ASEG 2026
CAIRNS

The GEMSModels Suite: Modernisation of the P223 Suite

David Annetts
Aegis Geophysics
david@aegisgeophysics.com.au

SUMMARY

The P223 Suite of electromagnetic modelling codes is an archetypical example of a successful research project. However, codes developed over the 27 years of the program and released to the public domain 18 years ago have not been maintained. Organisations continuing to use such codes accumulate technical debt. Algorithms implemented in these codes remain valid and therefore the suite is an excellent candidate for modernisation.

This Abstract describes the modernisation of two members of the P223 Suite. It describes the goals of the process and some of the benefits that have been gained as a result. The flexibility, ease of maintenance and greatly improved capability of one modernised code, `lyr`, demonstrates a proof of concept and will provide the basis for modernisation of selected remainders of the suite over time. The modernised suite of geophysical electromagnetic survey models is called `GEMSModels`.

Key words: Electromagnetic, numerical modelling, software, P223, modernisation, GEMSModels.

INTRODUCTION

The P223 Suite of electromagnetic (EM) modelling codes is an exemplary example of a successful research program. Over 27 years (Raiche et al., 2007), the project produced a series of forward and inverse EM modelling codes co-funded by CSIRO and industry through a series of AMIRA Consortia. Codes were used for a variety of purposes including planning surveys and interpreting EM survey data for mining, environmental and defence industries. In 2008, the code was placed into the public domain. The code's fate in the public domain was documented by Annetts and Cucuzza (2018) who concluded that although the project had been in the public domain for 10 years, it had yet to make the transition to a successful open-source project using the Subramaniam et al. (2009) criterion.

Lack of activity could be considered an impediment to code adoption. When codes are not developed, even within a small group, potential users may be reluctant to use them (Shipman and Randles, 2023). For P223 codes, there is an additional barrier. Although codes are advertised as Fortran 90/95 (F90/95), they are very much F77 in spirit. The F95 language offers many facilities, including modules and user-defined types, that aid development of large, complex codes such as the P223 Suite. Data can be encapsulated using modules which can be reused much like libraries. Modules were poorly implemented in the P223 suite, and the data structure most often used is the multi-dimensional array. None of the options to (a) use unmaintained, undeveloped software as is, (b) wrap that unmaintained and undeveloped software for use in more modern codes or (c) maintain and modify the old code are especially appealing as long-term prospects; ultimately, these only exacerbate technical debt (Brown, 2024). With no palatable avenue forward, the decision was made to completely rewrite the suite. This Abstract describes the modernisation process and some of its consequences.

WHY MODERNISE?

Some reasons to modernise have touched upon in the Introduction. The main reason to do so is that languages and practices evolve (Fowler and Beck, 2019). When a code does not evolve with them, it either accrues technical debt (Brown, 2024) or dies. Technical debt is essentially the cost of not maintaining code. It is useful to highlight three particular areas in which the P223 Suite have accrued technical debt.

Codes in the P223 Suite were driven by batch files or shell scripts depending upon the operating system. These copied an input file to a fixed file name, ran the program, then renamed the fixed output to avoid overwriting existing files. One consequence of this style of operation is codes are limited by directories, limiting the code's use in parametric modelling studies. Modifications to shell scripts to create file-based directories in which to perform work are possible (and trivial) but any variation of this style of operation imposes an extra burden when large files which need to be copied. A better solution is to operate programs from command lines, deriving file names from input.

The P223 Suite consisted of eight codes, four codes modelling ground surveys and four codes modelling airborne surveys. In theory, there should be a strong degree of source-code commonality between codes and to some extent, there is. Routines concerned with the process of transforming frequency-domain results to time-domain results are similar over the code base. However, there are cases where codes have the same name over different codes yet have different arguments. This raises questions around which version of the is the definitive version, and adds an extra maintenance burden. A better solution is to use single definitive versions of functions over

multiple code bases. Such a design, where many codes can take advantage of common functionality, perhaps through a library, greatly simplifies maintenance.

A third reason to modernise codes is based on the use of intrinsic functions. While the suite uses F90/95 array arithmetic, considerably greater use of intrinsic functions could be made. For example, array size inquiry functions (e.g. `size` and `shape`) have negligible performance penalty and negate the need to provide array sizes as function arguments, reducing clutter. Other intrinsic functions (eg. `dot_product`) have greater scalable performance than functions used by the P223 Suite. Favouring intrinsic functions over custom code has the additional benefit of reducing the amount of code that must be maintained.

THE MODERNISATION PROCESS

The modernisation process targets four clear goals: accuracy, improved functionality, maintainability, and documentation. From these four goals, extendability becomes straightforward.

A modernised code should be accurate. P223 Codes used single-precision arithmetic for the most part and higher precision (either double precision or `SELECTED_REAL_KIND(12, 80)`, which maps to double precision on 64-bit hardware) for critical calculations.

Modernised codes should be improved over the originals. Improvement might be considered a nebulous concept. This Abstract addresses whether modernised codes have greater functionality than the originals and whether modernised codes can accomplish the same tasks more simply.

It should be easier to maintain modernised codes than the originals. Ease of maintenance can be directly related to code complexity through cyclomatic complexity (McCabe, 1976) which uses branching as a proxy for complexity. Code containing nested `if` blocks is more complex than code that is designed to perform a single task with no branching. Figure 1 compares cyclomatic complexity for each of `Airbeo`'s subroutines. One qualitative scheme (Wikipedia, 2026) classifies over 34% of subroutines as having complexities of medium or higher; three routines were classified as unmaintainable.

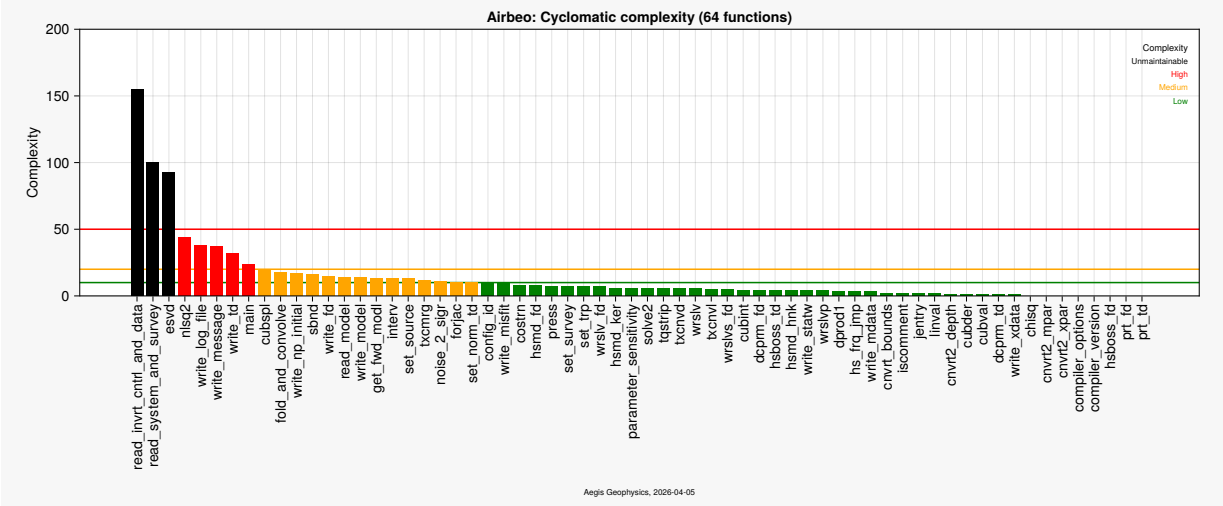

Figure 1. Cyclomatic complexity (McCabe, 1976) for Airbeo. This code is the simplest of the P223 codes, containing the fewest number of unmaintainable subroutines.

Finally, modernised codes should be documented. Documentation for P223 codes consisted of a long comment accompanied by some example files. Writing P223 input files required jumping between sections of this comment often missing requisite inputs leading to errors.

Julia (Bezanson et al., 2012) was chosen because it combines rapid development with high performance (Regier et al., 2019). It supports REPL-driven programming, has strong abstractions for scientific code, and compiles to efficient machine code through LLVM (Lattner and Adve, 2004). For technical computing, this matters. The language allows a fast iteration cycle without sacrificing execution speed. In practice, this means the rewrite can be both easier to develop and competitive in runtime performance with established compiled languages.

RESULTS

The modernised code `lyr`, the first component of the `GEMSModels` suite, demonstrates the practical benefits of the rewrite. It is designed for forward and inverse modelling of an *n*-layered earth and combines the capabilities of the legacy `Airbeo` and `Beowulf` codes. More importantly, it does so in a cleaner and more extensible form. Table 1 summarises the modernisation goals and shows that all were met: accuracy, improved functionality, maintainability, extendibility, and documentation.

Goal	Achieved?	Comments
Accurate	Yes	The default type is `Float64`; extended precision is `DoubleFloat64`. These types correspond to the IEEE 754 Fortran types `real64` and `real128` respectively. Julia's default integers are 64 bits contrasting with Fortran's 32 bits.
Improved	Yes	`lyr` is demonstrably more capable than either `Airbeo` or `Beowulf`. Some capability improvements are described in the text immediately following this Table.
Extendible	Yes	Use of user-defined types allow API's to be simplified and functionality to be improved with no real downstream consequences. In contrast, extending P223 codes required the addition of variables at each element of a call chain.
Documented	Yes	In contrast to a ReadMe file reproducing a long comment block at the start of each file, `lyr` is accompanied by a manual. Because the `lyr` input file is designed in terms of components, the input file can be constructed without jumping back and forth through this manual.
Maintainable	Yes	`lyr` has been developed with maintenance in mind. Figure 2 shows that functions are significantly simpler than P223 counterparts. Extensive use is made of intrinsic and standard library functions, further easing the maintenance burden and improving performance.

Table 1. Goals of the modernisation process. All goals have been met and in all cases, exceeded.

Two specific capability improvements are the ability to model multiple receiver types at each transmitter location and the ability to model multiple TEM systems in one run. The ability to be able to model multiple TEM systems in one run allows `lyr` to model multiple-moment prospecting systems. The program is driven from a command line, and additional file names are derived from the input file name. A third specific improvement is enabled though opinionated output. `lyr` Output is designed to be read, then filtered. Consequently, all results are always output and always in the same order. Indeed, a general principle of `lyr` is to provide as much information in output files as possible.

It is worthwhile expanding on earlier comments around input file construction in Table 1. The `lyr` input files were designed around components *viz.* the model excitation method, the survey configuration and the geophysical model. Consequences of this design are threefold. Firstly, input can be read using simple functions, greatly aiding debugging and data input. Secondly, there is a significant capability to cut and paste between different models. For example, an FEM excitation can be changed to a TEM excitation by swapping the excitation components. Moving-loop surveys can be changed to fixed-loop surveys using similar operations. It will be possible to easily convert layer-earth models to volumetric models. The capability of inverting forward model responses is greatly simplified when field and noise data files have exactly the same format as forward-model outputs. A third consequence of `lyr` input file design is that there is no backward compatibility with P223 codes. However, backward compatibility with the P223 Suite was never a design goal. Together, these changes lead to an improvement in productivity.

Expanding on earlier comments around maintainability and code simplicity, Figure 2 compares cyclomatic complexity between `lyr` and the P223 Suite. `lyr` Contains no functions classed as unmaintainable or highly complex. Over 93% of `lyr`'s routines are classified as simple. This represents a deliberate shift toward clarity, sustainability, and engineering discipline. Establishing and maintaining this level of simplicity creates a strong, reliable foundation for the ongoing P223 modernisation and future development.

CONCLUSIONS

Modernisation of the P223 Suite towards the `GEMSModels` Suite is an ongoing task. However, the modernised `lyr` is a significant improvement over equivalent P223 capabilities. A rewrite was justified by the accumulated technical debt and is successful because it preserves the scientific intent while improving maintainability and capability. Because the `GEMSModels` Suite was designed to be accurate, simple, maintainable and extendible, many components of `lyr` will form the basis of modernised `LeroiAir` and `Leroi` or `LokiAir` and `Loki` codes.

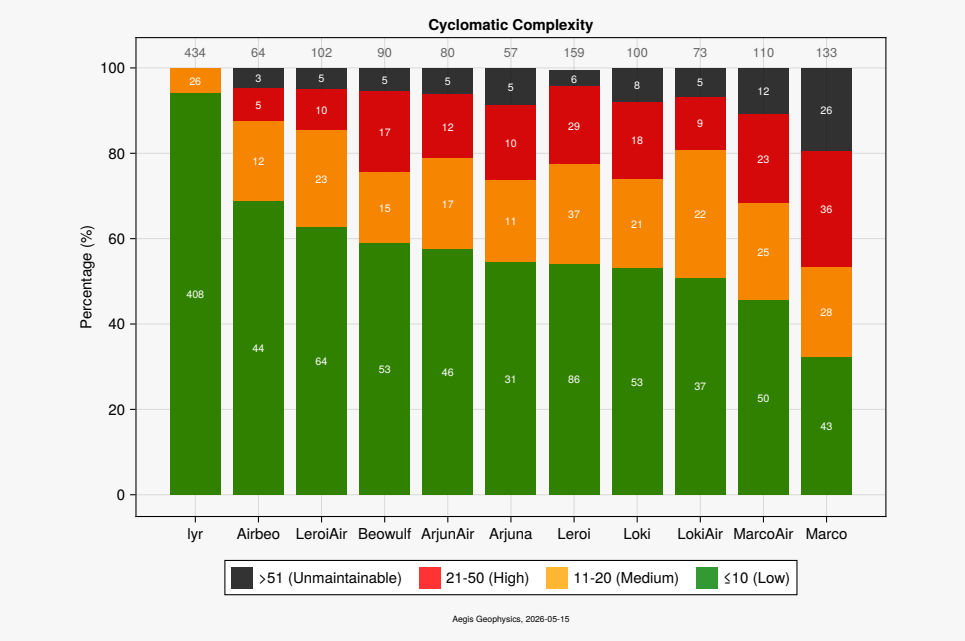

Figure 2. Comparison of cyclomatic complexity (McCabe, 1976) between `lyr` and each code in the P223 Suite. Numbers in each column reference the number of routines in each category while numbers at the top of each column indicate the number of routines in that code. Over 93% of `lyr`'s routines are categorised as simple and the code contains no unmaintainable routines.

ACKNOWLEDGMENTS

Concepts in this Abstract have evolved over many years of using and modifying various EM modelling codes. Critical to this evolution was time spent with various P223 projects, P223D as a student, P223E as a post-doctoral researcher, and post-P223F as support for the user base when codes were released to the general public. I thank Art Raiche for his support as P223's chief researcher. With project sponsors, he was primarily responsible for the code's original design. This is very much a case of 'standing on the shoulders of giants.'

REFERENCES

Annetts, D., and Cucuzza, J., 2018, Ten years in the wild: The P223 experiment: First Australasian Exploration Geoscience Conference, Sydney, NSW.

Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A., Stanford Exploration Project Report 2012, Julia: A Fast Dynamic Language for Technical Computing: arXiv:1209.5145.

Brown, D. A. R., 2024, Taming Your Dragon: Addressing Your Technical Debt: Apress, Berkeley, CA.

Fowler, M., and Beck, K., 2019, Refactoring: Improving the design of existing code:, The Addison-Wesley Signature Series Addison-Wesley, Boston Columbus New York San Francisco Amsterdam Cape Town Dubai London Munich, second edition edition.

Lattner, C., and Adve, V., 2004, LLVM: A compilation framework for lifelong program analysis & transformation: LLVM: A compilation framework for lifelong program analysis & transformation:, IEEE, International Symposium on Code Generation and Optimization, 2004. CGO 2004., 75–86.

McCabe, T., 1976, A Complexity Measure: IEEE Transactions on Software Engineering, **SE-2**, no. 4, 308–320.

Raiche, A., Wilson, G., and Sugeng, F., 2007, Practical 3D inversion: The P223F software suite: 19th ASEG International Geophysical Conference and Exhibition, Perth, WA, **131**, 68–69.

Regier, J., Fischer, K., Pamnany, K., Noack, A., Revels, J., Lam, M., Howard, S., Giordano, R., Schlegel, D., McAuliffe, J., Thomas, R., and Prabhat, 2019, Cataloging the visible universe through Bayesian inference in Julia at petascale: Journal of Parallel and Distributed Computing, **127**, 89–104.

Shipman, G., and Randles, T., An evaluation of risks associated with relying on Fortran for mission critical codes for the next 15 years:, Technical Report LA-UR-23-23992, 2023.

Subramaniam, C., Sen, R., and Nelson, M. L., 2009, Determinants of open source software project success: A longitudinal study: Decision Support Systems, **46**, no. 2, 576–585.

Wikipedia, Cyclomatic complexity:, Technical report, https://en.wikipedia.org/wiki/Cyclomatic_complexity, 2026.