior to optimisation)
  - Action = create a new panel by specifying position and span
  - Reward = inverse prediction-error minus detector costs
  - Lack of backprop through X_0 and scatter inference would reduce time considerably



$$\mathop{\mathrm{E}}_{s\sim\mathcal{D},\tilde{a}\sim\pi_\theta}\left(V_\psi(s)-\left(\min_{i=1,2}Q_{\phi_i}(s,\tilde{a})-\alpha\log\pi_\theta(\tilde{a}|s)\right)\right)^2$$

$$\mathop{\mathrm{E}}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q_{\phi_i}(s,a)-\left(r+\gamma V_{\psi_{\mathrm{targ}}}(s')\right)\right)^2\right]$$

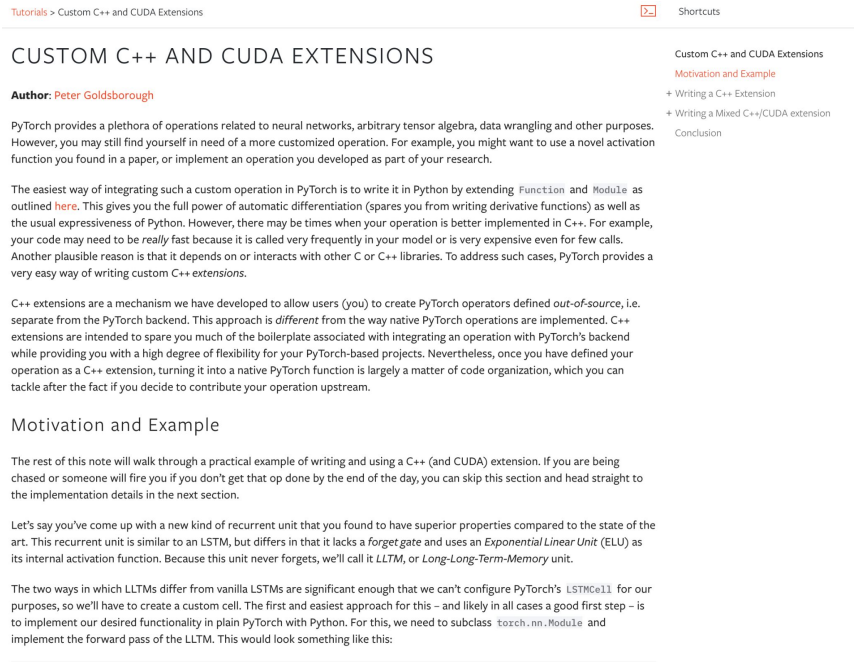
$$\tilde{a}_\theta(s,\xi)=\tanh\left(\mu_\theta(s)+\sigma_\theta(s)\odot\xi\right)$$

$$\mathop{\mathrm{E}}_{s\sim\mathcal{D},\xi\sim\mathcal{N}}\left[Q_{\phi_1}(s,\tilde{a}_\theta(s,\xi))-\alpha\log\pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right]$$

Soft Actor-Critic, Haarnoja et al., 2018

CODE OPTIMISATION

- Code is fully vectorised, but generally slow
- PyTorch makes it "easy" to JIT compile routines and include custom CUDA kernels
  - Once modules become stable, they could be compiled?
- GPU performance is 2x slower than CPU
  - Several intensive computations rather than big matrix multiplication
  - Forward pass computed serially for many batches of muons and then backproped
    - Potentially possible to parallelise muon batches on GPU
- Autodiff for uncertainty computation requires batchwise Jacobian
  - Parallelised using PyTorch vmap (experimental)
  - Doesn't seem to scale well
    - Quicker to run 10 batches of 100 muons than 1 batch of 1000 muons
  - Can this be improved by reducing the number of uncertainties to calculate?
  - Maybe move to JAX?
    - Would require heavy rewriting



Tutorials > Custom C++ and CUDA Extensions

Shortcuts

CUSTOM C++ AND CUDA EXTENSIONS

Custom C++ and CUDA Extensions
Motivation and Example
+ Writing a C++ Extension
+ Writing a Mixed C++/CUDA extension
Conclusion

Author: Peter Goldsborough

PyTorch provides a plethora of operations related to neural networks, arbitrary tensor algebra, data wrangling and other purposes. However, you may still find yourself in need of a more customized operation. For example, you might want to use a novel activation function you found in a paper, or implement an operation you developed as part of your research.

The easiest way of integrating such a custom operation in PyTorch is to write it in Python by extending `Function` and `Module` as outlined here. This gives you the full power of automatic differentiation (spares you from writing derivative functions) as well as the usual expressiveness of Python. However, there may be times when your operation is better implemented in C++. For example, your code may need to be *really* fast because it is called very frequently in your model or is very expensive even for few calls. Another plausible reason is that it depends on or interacts with other C or C++ libraries. To address such cases, PyTorch provides a very easy way of writing custom *C++ extensions*.

C++ extensions are a mechanism we have developed to allow users (you) to create PyTorch operators defined *out-of-source*, i.e. separate from the PyTorch backend. This approach is *different* from the way native PyTorch operations are implemented. C++ extensions are intended to spare you much of the boilerplate associated with integrating an operation with PyTorch's backend while providing you with a high degree of flexibility for your PyTorch-based projects. Nevertheless, once you have defined your operation as a C++ extension, turning it into a native PyTorch function is largely a matter of code organization, which you can tackle after the fact if you decide to contribute your operation upstream.

Motivation and Example

The rest of this note will walk through a practical example of writing and using a C++ (and CUDA) extension. If you are being chased or someone will fire you if you don't get that op done by the end of the day, you can skip this section and head straight to the implementation details in the next section.

Let's say you've come up with a new kind of recurrent unit that you found to have superior properties compared to the state of the art. This recurrent unit is similar to an LSTM, but differs in that it lacks a *forget gate* and uses an *Exponential Linear Unit* (ELU) as its internal activation function. Because this unit never forgets, we'll call it *LLTM*, or *Long-Long-Term-Memory* unit.

The two ways in which LLTMs differ from vanilla LSTMs are significant enough that we can't configure PyTorch's `LSTMCell` for our purposes, so we'll have to create a custom cell. The first and easiest approach for this – and likely in all cases a good first step – is to implement our desired functionality in plain PyTorch with Python. For this, we need to subclass `torch.nn.Module` and implement the forward pass of the LLTM. This would look something like this:

https://pytorch.org/tutorials/advanced/cpp_extension.html

INSTALLATION

- Installation instructions provided for MacOS & Linux
- Assumes Conda for python
- Requirements listed for easy environment setup
- Installs the tomopt package locally
- Sets up *pre-commit* to validate new code locally (more on this later)

Installation

N.B. Whilst the repo is private, you will need to make sure that you have registered the public ssh key of your computer/instance with your GitHub profile. Follow these instructions to check for existing keys or these to generate a new key. After that follow this to associate the key.

Checkout package:

```
git clone git@github.com:GilesStrong/mode_muon_tomography.git
cd mode_muon_tomography
```

N.B. For GPU usage, it is recommended to manually setup conda and install PyTorch according to system, e.g.:

```
conda activate root
conda install nb_conda_kernels
conda env create -n tomopt python=3.8 pip ipykernel
conda activate tomopt
pip install torch==1.8.1+cu111 -f https://download.pytorch.org/whl/torch_stable.html
pip install -r requirements.txt
```

Minimum python version is 3.8. Recommend creating a virtual environment, e.g. assuming your are using Anaconda/Miniconda (if installing conda for the first time, remember to restart the shell before attemting to use conda, and that by default conda writes the setup commands to `.bashrc`):

```
conda activate root
conda install nb_conda_kernels
conda env create -f environment.yml
conda activate tomopt
```

Otherwise set up a suitable environment using your python distribution of choice using the contents of `environment.yml`. Remember to activate the correct environment each time, via e.g. `conda activate tomopt`.

Install package and dependencies

```
pip install -r requirements.txt
pip install -e .
```

Install git-hooks:

```
pre-commit install
```

INTRO TO THE CODEBASE

- Six examples provided as Jupyter Notebooks
  - Three for voxel-based detectors and three for panel-based detectors
- Hello World
  - High-level tutorial for general users
  - Demonstrates how to set up volumes and detectors, run optimisation, and produce images
- Two in-depth tutorials
  - Low-level tutorial for developers and advanced users
  - Demonstrates how all the sub-modules work and interact with one another

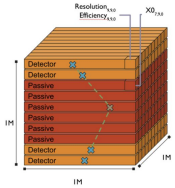
In [1]:
```
%matplotlib inline
%reload_ext autoreload
%autoreload 2
```

1 Hello World

This tutorial/example aims to give an simplistic overview of setting up a passive volume (scattering material to be imaged), detector (active material to record hits), and optimisation loop (to refine the detector to image the passive volume better or more cheaply)

1.1 Volume setup

The volume consists of both passive material to be imaged, and active detectors to record muons hits. We aim to adjust the resolution and efficiency of the detectors to minimise the cost of the detector whilst improving the precision of the imaging of the passive volume.



The above image shows a typical layout, with two detection layers above and below the passive volume consisting of layers of material. Each layer consists of sub-cubes (voxels). In the detector layers, these are elements of the detector with properties to be optimised, and in the passive layers these can be different materials, whose X0 will affect the scattering of muons to deferring extents.

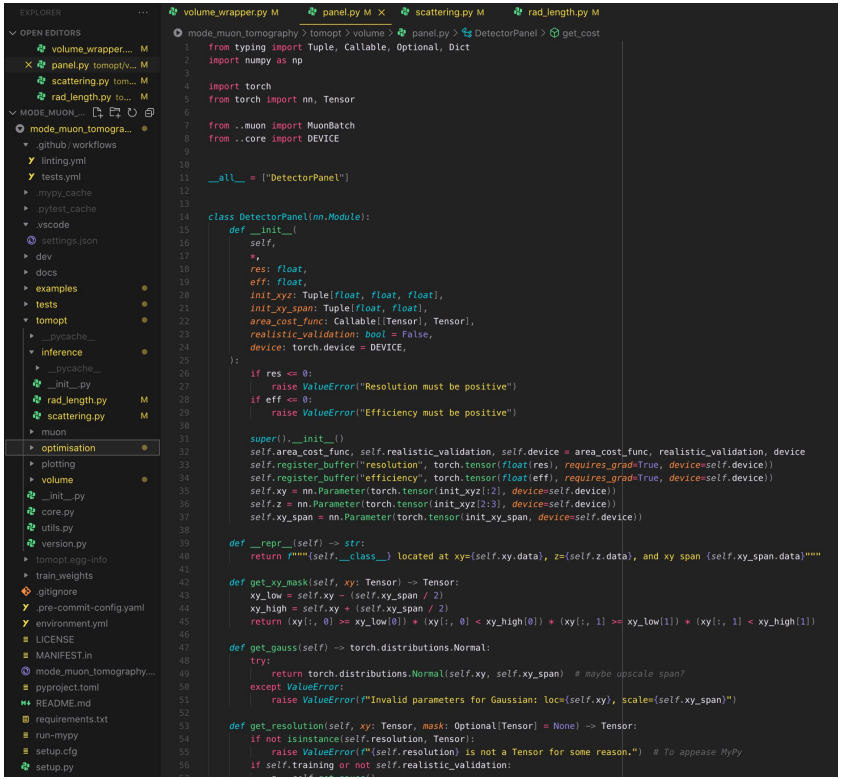
We need to define the cost of the detector elements according to resolution and efficiency. These are currently arbitrary, and discussed further in the next tutorial:

In [2]:
```
import torch
from torch import Tensor
from torch.nn import functional as F

import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:
```
def eff_cost(x:Tensor) -> Tensor:
    return torch.expm1(3*F.relu(x))  # free for negative efficiency, sharp rise as efficiency increases

x = torch.linspace(0,1,10)
plt.plot(x, eff_cost(x))
```

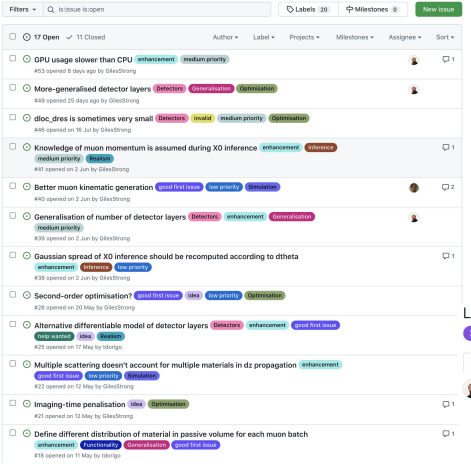
INTERACTING WITH THE CODE

- Code divided into submodules
  - .py files, no need to be familiar with developing in Jupyter Notebooks
- No docstrings (yet)
  - Removes chance of them becoming stale due to rapid development
- Code is fully type-hinted
  - Checked by MyPy
- Format is automatically set by Black
  - Appearance of code is the same, regardless of who writes it

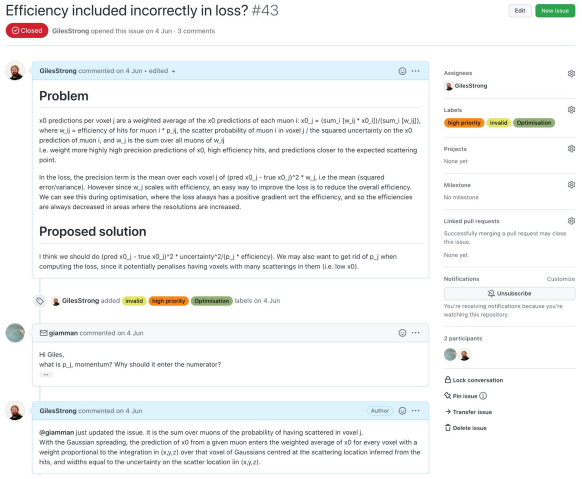


CONTRIBUTING TO THE CODEBASE

1. Pick work task



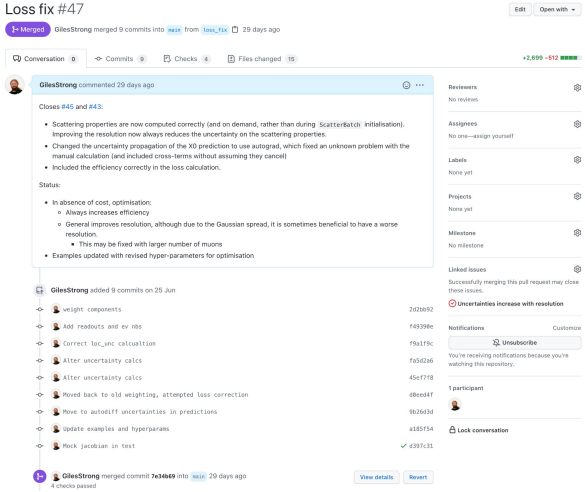
2. Discuss solution



3. Develop code

```python
class DetectorLoss(nn.Module):
    def __init__(self, cost_coef: Optional[Union[Tensor, float]] = None):
        super().__init__()
        self.cost_coef = cost_coef
        self.sub_losses: Dict[str, Tensor] = {}  # Store subcomponents in dict for telemetry

    def _compute_cost_coef(self, cost: Tensor, inference: Tensor) -> None:
        self.cost_coef = inference.detach() / cost.detach()
        print(f'Automatically setting cost coefficient to {self.cost_coef}')

    def forward(self, pred_x0: Tensor, pred_weight: Tensor, volume: Volume) -> Tensor:
        true_x0 = volume.get_rad_cube()
        inference = torch.mean((pred_x0 - true_x0).pow(2) / pred_weight)
        self.sub_losses["error"] = inference
        cost = volume.get_cost()
        if self.cost_coef is None:
            self._compute_cost_coef(cost, inference)
        self.sub_losses["cost"] = self.cost_coef * cost
        return self.sub_losses["error"] + self.sub_losses["cost"]
```

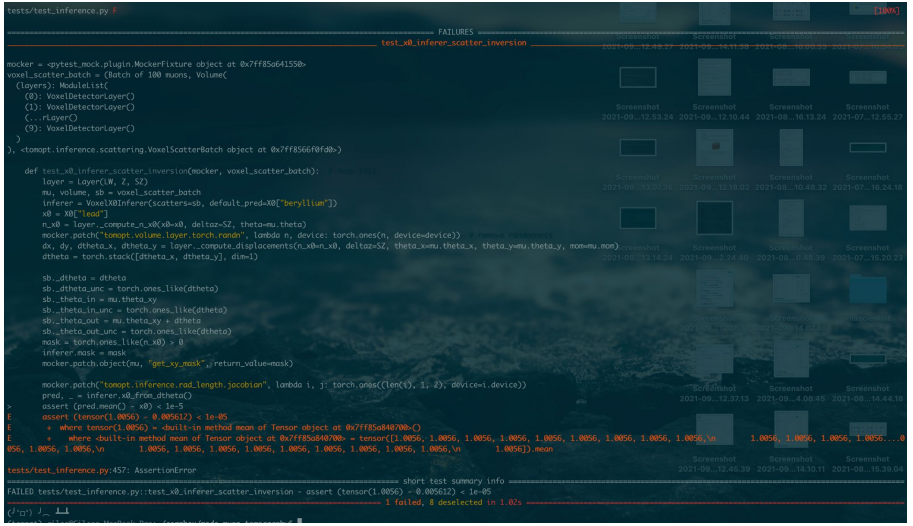
4. Merge & discuss changes



CONTINUOUS INTEGRATION & TESTING

- Pre-commit runs local checks of new code
  - Prevents invalid code being committed (Flake8)
  - Ensures code is properly type-hinted (MyPy)
  - Automatically reformats code (Black)
- All code is unit-tested (pytest):
  - Every class & function has tests to ensure they work as expected
- Tests and linting checks run automatically when creating a pull-request
  - Can also be run locally

```python
def test_x0_inferer_scatter_inversion(mocker, voxel_scatter_batch):  # noqa F811
    layer = Layer(LW, Z, SZ)
    mu, volume, sb = voxel_scatter_batch
    inferer = VoxelX0Inferer(scatters=sb, default_pred=X0["beryllium"])
    x0 = X0["lead"]
    n_x0 = layer._compute_n_x0(x0=x0, deltaz=SZ, theta=mu.theta)
    mocker.patch("tomopt.volume.layer.torch.randn", lambda n, device: torch.ones(n, device=device))  # remove randomness
    dx, dy, dtheta_x, dtheta_y = layer._compute_displacements(n_x0=n_x0, deltaz=SZ, theta_x=mu.theta_x, theta_y=mu.theta_y, mom=mu.mom)
    dtheta = torch.stack([dtheta_x, dtheta_y], dim=1)

    sb._dtheta = dtheta
    sb._dtheta_unc = torch.ones_like(dtheta)
    sb._theta_in = mu.theta_xy
    sb._theta_in_unc = torch.ones_like(dtheta)
    sb._theta_out = mu.theta_xy + dtheta
    sb._theta_out_unc = torch.ones_like(dtheta)
    mask = torch.ones_like(n_x0) > 0
    inferer.mask = mask
    mocker.patch.object(mu, "get_xy_mask", return_value=mask)

    mocker.patch("tomopt.inference.rad_length.jacobian", lambda i, j: torch.ones((len(i), 1, 2), device=i.device))  # remove randomness
    pred, _ = inferer.x0_from_dtheta()
    assert (pred.mean() - x0) < 1e-5
```



SUMMARY

- TomOpt is setting out to investigate the practicality of the optimisation pipeline proposed in MODE, 2021
  - Industry contacts via Louvain and Ruiz Del Arbol mean that MODE is well positioned to work on this, making muon tomography is a suitable first application
- TomOpt framework generally populated
  - Still lots of opportunities to provide valuable contributions
  - Up-to-date examples and tutorials to help with on-boarding