Building and Solving Probabilistic Instrument Models with CaliPy

Jemil Avers BUTT[1,2,*]

[1] *ETH Zurich IGP, Zurich, Switzerland, (jemil.butt@geod.baug.ethz.ch)*

[2] *Atlas optimization GmbH, Zurich, Switzerland, (jemil.butt@atlasoptimization.ch)*

[*] *corresponding author*

Abstract

Probabilistic models of geodetic measurement instruments are essential ingredients for uncertainty quantification and optimal estimation. Traditionally, these models are formulated and analyzed manually with inference relying on classical maximum-likelihood-based parameter estimation. However, this approach typically falls short when dealing with nonlinear models, non-Gaussian distributions, or latent random variables. To overcome these limitations, we developed CaliPy (Calibration library Python), a Python library built on top of Pyro and PyTorch. CaliPy is designed to facilitate the construction, solution, and exchange of probabilistic instrument models by chaining together pre-built stochastic effects. It leverages advances in deep probabilistic programming and automatic differentiation to perform automated Bayesian inference. In this paper we demonstrate CaliPy's architecture and practical application through examples involving instrument models featuring drifts and noise. Our results highlight CaliPy's ability to handle complex probabilistic models in a unified framework, thereby offering significant benefits to the users of measurement instruments by streamlining model formulation, solution, and exchange while also providing a framework for implementing chainable stochastic effects in a Python-friendly ecosystem.

Keywords: Probabilistic programming, Instrument models, Calibration, Stochastic models, Machine learning

1 Introduction

Interpreting real-world measurement data beyond simple summary statistics requires a probabilistic instrument model, i.e. a mathematical model of the measurements that features randomness. Such an instrument model tying observations to unknown quantities of interest allows the allocation of observed effects in the data to the assumed inner workings of the measurement instrument or some absolute standard (i.e. calibration, Phillips et al. (2001)). Passing from descriptive to inductive statistics has benefits ranging from the inclusion of prior domain knowledge to the possibility of model checking, comparison, and understanding in a fully Bayesian workflow (Gelman et al., 2020).

As noticed in Gelman et al. (2020), Henderson et al. (2010), such an iterative workflow enabling models to be fitted quickly for the purpose of model exploration and navigation is an advantage when models, model components, and their relationship to the data are not yet definitely established and the subject of investigation. This is especially relevant to geodesy where probabilistic models abound, see e.g. the compilation in Neitzel (2021). For example, the development of mathematical instrument models representing terrestrial laser scanners (TLS) followed a similar pattern with complexity increasing from straightforward deterministic models over models with multiple deterministic trend functions (Lichti, 2010) towards more stochastically oriented models (Kerekes and Schwieger, 2020).

However, the standard workflow for geodetic model development still seems to be centered around the method of Least Squares (LS) estimation (Ghilani and Wolf, 2006, pp. 8 -10) with model selection constrained to those solvable via LS. Furthermore, manually written computer code for the LS problems and their closed form solutions or heuristic iterative schemes are still the prevalent mode of implementation and documentation which can have negative implications for correctness and exchangeability of these instrument models. For the remainder of the paper, LS refers to traditional solution

procedures for Gauss-Markov/Gauss-Helmert models (Niemeier, 2008, p. 172); mixed integer convex optimization solvers are not considered.

In principle, probabilistic programming languages (PPLs) such as Pyro (Bingham et al., 2018) enable rapid building and solving (i.e inferring unknown parameters and posteriors) of probabilistic models thereby making them ideal for experimental geodetic data analysis. By combining the autodifferentiation capabilities of packages like PyTorch (Paszke et al., 2019) with probabilistic concepts like distributions, independence, and sampling, these languages unite formal probabilistic model building with the convenience of automatic Bayesian inference. In practice, however, they are difficult to use owing to the demanding theoretical probabilistic background and the amount of PPL-specific expertise needed to operate them. As a consequence, PPLs did so far not find the widespread adoption that their deterministic predecessors did. As a matter of fact, the authors did not find any publication from the areas of engineering geodesy or deformation monitoring employing PPLs.

CaliPy aims to bridge the gap between theoretically universal and practically applicable probabilistic modeling by making geodetic instrument models easy to create and solve by building on top of the PPL Pyro and creating composable stochastic effects. With CaliPy, probabilistic models can be built conveniently by chaining together these stochastic effects and bespoke inference schemes are set up automatically. No manual calculations are necessary that convert the model into a form suitable for LS estimation. In short: Declaring a model is basically the same as solving it.

In summary, the contributions of this work are:

- We describe CaliPy, a library for building and solving probabilistic instrument models
- We illustrate CaliPy's usage by applying it to archetypic instrument modeling problems

Section 2 provides a review of classical instrument modeling and introduces CaliPy as a library for building and solving probabilistic instrument models. Section 3 describes CaliPy's architecture and underlying principles while section 4 illustrates CaliPy's abstract design patterns by means of concrete practical applications. Section 5 concludes the paper and outlines further developments.

2 Instrument models and CaliPy

2.1 Classic Approach to Instrument Models

Suppose that measurement data $y = [y_1, ..., y_{n_{\text{nobs}}}]^T \in \mathbb{R}^{n_{\text{nobs}} \times n_y}$ and input variables $x = [x_1, ..., x_{n_{\text{nobs}}}]^T \in \mathbb{R}^{n_{\text{nobs}} \times n_x}$ are given where n_{nobs} is the number of observations, and n_x, n_y are the dimensions of a single input variable or observation. The task of geodetic data analysis is often to find the best estimates of parameters $\theta \in \mathbb{R}^{n_\theta}$ that stand in some generative relationship to the measurement data y (Ghilani and Wolf, 2006, p. 179).

The parameters θ may correspond to unknown coordinates, deformation rates, instrument-specific biases, or other unobserved random or nonrandom quantities of interest in our model. The input variables x represent known deterministic quantities, e.g. the geometric configuration, meteorological conditions, or the time of an experiment. The formulation

$$y \sim \mathscr{P}(x, \theta) \tag{1}$$

declaring y as being distributed according to $\mathscr{P}(x, \theta)$ condenses these relations into a single statement. The probabilistic models (1) have a role similar to the observation equations in classical LS and include the narrower stochastic models, which are understood to be models for the covariance matrix of a multivariate Gaussian dstribution (Ghilani and Wolf, 2006, pp. 177 - 181). Estimating θ in the most general sense is solved by converting the joint distribution $p(y, x, \theta)$ into the posterior distribution $p(\theta|y, x)$ of the parameters given the data by employing Bayes theorem. A popular point estimate $\hat{\theta}$ summarizing $p(\theta|y, x)$ is the Maximum Likelihood (ML) estimator (Hastie et al., 2009, p. 265):

$$\hat{\theta}_{ML} = \text{argmax}_\theta \, p(y, x|\theta)$$

The ML estimator is widely used in geodesy as for linear models and a Gaussian assumption it is equivalent to Least Squares (Hastie et al., 2009, p. 265) via

$$\begin{aligned} \hat{\theta}_{LS} &= \text{argmin}_\theta \, \|A(x)\theta - y\|_2^2 \\ &= \text{argmax}_\theta \, c_0 * \exp\left(-\frac{\|A(x)\theta - y\|_2^2}{c_1}\right) \\ &= \text{argmax}_\theta \, p(y, x|\theta) \end{aligned}$$

Here $A(x)$ is a design matrix, c_0, c_1 are appropriately chosen positive constants, and $\|\cdot\|_2$ denotes the ℓ^2-Norm. The advantages of LS are its direct interpretability as the ML estimate under Gaussian assumption, the existence of closed form solutions to the LS problem, and well-established auxiliary results like the variance of the estimates. Furthermore, even when the Gaussian assumption might not hold, the LS estimate still minimizes a meaningful measure of discrepancy.

The disadvantages of standard LS entail its inability to handle in an systematic manner non-Gaussianity, non-convex functions, latent unobserved variables and discrete variables. For an overview of capabilities, see Nievergelt (2000) and for examples detailing the complexities of adopting LS for more flexible random effects, nonconvex objectives, latent variables, or discrete variables, see Schaffrin (2020),(Boyd and Vandenberghe, 2004, p. 9), Bollen (1996), and Del Pia et al. (2014) respectively. In combination with LS only delivering point estimates and not the full posterior distribution, this suggests the use of more suitable Bayesian inference methods like Stochastic Variational Inference (SVI).

2.2 Stochastic Variational Inference

SVI is an approximate inference procedure for addressing general Bayesian inference problems in a data-heavy context and is employed by CaliPy to estimate unknown parameters and posterior distributions. The goal of SVI is to approximate the posterior distribution $p(\theta|y,x)$ of the θ as well as possible by finding an approximating distribution q_ϕ from a simple family of functions indexed by the parameters ϕ. The objective function of SVI is the evidence lower bound (ELBO) (Blei et al., 2017)

$$\begin{aligned}\mathrm{ELBO}(q_\phi, \theta_D) = \log p_{\theta_D}(y) \\ - \mathrm{KL}(q_\phi(\theta_S) \| p_{\theta_D}(\theta_S|y,x))\end{aligned} \quad (2)$$

for $\theta = (\theta_D, \theta_S)$ a split of θ into unknown deterministic parameters θ_D and unobserved stochastic variables θ_S. The name ELBO derives from the fact that the Kullback-Leibler Divergence KL is nonnegative which makes the right-hand side of equation 2 a lower bound for the log evidence. The ELBO enables the dual goals of:

- Finding parameters θ_D corresponding to high evidence $p_{\theta_D}(y)$
- Finding parameters ϕ such that q_ϕ approximates well the posterior distribution $p_{\theta_D}(\theta_S|y,x)$ of θ_S.

Maximizing the ELBO jointly maximizes the evidence of the model and minimizes the divergence between true posterior and approximate posterior. It is the default objective measuring model quality in the PPL Pyro (Bingham et al., 2018) which serves as a basis for CaliPy. When there are unobserved stochastic variables θ_S whose posterior needs to be approximated, the gradients necessary for optimizing the ELBO have to be sampled instead of simply computed. When only deterministic parameters are present, the ELBO degenerates to the maximum likelihood objective $\log p_{\theta_D}(y)$ and therefore basically Least Squares if all involved distributions are Gaussian.

2.3 Specifying Models in CaliPy

Maximizing the ELBO via SVI requires differentiating $p_{\theta_D}(\cdot)$ and $q_\phi(\cdot)$ and sampling $q_\phi(\theta_S)$ but not much else (Kingma and Welling, 2019, p.20). It is therefore sufficient for SVI to specify a model function that determines the joint probability density $p_{\theta_D}(y,x,\theta_S)$ (or equivalently likelihood and prior) and to specify the variational distribution $q_\phi(\theta_S)$. In Pyro, this is done by the two user-built Python functions model() and guide() which might contain nonlinear transforms, sampling statements, and complicated control flow. In CaliPy however, we emphasize a modular design built around pre- or user-defined stochastic effects encoded in Python classes like NoiseAddition or PolynomialTrend. Instantiating one such class leads to an object of well-defined shape and conditional independence. Executing its forward() method generates one sample of this effect. The code for a simple mean estimation is provided below for illustration purposes.

```
# dims setup
data_dims = dim_assignment(dim_names = ['bd', 'ed'])
# mu setup
mu_ns = NodeStructure(UnknownParameter)
mu_ns.set_dims(['Independent_dims'], data_dims[0:1])
mu_ns.set_dims(['Dependent_dims'], data_dims[1:2])
mu_object = UnknownParameter(mu_ns, name = 'mu')
# noise setup
noise_ns = NodeStructure(NoiseAddition)
noise_ns.set_dims(['Independent_dims'], data_dims[0:2])
noise_object = NoiseAddition(noise_ns, name = 'noise')
```

The above is enough to specify the unknown mean and noise for the model. As there are no unobserved latent variables and therefore also no guide / varia-

tional distribution needs to be defined, the information necessary for SVI is complete. Integrating both model and empty guide function into a CalipyProbModel instance allows for automatic inference.

```
class DemoProbModel(CalipyProbModel):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # integrate nodes
        self.mu_object = mu_object
        self.noise_object = noise_object
    # Define model by forward passing
    def model(self, input_vars = None,
        observations = None):
        mu = self.mu_object.forward()
        output = self.noise_object.forward((mu,
            sigma_true), observations = observations)
        return output
    # Define guide (trivial since no posteriors)
    def guide(self, input_vars = None,
        observations = None):
        pass

demo_probmodel = DemoProbModel()
demo_probmodel.train(data)
```

Note that it is allowed to apply arbitrary differentiable Pytorch functions to the output of the intermediate processing steps thereby providing the possibility to build complicated stochastic interactions that can represent nontrivial real-world behavior. The command **probmodel.train()** works exactly the same in these cases as well.

3 Architecture & Implementation

3.1 Software Architecture

The design goal of CaliPy is twofold: enable simple instrument model building, solving, and sharing and maintain as much as feasible compatibility to the Pytorch and Pyro ecosystems. We achieve this by repackaging Pyro's functionality into separate effects that represent nodes in a DAG and can be chained to form a model without the users spelling out all the details themselves. Users need to call a prebuilt subclass (like UnknownParameter or NoiseAddition) of the CalipyNode AbstractBaseClass and combine it with a NodeStructure object that encodes dimensions, sizes, and independence assumptions. The result is an effect object with a **forward()** method that produces concrete numbers for forward simulations and keeps track of gradients and probabilities for the backward inference pass. Model and guide function are integrated into a CalipyProbModel object which manages inference by compiling the problem and handing it to Pyro. The procedure is illustrated in figure 1.

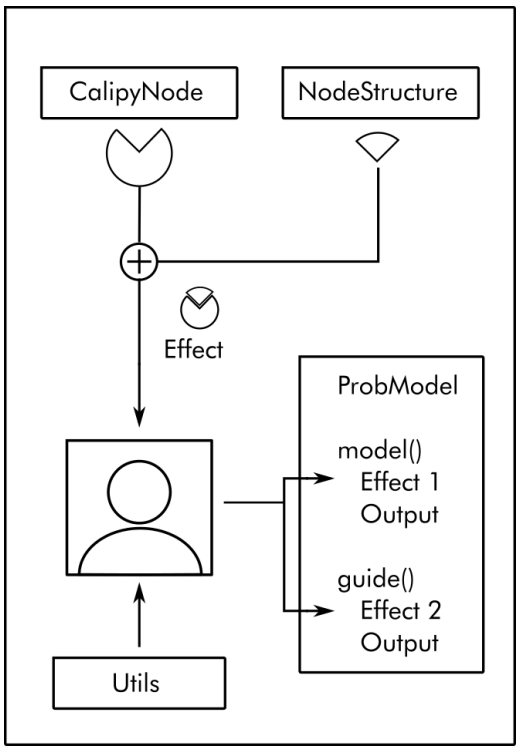

Figure 1. The user specifies models by calling nodes, imprinting them with structure, and then chaining the resulting effects together.

Apart from the three abstract classes CalipyNode, NodeStructure, and CalipyProbModel, a collection of utility functions act as an interface to Pyro and support the user when designing new effects. Some functions provide convenient, independence-aware wrappers for Pytorch or Pyro functions while some other constructs are entirely new. We introduced for example the classes CalipyTensor, CalipyDim, CalipyIndex that allow tensors to be associated with dimensions, which subsequently can be declared independent, subjected to subsampling, or processed by further functions. This way of keeping track of dimensions and employing them to inform functions during processing is similar to the functorch.torchdims module for which we provide an interface to enable einops-type function declarations.

3.2 Implementation & Access

CaliPy depends on Pytorch for its autograd functionality and extensive library of differentiable functions, distributions and gradient-based optimizers. Pyro is used for its probabilistic primitives and overall inference capabilities with CaliPy's core classes

depending heavily on it. The functorch package provides functionality for handling dimensions while matplotlib and torchviz are needed for visualization. Python's contextlib, itertools, and copy modules are used for managing and manipulating context, size ranges, or to copy and load full models to enable exchange. CaliPy is offered under the terms of the prosperity public license that is free for non-commercial use (and permits a free trial period for commercial use); a commercial license is available, too. Contributing to the library by bringing up issues, fixing bugs, sharing models, or designing new effects is possible and encouraged. The source code for the library is available on GitHub at the url `https://github.com/atlasoptimization/calipy`. The library will be listed on the Python packaging index PyPI where it can be installed via `pip`. Currently, the library is still in the early stages with simulation and inference mostly functional but a limited amount of effects and possibilities to share models. Development goals for now include better tutorials and documentation as well as support for subbatching, which allows handling big datasets by processing them in small subsets.

4 Usage & Experiments

The following three toy examples have been chosen for their simplicity and showcase the difference between formulating and solving a model analytically and doing it with CaliPy. Results for both approaches coincide up to 1e-5. In all cases, CaliPy allows us to skip from the model directly to the solution and bypass the manual calculations necessary to enable LS estimation. Skeleton code highlights the key details of the model formulation; fully executable code with explanations and annotations can be found in the repository's example folder. [1]

To extend each of the three examples, we could introduce:

- parameters as random and unobserved
- non-Gaussian observations
- nonlinearities in parameters and observations.

Deriving a solution with LS would then induce extra effort; CaliPy handles the examples still effortlessly.

[1] https://github.com/atlasoptimization/calipy/tree/dev_branch/calipy/examples/engineering_geodesy

4.1 Example 1: Bias Estimation

Suppose for illustration purposes the task of estimating a constant bias θ in tape measurements from observations y of a rod with known length μ as illustrated below.

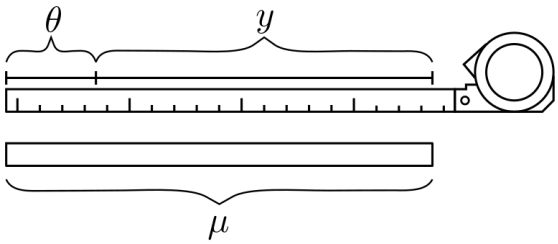

Figure 2. Illustration of observations made by a measurement tape with bias θ.

Since $y \approx \mu - \theta$, a probabilistic model of the form $y \sim \mathcal{N}(\mu - \theta, \sigma), y \in \mathbb{R}^{n_{\text{obs}}}$ seems appropriate, thereby declaring the observations to be normally distributed around expected value $\mu - \theta$ with standard deviation σ. We can infer an estimator for θ under the assumption of i.i.d. observations y in the following way via maximum likelihood:

$$\begin{aligned}\hat{\theta} &= \text{argmax}_\theta\, p(y) \\ &= \text{argmax}_\theta \prod_{k=1}^{n_{\text{obs}}} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_k - (\mu-\theta))^2}{2\sigma^2}\right) \\ &= \text{argmin}_\theta \sum_{k=1}^{n_{\text{obs}}} (y_k - (\mu-\theta))^2 \\ &= \frac{1}{n}\sum_{k=1}^{n_{\text{obs}}} (\mu - y_k)\end{aligned} \tag{3}$$

where the last few lines simply re-establish that maximizing Gaussian likelihood corresponds to minimizing a squared penalty which is achieved by the arithmetic mean. This result is neither hard to derive nor to guess. Nonetheless, in CaliPy, we can directly implement the probabilistic model $y \sim \mathcal{N}(\mu - \theta, \sigma)$, avoid any manual computation as in eq. 3 and have inference performed automatically via the following code:

```
...
theta_object = UnknownParameter(theta_ns)
noise_object = NoiseAddition(noise_ns)
...

class BiasProbModel(CalipyProbModel):
    ...
    def model(self, input_vars = None,
        observations = None):
        theta = theta_object.forward()
```

```
        output = noise_object.forward((mu - theta,
            sigma), observations = observations)
        return output
    ...
```

4.2 Example 2: Two-peg Test

Suppose the task is to perform a two-peg level test (Uren and Price, 2006, p.40) to estimate the collimation angle α by which a level's sight axis deviates from the horizontal plane. The leveling rods are placed at static positions $60m$ apart and multiple sets of height measurements are made for different positions of the level, see the illustration below.

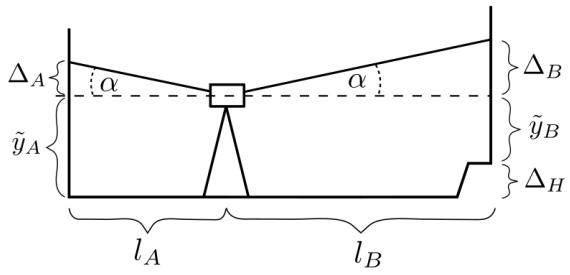

Figure 3. The two-peg test consists in observing height differences affected by a nonzero collimation angle α.

The relationship between the sketched quantities is given by:

$$y_A \approx \tilde{y}_A + \Delta_A, \qquad y_B \approx \tilde{y}_B + \Delta_B$$
$$\Delta_A = l_A \tan\alpha, \qquad \Delta_B = l_B \tan\alpha$$

If we encode n_{conf} different geometric configurations as input variables $x \in \mathbb{R}^{n_{\text{conf}}\times 2}, x_{k,\cdot} = [l_A^k, l_B^k] \in \mathbb{R}^{1\times 2}$ for the configurations $k = 1,\ldots n_{\text{conf}}$, we may formulate the probabilistic model

$$y(x) \sim \mathcal{N}(\tilde{y} + x\tan\alpha, \sigma) \tag{4}$$
$$y(x) = \begin{bmatrix} y_A^1 & y_B^1 \\ \vdots & \vdots \\ y_A^{n_{\text{conf}}} & y_B^{n_{\text{conf}}} \end{bmatrix} \tilde{y}(x) = \begin{bmatrix} \tilde{y}_A^1 & \tilde{y}_B^1 \\ \vdots & \vdots \\ \tilde{y}_A^{n_{\text{conf}}} & \tilde{y}_B^{n_{\text{conf}}} \end{bmatrix}$$

The classic two-peg test uses the two configurations $x_{1,\cdot} = [30m, 30m]$ and $x_{2\cdot} = [0m, 60m]$ which allows the observations $y \in \mathbb{R}^{2\times 2}$ to be converted directly into an estimation of α by means of the following reformulations.

$$y_A^k - y_B^k \sim \mathcal{N}(\Delta H + (l_A - l_B)\tan\alpha, \sigma_\Delta)$$

with $\sigma_\Delta = \sqrt{2}\sigma$. This implies expected values of ΔH and $\Delta H - 60m\tan\alpha$ for $y_A^1 - y_B^1$ and $y_A^2 - y_B^2$ respectively and determines the estimators $\widehat{\Delta H}, \widehat{\tan\alpha}$ that maximize $p(y_A - y_B)$ as

$$\text{argmax}_{\Delta H, \tan\alpha}\, p(y_A^1 - y_B^1)\, p(y_A^2 - y_B^2).$$

This can be simplified by resolving the Gaussian densities similar to what was done in eq. 3 and yields

$$\text{argmin}_{\Delta H, \tan\alpha}(y_A^1 - y_B^1 - \Delta H)^2 + (y_A^2 - y_B^2 - \Delta H + 60m\tan\alpha)^2.$$

Subsequently, $\widehat{\Delta H} = y_A^1 - y_B^1$ and $\widehat{\tan\alpha} = (60m)^{-1}[\Delta H - (y_A^2 - y_B^2)]$ as this achieves a value of 0 for the squared penalty. The best guess for α is then $\hat{\alpha} = \text{atan}(\widehat{\tan\alpha})$. Note that the results of this computation would be non-trivially different in case we had more configurations, measurements, or configuration-dependent accuracies. In CaliPy, we can completely skip the manual computation and let inferences be computed automatically.

```
...
alpha_object = UnknownParameter(alpha_ns)    # scalar
ytilde_object = UnknownParameter(ytilde_ns) # [n,2]
noise_object = NoiseAddition(noise_ns)
...

class LevelProbModel(CalipyProbModel):
    ...
    def model(self, input_vars, observations = None):
        alpha = alpha_object.forward()
        ytilde = ytilde_object.forward()
        output = noise_object.forward((ytilde +
            input_vars* torch.tan(alpha), sigma),
            observations = observations)
        return output
    ...
```

The code above directly implements eq. 4; inference can be performed via `level_prob_model.train(data)` and the results coincide with the analytical solution..

4.3 Example 3: Axis Errors

Suppose the task is to estimate the collimation error c and the trunnion axis error i of a total station, two axis misalignments that impact the horizontal angle measurement ϕ_{obs}. The relevant geometric configurations are presented in fig. 4.

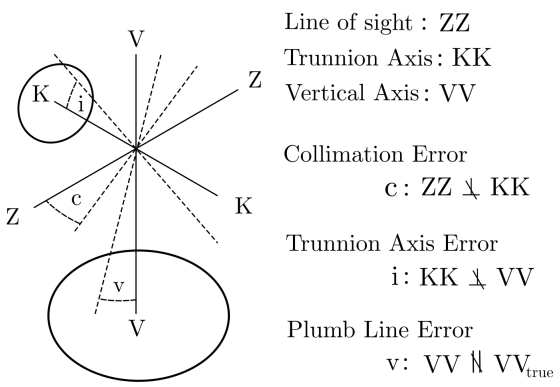

Figure 4. Typical axis misalignments in total stations.

The relationship between observed horizontal angles ϕ_{obs} and true horizontal angles $\tilde{\phi}$ as well as elevation angles β is in principle nonlinear but can be approximated linearly for small error magnitudes according to (Deumlich, 1980, pp. 132 - 134).

$$\begin{aligned} \phi_{obs} &\sim \mathcal{N}(\mu_\phi, \sigma) \\ \mu_\phi &\approx \tilde{\phi} + \text{face}\,200\text{gon} \\ &\quad - \gamma_c + \text{face}\,2\gamma_c \\ &\quad - \gamma_i + \text{face}\,2\gamma_i \end{aligned} \tag{5}$$

Here, $\gamma_c = c/\cos\beta$ is the systematic impact of the collimation error c on ϕ_{obs}, $\gamma_i = i\tan\beta$ is the systematic impact of the trunnion axis error i on ϕ_{obs} and face $\in \{0,1\}$ is a binary variable indicating if measurements have occurred in the Face I (face $= 0$) or the Face II (face $= 1$) configuration. The usual approach to infer c and i consists in measuring first a point in the horizontal plane and a point far out of the horizontal plane in both faces. Since $\beta = 0$ implies $\gamma_i = 0$ and $\gamma_c = c$, the first pair of measurements allows inferring and correcting c upon which the second pair of measurements yields $\gamma_c = 0$ and $\gamma_i = i\tan\beta$. With $\gamma_i = (\phi_{obs}^{face2} - \phi_{obs}^{face1} - 200\text{gon})/2$, it is possible to solve for i.

Alternatively, we can directly implement the probabilistic model 5 for observed points at arbitrary elevation angles in CaliPy with the following code.

```
...
class AxisErrorProbModel(CalipyProbModel):
    ...
    def model(self, input_vars, observations = None):
        beta = input_vars['beta']
        face = input_vars['face']
        c = c_object.forward()
        i = i_object.forward()
        gamma_c = c / torch.cos(beta)
        gamma_i = i * torch.tan(beta)

        phi_tilde = phi_tilde_object.forward()
        mu_phi = phi_tilde + face*(torch.pi/2 +
            2*gamma_c + 2* gamma_i) - gamma_c - gamma_i

        output = noise_object.forward((mu_phi, sigma),
            observations = observations)
        return output
    ...
```

Upon instantiation, it can be solved in the usual way by calling the `.train()` method. Adopting the original nonlinear relationships is as easy as swapping `gamma_c = c/torch.cos(beta)` for `c_quot = torch.sin(c)/torch.cos(beta)` and `gamma_c = torch.asin(c_quot)`.

5 Discussion & Conclusion

We presented how CaliPy, a Python library for calibration, can be used to build and solve probabilistic instrument models that arise in the analysis of measurements. It extends the set of models solvable by classical Least Squares and allows probabilistic models featuring almost arbitrary differentiable functions and Python control flow by building on top of Pyro and PyTorch. Stochastic variational inference guarantees that even complicated nonlinear problems with immense amounts of data can be tackled. By ensuring a strict separation of concerns by splitting probabilistic models into separate effects that communicate via clearly defined interfaces, probabilistic models can be built conveniently.

Simple examples concerning the calibration of tape measures, levels, and total stations demonstrated the advantages of formulating and solving instrument models in CaliPy. Dimension-aware processing and automatic inference let the user focus on building forward models without worrying too much about the solution procedure. We aim to use the library for modeling the impact of environmental conditions on data gathered by Total Stations and TLS where the complex interplay between instrument and environment has proven hard to model and the amount of data complicates inference. Future work includes an increase in the amount of available effects and better support for sharing models. With this paper, the author hopes to provide an introduction to an extendable library that can be used for rapid prototyping of real-world-suitable instrument models both in research and in industry.

Acknowledgements

The author thanks Andreas Wieser, Zhaoyi Wang, Tomislav Medic and Nicholas Meyer for fruitful discussions on the topics of instrument models and calibration. CaliPy was developed at Atlas Optimization GmbH. The geodetic examples were worked out as part of the author's academic work at ETH Zurich.

References

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. (2018). Pyro: Deep Universal Probabilistic Programming. *arXiv e-prints*, page arXiv:1810.09538.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877.

Bollen, K. A. (1996). An alternative two stage least squares (2sls) estimator for latent variable equations. *Psychometrika*, 61(1):109–121.

Boyd, S. P. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, Cambridge.

Del Pia, A., Dey, S. S., and Molinaro, M. (2014). Mixed-integer Quadratic Programming is in NP. *arXiv e-prints*, page arXiv:1407.4798.

Deumlich, F. (1980). *Instrumentenkunde der Vermessungstechnik*. VEB Verlag fuer Bauwesen, Berlin, 7 edition.

Gelman, A., Vehtari, A., Simpson, D., Margossian, C. C., Carpenter, B., Yao, Y., Kennedy, L., Gabry, J., Bürkner, P.-C., and Modrák, M. (2020). Bayesian Workflow. *arXiv e-prints*, page arXiv:2011.01808.

Ghilani, C. D. and Wolf, P. R. (2006). *Adjustment Computations - Spatial Data Analysis*. John Wiley & Sons, New York.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*. Springer, Berlin, Heidelberg.

Henderson, L., Goodman, N. D., Tenenbaum, J. B., and Woodward, J. F. (2010). The structure and dynamics of scientific theories: A hierarchical bayesian perspective. *Philosophy of Science*, 77(2):172–200.

Kerekes, G. and Schwieger, V. (2020). Elementary error model applied to terrestrial laser scanning measurements: Study case arch dam kops. *Mathematics*, 8(4).

Kingma, D. P. and Welling, M. (2019). An introduction to variational autoencoders. *Found. Trends Mach. Learn.*, 12(4):307–392.

Lichti, D. D. (2010). Terrestrial laser scanner self-calibration: Correlation sources and their mitigation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 65(1):93–102.

Neitzel, F. (2021). *Stochastic Models for Geodesy and Geoinformation Science*. MDPI, Basel.

Niemeier, W. (2008). *Ausgleichungsrechnung - Statistische Auswertemethoden*. Walter de Gruyter, Berlin.

Nievergelt, Y. (2000). A tutorial history of least squares with applications to astronomy and geodesy. *Journal of Computational and Applied Mathematics*, 121(1):37–72.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

Phillips, S. D., Estler, W. T., Doiron, T., Eberhardt, K. R., and Levenson, M. S. (2001). A careful consideration of the calibration concept. *Journal of Research of the National Institute of Standards and Technology*, 106:10.

Schaffrin, B. (2020). Total least-squares collocation: An optimal estimation technique for the eiv-model with prior information. *Mathematics*, 8(6).

Uren, J. and Price, W. F. (2006). *Surveying for Engineers*. Palgrave Macmillan, Houndmills, Basingstoke, Hampshire.