Antenna Gain Calibration Implementation and Optimization in Pure-Python SKA-RASCIL (Postprint)

Authors: Li Jing, Cao Zhong, Wang Feng, Deng Hui, Mei Ying, Dai Wei

Date: 2020-09-23T00:00:00+00:00

Abstract

Antenna gain is a parameter used to measure the capability of an antenna to receive and transmit signals in a specified direction. With the construction of the Square Kilometre Array (SKA) drawing near, the Radio Astronomy Simulation, Calibration and Imaging Library (RASCIL), a Python-based software library, is currently under active development and refinement. High-performance implementation of antenna gain calibration algorithms constitutes an important component thereof. Based on the development work of the RASCIL antsol antenna gain calibration algorithm, this paper first discusses the fundamental principles of the antsol algorithm, subsequently analyzes and illustrates the implementation process based on pure Python, and further discusses optimization methods for the algorithm in the Python language on this foundation. Results demonstrate that the antsol algorithm implementation, after low-level optimization, can achieve very high performance. The optimized algorithm will be applied to SKA gain data calibration experiments, and the optimization methods introduced in this paper provide valuable reference for the development of other astronomical software.

Full Text

The Implementation and Optimization of Antenna Gain Calibration Based on Pure Python for SKA-RASCIL

Li Jing[1], Cao Zhong[1], Wang Feng[1], Deng Hui[1], Mei Ying[1], Dai Wei[2]

[1]Center for Astrophysics, School of Physics and Materials Science, Guangzhou University, Guangzhou, 510006, China
[2]Key Laboratory of Applications of Computer Technology of Yunnan Province, Kunming University of Science and Technology, Kunming, 650051, China

Abstract

Antenna gain is a key parameter that measures an antenna' s ability to receive and transmit signals in a specified direction. With the construction of the Square Kilometre Array (SKA) approaching, the Python-based Radio Astronomy Simulation, Calibration and Imaging Library (RASCIL) is currently under active development. A high-performance implementation of antenna gain calibration algorithms constitutes an important component of this effort. Based on our development work for the RASCIL antsol antenna gain calibration algorithm, this paper first discusses the fundamental principles of the antsol algorithm, then analyzes its implementation using pure Python, and further explores optimization methods for the algorithm in Python. The results demonstrate that the optimized antsol implementation can achieve very high performance. The optimized algorithm will be applied to SKA gain data calibration experiments, and the optimization methods introduced herein provide valuable reference for the development of other astronomical software packages.

Keywords: RASCIL; antsol; antenna gain; algorithm optimization

1. Introduction

Research frontiers in astronomy such as cosmic reionization, pulsars, and the search for extraterrestrial intelligence urgently require radio telescopes with higher sensitivity and resolution. The next-generation radio telescope—the Square Kilometre Array (SKA)—features ultra-high sensitivity and resolution thanks to its enormous signal collection area and thousand-kilometer baselines. Moreover, SKA will operate with nanosecond-level sampling frequencies, generating unprecedentedly large volumes of observational data. By combining high resolution, high dynamic range, and big data technologies, SKA will revolutionize astronomical research. High-quality imaging is essential for achieving SKA' s scientific objectives, and high-precision calibration is the prerequisite for realizing wide-field, high-dynamic-range imaging, representing a necessary process for transforming raw observational data into research-grade data. Our research group has previously conducted in-depth studies on MS file generation for SKA basic data formats [1] and wide-field imaging grid algorithms [2]. Calibration of extremely weak and complex systematic and environmental effects remains a critical challenge for current SKA research. To further improve imaging precision, antenna gain calibration has become an urgent research priority.

Radio interferometers obtain visibility data from target sources, which can be inverse Fourier transformed to produce sky brightness distribution maps [3]. Visibility data is output by the correlator of the antenna array. However, ionospheric variations, temperature fluctuations, ground reception, antenna blockage, noise from various electronic components, background temperature, and other factors introduce phase and amplitude errors in antennas, causing antenna gain variations that corrupt correlator outputs and severely compromise

the fidelity of visibility data. Therefore, antenna gain calibration of radio interferometer observations is essential to obtain authentic sky brightness distributions for subsequent astronomical research.

Current antenna gain calibration methods include standard calibration, iterative self-calibration, and radio source methods. Two calibration techniques are commonly used in imaging applications. One is "blind" iterative self-calibration, which adjusts parameters from initial estimates until the image results match a pre-established parameter model (typically a point source model). The other involves selecting a known, relatively stable strong point source in the sky for dedicated calibration observations [5].

Over the past decades, scientists have devoted considerable attention to phase and amplitude errors in radio interferometer imaging, achieving notable success in gain calibration algorithms [6]. In 1974, Rogers et al. [7] first applied closure phases to phase calibration in Very Long Baseline Interferometry (VLBI). Subsequently, Fort and Yee [8] and Readhead and Wilkinson [9] further developed gain calibration algorithms based on Rogers' work. However, these algorithms had significant limitations, being effective only for radio interferometers with relatively few telescopes and requiring identical signal-to-noise ratios across different baselines. Schwab [10] therefore proposed a more generally applicable algorithm. Later, Cornwell and Wilkinson [11] and Thompson et al. [3] improved and refined gain calibration algorithms, applying them to data processing for the Multi-Telescope Radio-Linked Interferometer (MTRLI) and the Very Large Array (VLA), respectively. Due to its robust performance, the antsol algorithm has remained in use for many years in GMRT (Giant Metrewave Radio Telescope) and CASA (Common Astronomy Software Applications) and serves as the default algorithm in AIPS. In addition to the classic "antsol" algorithm, there is also a "Stefcal" algorithm [12] that has been implemented in several stages of the LOFAR (Low Frequency Array) calibration pipeline.

Antenna gain calibration for SKA is one of the key tasks for ultimately achieving high-dynamic-range imaging with SKA. During the development and refinement of RASCIL, the algorithmic implementation of antenna gain calibration represents an important module. This paper investigates the implementation of antenna gain calibration algorithms using Python, focusing on code implementation and optimization.

2. Basic Principles of the Antenna Gain Calibration Algorithm (antsol)

In aperture synthesis imaging, signals received by different antennas are combined and output through multiplication and time-averaging circuits that constitute the correlator. Each correlator output of an interferometer array represents the sum of antenna signals and system noise. For antenna pair p and q, the correlator output is:

$$V_{pq}^{\text{obs}} = g_p g_q^* V_{pq}^{\text{model}} + \epsilon_{pq}$$

where V_{pq}^{obs} denotes the visibility observed by the baseline formed by antennas p and q, V_{pq}^{model} represents the model visibility (typically a point source model), ϵ_{pq} is the additive noise for the baseline formed by antennas p and q, and g_p and g_q^* are the complex gain factor of antenna p and the complex conjugate of antenna q, respectively. For any antenna k, the gain can be expressed as:

$$g_k = A_k e^{-i\varphi_k}$$

where A_k represents the amplitude of the signal received by the antenna and φ_k denotes the phase based on the antenna. Therefore, the complex gain for antenna pair p, q is:

$$G_{pq} = g_p g_q^*$$

Given V_{pq}^{obs} and known model visibility V_{pq}^{model}, we can solve for the antenna complex gains g_p. Assuming that antenna complex gains are independent and follow a Gaussian probability density function (implying that real and imaginary parts are independent Gaussian random processes), we can estimate g_p by minimizing the function S:

$$S = \sum_{p \neq q} w_{pq} \left| V_{pq}^{\text{obs}} - G_{pq} V_{pq}^{\text{model}} \right|^2$$

where w_{pq} represents weight values. Taking partial derivatives of S with respect to the real part R_p and imaginary part I_p of the complex gains and setting them to zero, then combining real and imaginary components:

$$g_p = \frac{\sum_{q \neq p} w_{pq} V_{pq}^{\text{model}*} V_{pq}^{\text{obs}}}{\sum_{q \neq p} w_{pq} |g_q|^2 |V_{pq}^{\text{model}}|^2}$$

After calibration completion, residuals can be estimated as $r_{pq} = V_{pq}^{\text{obs}} - G_{pq}$. To minimize residuals, iterative methods are generally employed during calibration to improve precision.

3. Algorithm Implementation

As previously mentioned, the antsol algorithm has been implemented in various forms, including Fortran and C versions, which are not particularly difficult from a programming perspective. To maintain consistency with RASCIL's objectives, all code was developed using pure Python, with software packages limited to

commonly used Python libraries. The antenna gain calibration program is located in the RASCIL package under `processing_{components}/calibration` with the filename `solver.py`. All source code is available for download from GitLab.

SKA software design specifications require that code in the bridging phase be developed in Python. To enable distributed computing, SKA-SDHP code uniformly adopts DASK (https://www.dask.org) as the multi-task distributed computing framework. The Dask library can scale computations across multiple cores and even multiple machines. Therefore, in antenna gain calibration development, only corresponding modules are implemented without using multi-task parallel computing within the modules. Additionally, to ensure code readability, GPU parallelism is not considered at the current stage. Consequently, during RASCIL module development, to accelerate progress and ensure correctness, we directly adapted the earliest VLA source code, which employs the basic algorithm from Appendix 1 of Thompson' s classic text [3].

Considering SKA data processing requirements, the implementation comprehensively addresses two main processes in antenna gain calibration: calculating antenna gains (see Equation 5) and calculating residuals (see Equation 6). Descriptions of relevant functions in the algorithm implementation are provided in Table 1 .

Table 1. Description of functions implementing the antsol algorithm

Function Name	Description
solve_{gaintable}	Solve a gain table by fitting observed visibility to model visibility
solve_{{{antenna}}*{{gains}}*{{{itsubs}}_{{scalar}}}	Solve for antenna gains, x(antenna2, antenna1) = gain(antenna1) × conj(gain(antenna2))
solve_{{{antenna}}*{{gains}}*{{{itsubs}}_{{matrix}}}	Solve for antenna gains using full matrix expressions
solve_{{{antenna}}*{{gains}}*{{{itsubs}}_{{vector}}}	Solve for antenna gains using full vector expressions
gain_{{{substitution}}_{{matrix}}}	Gain calculation subfunction (matrix expressions), called in function 2
gain_{{{substitution}}_{{scalar}}}	Gain calculation subfunction (scalar expressions), called in function 3
gain_{{{substitution}}_{{vector}}}	Gain calculation subfunction (vector expressions), called in function 4

Function Name	Description
Solution_{{{residual}}_{{matrix}}}	Calculate residual across all baselines of gain for point source equivalent visibilities, results expressed in matrix form, called in function 5
Solution_{{{residual}}_{{vector}}}	Calculate residual across all baselines of gain for point source equivalent visibilities, results expressed in vector form, called in function 7

Due to space limitations, we illustrate with a typical implementation of the `solution_{{{residual}}_{{matrix}}}` function. The original code (omitted) follows standard Fortran style, so loops and assignment statements were simply modified according to Python syntax. The function parameters are: `gain` (calculated gain array), `x` (point source equivalent visibility array), and `xwt` (point source equivalent weight array). The function returns the residual array `f`, with specific code shown in Table 2 .

Table 2. Code for the solution_{{{residual}}_{{matrix}}} function

```
def solution_{{{residual}}_{{matrix}}}(gain, x, xwt):
    nants, _, nchan, nrec, _ = x.shape
    residual = numpy.zeros([nchan, nrec, nrec])
    sumwt = numpy.zeros([nchan, nrec, nrec])
    # Code for residual calculation begins below
    for ant1 in range(nants):      # First loop: antennas
        for ant2 in range(nants):  # Second loop: antennas
            for chan in range(nchan):  # Third loop: channels
                for rec1 in range(nrec):
                    for rec2 in range(nrec):
                        error = x[ant2, ant1, chan, rec2, rec1] - \
                                gain[ant1, chan, rec2, rec1] * numpy.conjugate(
                                gain[ant2, chan, rec2, rec1])
                        residual[chan, rec2, rec1] += (error * xwt[ant2, ant1, chan, rec2, r
                                numpy.conjugate(error)).real
                        sumwt[chan, rec2, rec1] += xwt[ant2, ant1, chan, rec2, rec1]
    residual[sumwt > 0.0] = numpy.sqrt(residual[sumwt > 0.0] / sumwt[sumwt > 0.0])
    residual[sumwt <= 0.0] = 0.0
    return residual
```

In the above code segment, five nested loops from line 8 onward process different antennas and channels, using `numpy.conjugate` for the conjugate function. Line 13 calculates the error value, and line 16 computes the residual. This coding style is consistent with Fortran or C programs, making the code easy to understand.

During development, this approach ensured rapid porting and implementation, with results meeting expectations.

However, subsequent testing revealed that this code exhibited very low runtime efficiency. Using PyCharm' s profiling functionality and a series of experiments, we analyzed the performance bottlenecks. The fundamental issue is that Python is an interpreted language. In compiled languages, multiple loops do not significantly impact overall performance, but in Python as an interpreted scripting language, this implementation pattern is extremely inefficient. Lines 13-18, which calculate error and residual accumulations for different antennas and channels through loops, were identified as the most time-consuming code during profiling.

Additionally, while Python' s fancy indexing technique is convenient in practice (as shown in lines 19-20), which in traditional C/C++ or Fortran programming would generally be implemented through loops with conditional statements, performance tests indicate that this fancy indexing technique is also inefficient in Python, representing the second most time-consuming part of the entire function.

4. Code Optimization Methods

Numerous methods exist for Python code performance optimization. However, SKA project-related software regulations mandate pure Python development and recommend using only commonly used Python packages, discouraging excessive third-party packages. This restricts the approach of developing in C/C++ and calling from Python. Therefore, to address the aforementioned performance issues, we further investigated Python code optimization and performance tuning. Through profiling of the above code, performance limitations were found primarily in two aspects: matrix calculations implemented in code, and fancy indexing in NumPy.

4.1 Matrix Calculation Tuning

Careful analysis of the above code, combined with Equations (5) and (6), reveals that the core requirement of the algorithm implementation is to multiply a matrix of antenna gains with the conjugate matrix of corresponding antennas. Therefore, to improve performance and meet SKA bridging phase data requirements, we must fully leverage Python' s language features.

NumPy is a crucial package in Python, providing extensive matrix operation functions. Developed in C/C++, NumPy theoretically offers far superior performance to pure script code. The NumPy package provides a series of array or matrix processing functions, including inner and outer products. We sequentially tested the performance of array calculation functions provided by NumPy, and the results showed that these built-in functions are far more efficient than Python script implementations. Clearly, the key to optimization lies in process-

ing the multi-antenna, multi-channel data that appears in Section 3 in a single operation, avoiding Python loops as much as possible.

Through experimentation, we ultimately employed NumPy' s `einsum` function. Einsum, short for Einstein summation convention, is a notation introduced by Einstein in 1916. Simply put, applying einsum eliminates explicit summation symbols from summation expressions. The greatest advantage of the `einsum` function is its ability to multiply or sum specific dimensions across several multi-dimensional arrays.

For radio interferometer data processing, which generally involves multiple dimensions such as antennas (or baselines), frequency channels, and polarization, `einsum` enables flexible matrix calculations including matrix transposition, multiplication, trace computation, tensor multiplication, and array summation as needed. Implementing these operations separately using `transpose`, `sum`, `trace`, `tensordot`, and other functions would be not only complex but also error-prone. Moreover, since the development team has optimized `einsum` multiple times, it generally offers good performance and represents a highly effective method. The challenge in developing with `einsum` lies in the developer' s need to thoroughly understand its computational principles and have a very clear grasp of the data processing procedures within arrays.

4.2 Fancy Indexing Optimization

Processing partial data in an array is a common programming requirement. In C/C++ or Fortran, a loop statement can efficiently process partial array data. However, in Python, loop statements are inefficient due to script interpretation. Through testing of relevant functions, we ultimately found that the `putmask` function is currently the fastest function in the NumPy package for replacing partial elements in arrays.

4.3 Final Implementation and Performance Comparison

After a series of improvements, the final implementation in RASCIL is shown in Table 3.

Table 3. Optimized code for the solution_{{{residual}}_{{matrix}}} function

```
def solution_{{{residual}}_{{matrix}}}(gain, x, xwt):
    nants, _, nchan, nrec, _ = x.shape
    n_{residual} = numpy.zeros([nchan, nrec, nrec])
    n_{sumwt} = numpy.zeros([nchan, nrec, nrec])
    n_{gain} = numpy.einsum('i...,j...->ij...', numpy.conjugate(gain), gain)
    n_{error} = numpy.conjugate(x - n_{gain})
    nn_{residual} = (n_{error} * xwt * numpy.conjugate(n_{error})).real
    n_{residual} = numpy.einsum('ijk...->k...', nn_{residual})
    n_{sumwt} = numpy.einsum('ijk...->k...', xwt)
```

```
    numpy.putmask(n_{residual}, n_{sumwt} > 0.0,
                  numpy.sqrt(n_{residual}[n_{sumwt} > 0.0] / n_{sumwt}[n_{sumwt} > 0.0]))
    numpy.putmask(n_{residual}, n_{sumwt} <= 0.0, 0.0)
    return n_{residual}
```

Line 6 calculates the product of a gain matrix and its conjugate matrix through `einsum`, and the error can be obtained through simple computation (see line 7). Lines 6, 9, and 10 are particularly representative, processing visibility data for all antennas in a single operation. The complete functionality is identical to that in Table 2. Meanwhile, lines 13-14 use the `putmask` function to replace the previous fancy indexing approach.

5. Performance Analysis and Discussion

After multiple rounds of optimization and code refinement, the computational performance of the optimized program has significantly improved. In a server environment (CentOS 7.8 operating system, Intel E5-2620 V4 CPU, and 128 GB memory), we simulated visibility functions for SKA1-LOW and conducted computational time tests using these visibility functions.

Table 4 lists the computational time comparisons before and after optimization. The pre-optimization times represent profiling results from the original code, while post-optimization times represent profiling results after optimizing matrix calculations and fancy indexing. The `gain_{{{substitution}}_{{matrix}}}` function achieved a speedup of approximately 100×, `Solution_{{{residual}}_{{matrix}}}` about 250×, and `Solution_{{{residual}}_{{vector}}}` about 500×. The optimized code meets SKA bridging phase performance requirements. Through performance comparisons between pre- and post-optimization code, we gained deeper insights into Python-based astronomical software development:

1. Do not rely on traditional C/C++ or Fortran programming foundations when developing Python code; instead, transition from C-style or Fortran-style programming to genuine Python-style programming.
2. Thoroughly analyze code implementation patterns and maximize the use of built-in NumPy functions. Since different NumPy functions exhibit performance differences, prior testing of various functions is necessary when performance is critical.
3. NumPy is developed purely in C/C++. Under ideal circumstances, well-optimized Python code can achieve performance comparable to C/C++ implementations. Due to current SKA code development specification constraints, this paper does not discuss direct C/C++ implementation or just-in-time compilation using Numba. However, our results demonstrate that optimized Python code can achieve performance comparable to C/C++, meeting performance requirements while maintaining code readability.

4. This paper does not compare Python implementations with traditional Fortran or C/C++ implementations, as this aspect involves extensive related research. Generally, C/C++ code can be dozens or even hundreds of times more efficient than Python programs. However, based on our optimization results, the computational time of optimized Python code is essentially comparable to pure C/C++ code.

Table 4. Time consumption comparison before and after optimization

Function Name	Time consumption before optimization	Time consumption after optimization
gain_{{{substitution}}_{{matrix}}}		
gain_{{{substitution}}_{{scalar}}}		
gain_{{{substitution}}_{{vector}}}		
Solution_{{{residual}}_{{matrix}}}		
Solution_{{{residual}}_{{vector}}}		

This paper first analyzed the basic principles of the antsol antenna gain calibration algorithm and the porting process from standard C/C++ or Fortran code, then proposed optimization methods for the Python implementation. Finally, we conducted experiments and comparisons of algorithm execution time before and after optimization. The code implementation results demonstrate that the optimized Python algorithm can achieve performance comparable to traditional Fortran implementations. The optimized algorithm will be applied to SKA gain data experiments, and the optimization methods introduced in this paper provide valuable reference for the development of other astronomical software.

References

[1] Sun Haomin, Deng Hui, Mei Ying, et al. A method for generating radio measurement set files based on Python-casacore[J]. Astronomical Research & Technology, 2020, 17(02): 210-216.

[2] Yu Xiaoyu, Deng Hui, Mei Ying, et al. Research on optimal empirical values of w-plane in wide-field imaging grid algorithm[J]. Astronomical Research & Technology, 2019, 16(02): 218-224.

[3] THOMPSON A, D' ADDARIO L. Frequency response of a synthesis array: Performance limitations and design tolerances[J]. Radio Science, 1982, 17(02): 357-69.

[4] BHATNAGAR S: Tech. rep., National Centre for Radio Astrophysics, Pune, 1999.

[5] BOONSTRA A-J, VAN DER VEEN A-J. Gain calibration methods for radio telescope arrays[J]. IEEE Transactions on Signal Processing, 2003, 51(1): 25-38.

[6] WIERINGA M H. An investigation of the telescope based calibration methods 'redundancy' and 'self-cal' [J]. Experimental Astronomy, 1992, 2(4): 203-25.

[7] ROGERS A, HINTEREGGER H, WHITNEY A, et al. The structure of radio sources 3C 273B and 3C 84 deduced from the 'closure' phases and visibility amplitudes observed with three-element interferometers[J]. The Astrophysical Journal, 1974, 193: 293-301.

[8] FORT D, YEE H. A Method of Obtaining Brightness Distributions from Long Baseline Interferometry[J]. Astronomy and Astrophysics, 1976, 50: 19.

[9] READHEAD A, WILKINSON P. The mapping of compact radio sources from VLBI data[J]. The Astrophysical Journal, 1978, 223: 25-36.

[10] SCHWAB F. Robust solution for antenna gains[J]. Very Large Array (VLA) Scientific Memorandum, 1982, 136: 1-20.

[11] CORNWELL T, WILKINSON P. A new method for making maps with unstable radio interferometers[J]. Monthly Notices of the Royal Astronomical Society, 1981, 196(4): 1067-86.

[12] SALVINI S, WIJNHOLDS S J. StEFCal—An Alternating Direction Implicit method for fast full polarization array calibration; proceedings of the 2014 XXXIth URSI General Assembly and Scientific Symposium (URSI GASS), F, 2014[C]. IEEE.

Note: Figure translations are in progress. See original paper for figures.