HERA Memorandum #84: A Generalized Approach to Redundant Calibration with JAX

Matyas Molnar and Bojan Nikolic

Astrophysics Group, Cavendish Laboratory, University of Cambridge

Received 19th November 2020, updated 6th June 2022

Abstract

The Hydrogen Epoch of Reionization Array (HERA) relies on redundant calibration with redcal to calibrate its data, which assumes Gaussian noise statistics, linearizes the measurement equation and minimizes the χ^2. We show generalization of this maximum likelihood estimation (MLE) to non-Gaussian statistics and without the need for linearization, which can be achieved at good computational performance with very little programming effort by repurposing open-source libraries intended for machine learning (ML), in this case JAX. As an example, we show a comparison between Gaussian and Cauchy assumed noise distributions in the calibration of a sample HERA dataset, with the latter showing expected resilience to radio-frequency interference (RFI).

The code for the work described in this section can be found at `https://github.com/bnikolic/simpleredcal`. The example HERA data used throughout this memo was observed on Julian date (JD) 2458098.43869 and taken from the H1C_IDR2.2 dataset. We only look at the EE polarization.

In §1, we review redundant calibration and extend its MLE to a Cauchy model. In §2, we show how redundant calibration with other models can be significantly and easily sped up with JAX.

1 Robust redundant calibration

An array with regularly spaced antennas has many redundant visibilities that are sensitive to the same modes on the sky. Redundant calibration uses the fact that the true visibilities from redundant baselines are equal. Supposing there are no direction-dependent calibration effects, we therefore have a system of equations for all antenna pairs i and j:

$$V_{ij}^{\text{obs}}(\nu) = g_i(\nu) g_j^*(\nu) U_\alpha(\nu) + n_{ij}(\nu) \tag{1}$$

where $U_\alpha(\nu) = V(\mathbf{r}_i - \mathbf{r}_j)$, the visibility for the baseline vector $\mathbf{b}_{ij} = \mathbf{r}_i - \mathbf{r}_j$, corresponds to a redundant baseline set that we index by α, and n_{ij} is the noise.

For the planned full HERA array, there will be 331 elements in the hexagonal core, corresponding to $N_{\text{bl}} = 331(331-1)/2 = 54{,}615$ baselines. The core only has 630 unique baselines, which means that we have a non-linear system of 54,615 equations to determine the 630 true visibilities and 331 gains.

1.1 Relative calibration

With the redundant calibration prior, an MLE for the gains and true visibilities can be constructed by assuming a distribution for the observed visibility noise.

1.1.1 MLE under Gaussian noise

Assuming Gaussian uncorrelated noise with variance σ_{ij}^2, which is the expected noise from the receivers and the sky, through MLE considerations, the gains and true visibilities can be found by minimizing the following negative log-likelihood function [1]:

$$-\ln\Big(\mathcal{L}_{\text{rel}}^{G}\Big)(\nu) = \frac{1}{2}\sum_{\alpha}\sum_{\{i,j\}_\alpha}\ln\Big(2\pi\sigma_{ij}^2(\nu)\Big) + \frac{\Big|V_{ij}^{\text{obs}}(\nu) - g_i(\nu)g_j^*(\nu)U_\alpha(\nu)\Big|^2}{\sigma_{ij}^2(\nu)} \tag{2}$$

where $\{i,j\}_\alpha$ are sets of antennas that belong to baseline group α. This minimization is equivalent to minimizing the χ^2.

This non-linear least-squares optimization can be done independently between frequencies and time. Solving Eq. (2) has been the main focus of redundant calibration methods, with current efforts opting to linearize Eq. (1) for computational ease [2].

1.1.2 MLE under Cauchy noise

We can extend the MLE analysis to different distributions for the visibility noise, and in §2, we show how such computations can be done efficiently. As an example, we can assume a Cauchy distribution for the visibility noise, which has the median as its location parameter. This is a robust measure of central tendency and is used to reduce the effect of RFI. In Fig. 1, we fit both the Cauchy and Gaussian distributions to frequency cross-sections of redundant visibility amplitudes, for comparison. An extreme case with outliers is plotted in Fig. 2. In the presence of RFI, a Cauchy model is superior, however, in the majority of cases, a Gaussian model better represents the data.

If we assume Cauchy distributed data, the negative log-likelihood when solving for redundant baseline sets is given by

$$-\ln\Big(\mathcal{L}_{\text{rel}}^{C}\Big)(\nu) = \sum_{\alpha}\sum_{\{i,j\}_\alpha}\ln\big(\pi\gamma_{ij}(\nu)\big) + \ln\left(1 + \left(\frac{\Big|V_{ij}^{\text{obs}}(\nu) - g_i(\nu)g_j^*(\nu)U_\alpha(\nu)\Big|}{\gamma_{ij}(\nu)}\right)^2\right) \tag{3}$$

This MLE with Cauchy-distributed noise fully encapsulates the distribution of the data, without being distorted by outliers, and is the best median estimator of the data.

1.2 Constraining degeneracies

Relative calibration yields solutions with degeneracies that can be parameterized as four terms per frequency: an overall amplitude $A(\nu)$, an overall phase $\Delta(\nu)$, and two phase gradient components $\Delta_x(\nu)$ and $\Delta_y(\nu)$. The degenerate parameters can be calculated from a sky model in an absolute calibration step. Alternatively, we can solve for these degenerate parameters by calculating them directly from the $-\ln(\mathcal{L})$ by applying a few conditions [3]. This method, however, still ultimately needs to reference the sky to set the flux scale and phase centre.

Solving for these degenerate parameters becomes a constrained minimization problem (e.g. solvable with `scipy.optimize.minimize`). Such constraints, however, tend to slow down the solver and can sometimes produce erroneous results; classically, the parameterization of the problem can be reduced and the constraints implicitly added into the function being minimized. Below, we review the constrained minimization required to solve for the degeneracies from relative calibration, and in §2 we show how this computation can be significantly sped-up in JAX, without having to re-parameterize.

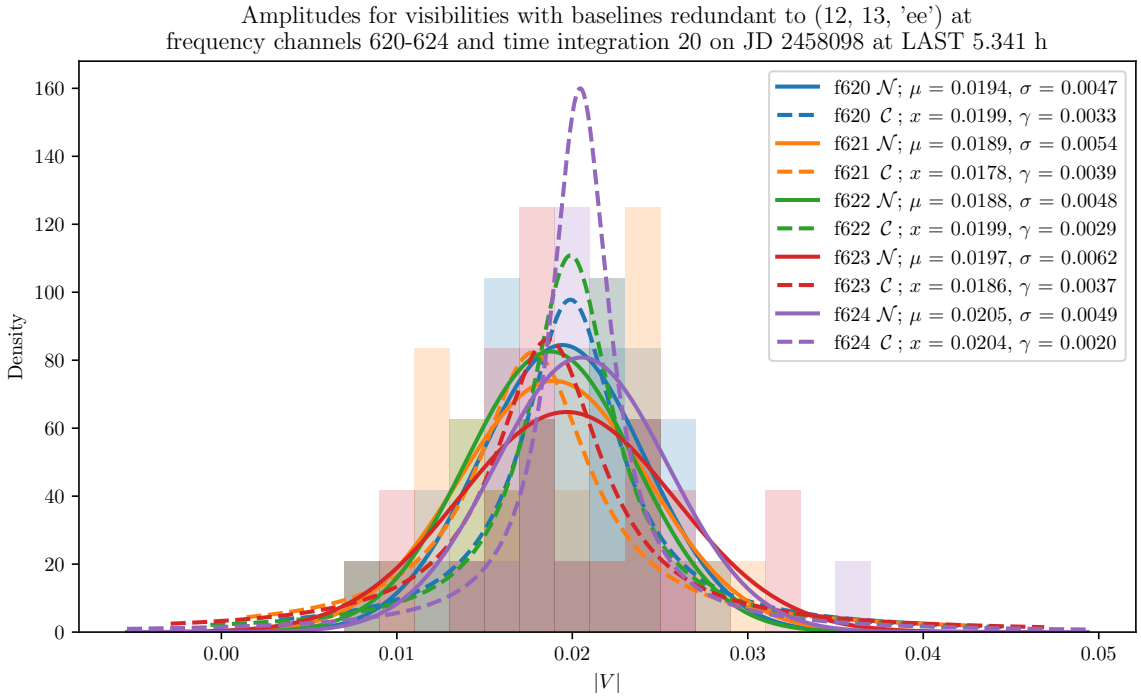

Figure 1: Histograms of visibility amplitudes for baselines redundant to $(12, 13, \mathrm{EE})$ for channels 620–624 at time integration 20, with Gaussian and Cauchy probability density functions fitted to these data. The Cauchy fits are sharper and have fatter tails than their Gaussian counterparts. The sample size is fairly small, with only 24 visibilities in this redundant set.

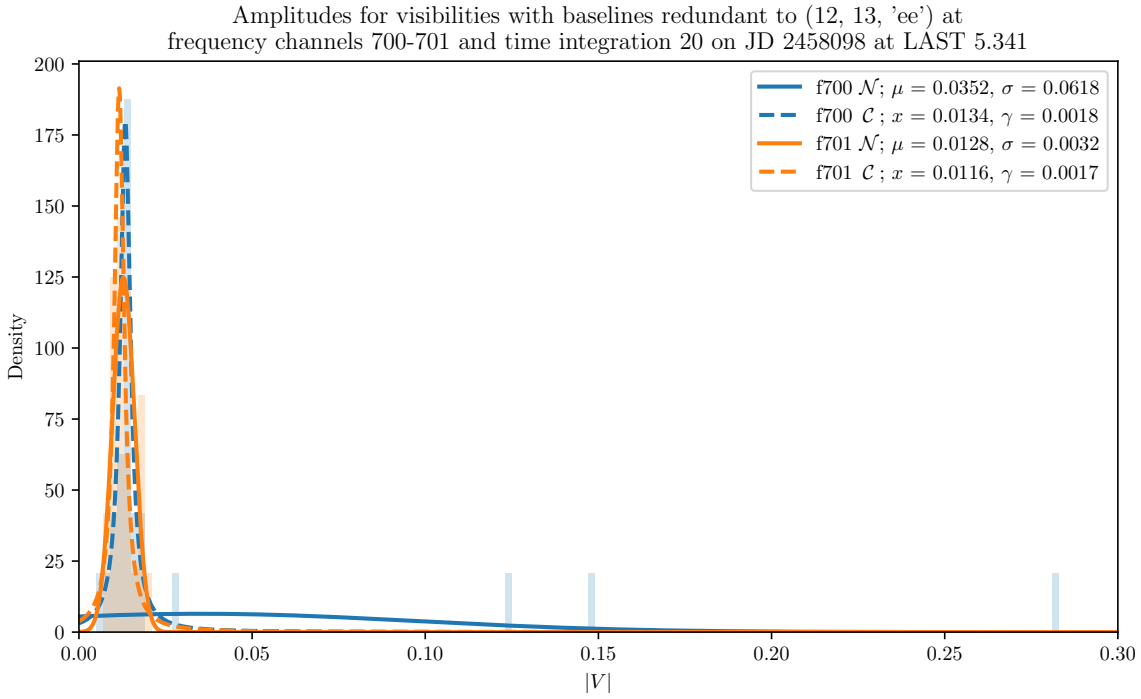

Figure 2: Same as in Fig. 1, but with frequency channels 700 & 701, the former of which contains outliers that greatly and inordinately affect the Gaussian fitting.

We first define a set of parameters h_i to be the gains that obey the following constraints:

$$\frac{1}{N}\sum_i^N |h_i| = 1 \quad \rightarrow \quad \text{mean gain amplitude of 1} \tag{4}$$

$$\frac{1}{N}\sum_i^N \text{Arg}(h_i) = 0 \quad \rightarrow \quad \text{mean gain phase of 0} \tag{5}$$

$$\left.\begin{array}{l} \sum_i^N x_i \text{Arg}(h_i) = 0 \\ \sum_i^N y_i \text{Arg}(h_i) = 0 \end{array}\right\} \quad \text{phase gradients of 0} \tag{6}$$

such that the antenna gains can be written as

$$g_i(\nu) = A(\nu) e^{i\left[\Delta(\nu) + \Delta_x(\nu) x_i + \Delta_y(\nu) y_i\right]} h_i(\nu) \tag{7}$$

where (x_i, y_i) is the position of antenna i, so that all degenerate dependencies are removed from h_i. We note that these constraints are arbitrary. Non-degenerate formulations of Eqs. (2) and (3) are therefore given by

$$-\ln\left(\mathcal{L}^G_{\text{constr}}\right)(\nu) = \frac{1}{2}\sum_\alpha \sum_{\{i,j\}_\alpha} \ln\left(2\pi\sigma^2_{ij}(\nu)\right) + \frac{\left|V^{\text{obs}}_{ij}(\nu) - h_i(\nu) h^*_j(\nu) W_\alpha(\nu)\right|^2}{\sigma^2_{ij}(\nu)} \tag{8}$$

$$-\ln\left(\mathcal{L}^C_{\text{constr}}\right)(\nu) = \sum_\alpha \sum_{\{i,j\}_\alpha} \ln\left(\pi\gamma_{ij}(\nu)\right) + \ln\left(1 + \left(\frac{\left|V^{\text{obs}}_{ij}(\nu) - h_i(\nu) h^*_j(\nu) W_\alpha(\nu)\right|}{\gamma_{ij}(\nu)}\right)^2\right) \tag{9}$$

where

$$W_\alpha(\nu) = A^2(\nu) e^{i\left[\Delta_x(\nu) x_\alpha + \Delta_y(\nu) y_\alpha\right]} U_\alpha \tag{10}$$

and (x_α, y_α) are the baseline coordinates of redundant set α.

The overall phase is also degenerate and is set by requiring that the phase of the gain of a reference antenna is null; it is an arbitrary convention with no physical significance.

1.3 Further work

The advantage of the Cauchy distribution in fitting for redundant baselines is not clear-cut: while it does reduce the impact of outliers, it only performs better than the Gaussian in a handful of cases. This is quantified through the negative log-likelihood when fitting the redundant visibilities at a given time for both distributions, as can be seen in Fig. 3. However, the use of the Cauchy could still be significant, especially if weak RFI is not picked up by flagging and the observations are integrated down. Further investigation is required to see if there is any merit in using the Cauchy distribution in such MLE computations.

Other examples of robust distributions that could be used to fit redundant sets of visibilities include the Student's t-distribution. The negative log-likelihood for the fitting of redundant visibility amplitudes using the t-distribution with degrees of freedom $\nu = 3$ is also shown in Fig. 3. The t-distribution is a trade-off (modulated by ν) between the Cauchy and Gaussian distributions.

2 JAX

Redundant calibration, as outlined in §1, is already computationally expensive. As we depart from Gaussianity, we can no longer use the Levenberg–Marquardt algorithm for non-linear least squares

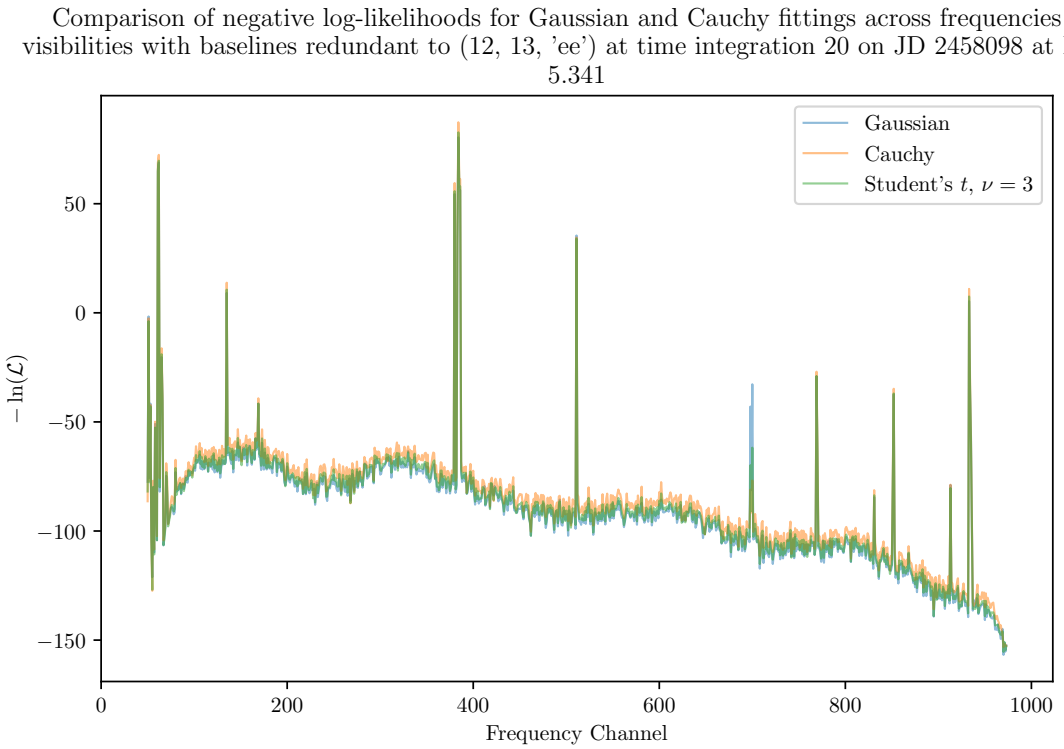

Figure 3: Negative log-likelihood from MLE fitting of Gaussian, Cauchy and t-distributions. It is clear that at channel 700, the Cauchy distribution provides a better fit to the data, as is also seen in Fig. 2.

problems, which further slows down calculations. In addition, if we wish to programmatically add constraints, such computations become even slower as to make them unfeasible.

We introduce JAX, a Google project that provides a framework for high-performance array computations on accelerated hardware by bringing together automatic differentiation (AD) and Accelerated Linear Algebra (XLA) to augment NumPy and Python code, therefore greatly speeding up operations common in ML. JAX has its own version of NumPy, which is (for most purposes) directly interchangeable with the original version[1]. Since JAX implements the NumPy API, it makes it straightforward to start working with JAX and to already significantly accelerate existing code with very few changes.

2.1 Automatic differentiation

AD is a powerful method to calculate derivatives used in many ML libraries (e.g. TensorFlow and PyTorch). It is preferable to more traditional methods, such as finite and symbolic differentiation, which perform poorly when applied to complex mathematical functions and when evaluating high derivatives [4]. AD refers to a general set of techniques that compute the derivatives of functions by exploiting the fact that computer programs ultimately execute a series of elementary arithmetic operations on elementary functions that can be combined with the chain rule to compute complicated derivatives of arbitrary order.

The established computational methods for differentiation are:

Symbolic differentiation: Here, algebraic expressions of functions can be differentiated by applying the rules for combined functions (e.g. linearity, product rule, chain rule etc.) to a database of basic functions (polynomials, trigonometric functions, exponentials and logarithms etc.) for which their derivatives are known. The resulting symbolic expressions, however, may grow exponentially large,

[1] JAX has its own version of SciPy too for that matter, although this is less complete than its NumPy peer.

producing unwieldy formulae that become inefficient to evaluate, in a problem known as *expression swell*.

Numerical differentiation: Also known as finite differences (FD), is the method that is based on the limit definition of the derivative. FD is computationally expensive, as it requires $O(n)$ evaluations of the function for a gradient in n dimensions. Moreover, truncation errors and round-off errors due to machine precision make FD ill-conditioned and unstable. This is what SciPy and other common libraries use by default.

Automatic differentiation: Much like symbolic differentiation, AD decomposes complex expressions through the systematic application of the chain rule into a finite sequence of primitive operations and elementary functions for which derivatives are known [5]. Unlike symbolic differentiation, however, the chain rule is applied to actual numerical values instead of symbolic expressions. With symbolic differentiation suffering from expression swell, and FD from precision errors, AD emerges as a powerful technique that yields exact results, unlike FD, at greater speed and memory efficiency than symbolic differentiation. See [6, 7] for detailed reviews of AD.

When computing the derivative of a function, a computer program will execute a sequence of internal operations, which are recorded in an *evaluation trace*. These form the foundation of AD. Reused subexpressions are algorithmically exploited, which leads to more efficient evaluation of functions and their derivatives.

Forward-mode: To compute $\frac{\partial f}{\partial x_1}$, we associate each intermediate variable v_i with

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} \tag{11}$$

Applying the chain rule to each elementary operation in the *primal* trace (which evaluates f), we then generate the corresponding *tangent* trace. Evaluating the primals v_i and their corresponding derivatives simultaneously and in order, gives us the required derivative in the output variable.

Reverse-mode: Derivatives are propagated backwards from a given output. This is a two-phase process, whereby all intermediate variables and their dependencies are initially populated, with their values stored in memory, in a forward phase. A second phase follows where derivatives are calculated by propagating *adjoints* $\bar{v}_i$ in reverse, from the outputs to the inputs, with each adjoint complementing their respective intermediate variable like so:

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i} \tag{12}$$

Reverse-mode has the advantage that it is considerably less expensive to evaluate than the forward-mode for functions with more inputs than outputs, whereas forward-mode is more efficient for a function with fewer inputs than outputs. For most purposes, reverse-mode is preferred, however, forward-mode is useful as it can be used to compute Hessians when combined with reverse-mode.

To get a better understanding of AD, we look at an example computational graph of JAX's intermediate representation (IR) to visualize its program tracing. We take the same example as in [4] - we refer the reader to Fig. 4 for a computational graph of the primal trace and Table 3 for the evaluations of the forward primal and reverse adjoint traces.

```
import jax.numpy as jnp
from jax import grad  # take derivatives with AD; built on reverse-mode

def f(x1, x2):
```

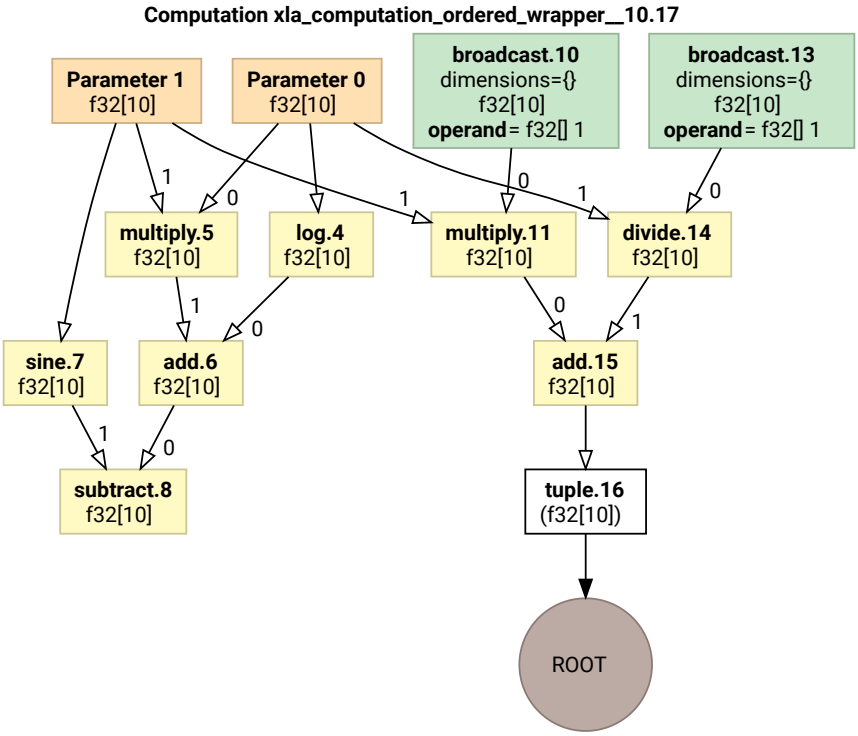

Figure 4: Computational graph from JAX's High-Level Optimizer (HLO) IR. The forward evaluation of the primals is in the left branch of the graph, while the reverse adjoint tangent operations are on the right. For a more complicated derivative, the computational graph would be deeper and the intermediate results from the primal trace would link into the reverse adjoint trace, thus further speeding up the computation.

```
        return jnp.log(x1) + x1*x2 - jnp.sin(x2)

grad_f1 = grad(f, argnums=0)  # derivative wrt x1
print(grad_f1(2.0, 5.0))
# 5.5
```

JAX's vectorization map `jax.vmap` can also be used to vectorize the above function, so that the output of `grad_f1` can be evaluated in parallel (over some axis) for some input array:

```
from jax import vmap

def gradv(x1, x2, argnum):
        return vmap(grad(f, argnums=argnum))(x1, x2)

grad_vf1 = gradv(jnp.arange(0, 10, dtype=float), \
                 jnp.arange(10, 20, dtype=float), 0)
print(grad_vf1)
#[ inf          12.          12.5         13.33333333  14.25
#  15.2         16.16666667  17.14285714  18.125       19.11111111 ]
```

The computational graph for the above code for vectorized gradient evaluation is shown in Fig. 4. These graphs, however, quickly swell up as the program becomes more complicated, and sifting through them becomes rapidly intractable.

2.2 JIT and XLA

JAX offers just-in-time (JIT) compilation via `jax.jit` (used as a decorator or explicit function), such that standard PYTHON and NumPy functions run efficiently with minimal coding overhead. While code performance using JIT is greatly improved when running on accelerators, there is also a noticeable difference on CPUs. With JIT, JAX will compile the function when it's called for the first time on the fly (or just-in-time) into a faster form, and will use the optimized version from the second call onwards.

JAX compiles functions with XLA, which is a domain-specific compiler for linear algebra created by TensorFlow. XLA analyzes the computational graph for a given program, specializes it for the actual runtime dimensions and types, fuses multiple operations into a single kernel, and outputs efficient native machine code, without having to write intermediate results to memory.

JIT compilation and AD in JAX can be composed arbitrarily[2], thus allowing for the creation of sophisticated programs that run at maximum performance, all while remaining in PYTHON.

2.3 Example and performance

To demonstrate the power of JAX, we perform relative redundant calibration with Cauchy noise (see §1.1.2) to show that with JAX, we can conduct MLE with considerable speed-up and ease (compared to pure NumPy and SciPy), even when moving away from Gaussianity and without the need to linearize or approximate. A full example notebook can be found at `SimpleRedCal.ipynb`.

We start by loading a HERA H1C_IDR2 dataset.

```
from jax import numpy as jnp
from simpleredcal.red_likelihood import doRelCal, group_data, relabelAnts
from simpleredcal.red_utils import find_flag_file, find_zen_file, get_bad_ants

# Select dataset to calibrate
JD = 2458098.43869
zen_fn = find_zen_file(JD)  # find path of dataset
bad_ants = get_bad_ants(zen_fn)  # get bad antennas from commissioning
flags_fn = find_flag_file(JD, 'first')  # import flags from firstcal

# Load dataset from uvh5 file to numpy array, with flagging applied
hdraw, RedG, cMData = group_data(zen_fn, pol='ee', chans=605, tints=0, \
                                 bad_ants=bad_ants, flag_path=flags_fn)
# 0 out of 741 data points flagged for visibility dataset
# zen.2458098.43869.HH.uvh5

cData = jnp.squeeze(cMData.filled())  # filled with nans for flags
no_ants = jnp.unique(RedG[:, 1:]).size  # number of antennas
no_unq_bls = jnp.unique(RedG[:, 0]).size  # number of redundant baselines
cRedG = relabelAnts(RedG)  # relabel antennas with consecutive numbering
```

We then perform relative redundant calibration

```
res_rel = doRelCal(cRedG, cData, no_unq_bls, no_ants, distribution='cauchy', \
                   coords='cartesian', bounded=False, norm_gains=True)
# Optimization terminated successfully.
```

[2] As well as vectorization (`jax.vmap`) and parallelization (`jax.pmap`)

	MLE noise assumption	Minimization method	Derivative information	JAX	NumPy	Speed-up
`doRedCal`	Cauchy	BFGS	Jacobian	2.57 s	9.27 s	×3.6
`doRedCal`	Gaussian	BFGS	Jacobian	3.55 s	10.3 s	×2.9
`doOptCal`	Cauchy	trust-constr	Hessian	9.65 s	110 s	×11
`doOptCal`	Gaussian	trust-constr	Hessian	9.53 s	156 s	×16

Table 1: Speed comparison for relative redundant calibration (§1.1) and the constraining of degeneracies (§1.2) with Cauchy and Gaussian noise assumptions, when run locally on a CPU with 4 cores.

All the magic happens in the `doRelCal` function. The negative log-likelihood is calculated in `relative_logLkl`, which is partially filled and JIT'd, and appears in `doRelCal` as

```
from jax import jit

ff = jit(functools.partial(relative_logLkl, credg, distribution, obsvis, \
                           no_unq_bls, coords))
```

We then seek to minimize the negative log-likelihood (Eq. (3)), which is done through the SciPy minimization:

```
from jax import jacfwd, jacrev

jac = jacrev(ff)  # Jacobian; reverse-mode faster for fewer outputs than inputs
hess = jacfwd(jacrev(ff))  # Hessian; forward-over-reverse is more efficient

res = minimize(ff, initp, bounds=bounds, method=method, \
               jac=jac, hess=hess, options={'maxiter':max_nit})
```

with the Jacobian being used for methods such as BFGS, L-BFGS-B and SLSQP, and the Hessian for trust-constr (these are the tried and tested minimization methods that we used). Recently, a JAX version of SciPy's `minimize` has been released, which can be found under `jax.scipy.optimize.minimize`. This solver, however, can only use the BFGS method for the time being.

The timings for the above setup, which performs relative redundant calibration using the unbounded BFGS minimization method, with and without JAX acceleration through both JIT and AD, are summarized in Table 1. Timings to constrain the degeneracies from relative calibration, which employs the trust-constr minimization method with `bounds` and `constraints` and also uses Hessian information, are also shown.

Caution is needed as to not mix and match JAX and pure NumPy, and to instead fully commit to one implementation, otherwise this may significantly increase runtime. The reason for this is that NumPy arrays are implemented in C, while JAX arrays are implemented in Python, with the latter only working efficiently with JAX functions.

The speed-up attained through JAX is remarkable, and only requires the ever so slight tweak in code; with these accelerated calculations, we can perform full redundant calibration without having to make any approximations or linearize the problem (as is done in redcal's `lincal` and `logcal`), and distributions other than a Gaussian can be used at no extra cost. As the problems scale up, we expect JAX to perform even better. We also expect even greater acceleration on GPUs or TPUs, however, this may need a bit of thought when coding up to ensure that the hardware is best utilized; this is currently being investigated.

2.4 Closing words

Libraries designed for ML can be used to accelerate redundant calibration. Through this increase in performance, and with the use of AD, we can generalize beyond Gaussian models and include constraints, with little added runtime.

We chose JAX for this research, as it has source-level compatibility, delivers both forward and reverse-mode AD (efficient for Hessian computation), and has an attractive functional framework and architecture that leverages from the Tensorflow ecosystem. With its simple and clean API (that is the same for CPU/GPU/TPU), JAX can be easily implemented into existing and new code to provide impressive performance out-of-the-box.

We are next looking at practical scientific impact of using non-Gaussian statistics for MLE estimation in calibration, in particular if it usefully improves resilience to RFI or instrumental artefacts over and above established flagging methods, and into integrating JAX with the `redcal` module.

References

[1] R. Byrne, M. F. Morales, B. Hazelton, W. Li, N. Barry, A. P. Beardsley, R. Joseph, J. Pober, I. Sullivan, and C. Trott, "Fundamental Limitations on the Calibration of Redundant 21 cm Cosmology Instruments and Implications for HERA and the SKA", ApJ, vol. 875, p. 70, Apr. 2019.

[2] J. S. Dillon and Hydrogen Epoch of Reionization Array (HERA) Collaboration, "Redundant-Baseline Calibration of the Hydrogen Epoch of Reioniztion Array", in *American Astronomical Society Meeting Abstracts*, ser. American Astronomical Society Meeting Abstracts, Jan. 2020, p. 153.08.

[3] R. Byrne, "HERA Memorandum #63: A Comparison of Two Absolute Calibration Methods", January 2019.

[4] A. G. Baydin, B. A. Pearlmutter, A. Andreyevich Radul, and J. M. Siskind, "Automatic Differentiation in Machine Learning: A Survey", *Journal of Machine Learning Research*, vol. 18, pp. 1–43, 2018.

[5] A. Verma, "An Introduction to Automatic Differentiation", *Current Science*, pp. 804–807, 2000.

[6] C. C. Margossian, "A Review of Automatic Differentiation and Its Efficient Implementation", *WIREs Data Mining and Knowledge Discovery*, vol. 9, p. e1305, March 2019.

[7] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, 2008.