The Future of GEMPACK

Florian Schiffmann

April 6, 2023

Abstract

The development of the GEMPACK software started in the early 80's. In its current form it is a versatile tool enabling CGE modelers around the world to implement and solve their models easily. Over the past decade, major improvements to the solution time of models have been made by reworking the self-contained math backend. However, hardly any new features that would require touching the core programming of GEMPACK have been implemented and in addition the time between releases keeps increasing. The reason for this is that it is becoming increasingly difficult to maintain and extend the old codebase. This phenomenon is known as 'software brittleness', where minor changes can break the software in unexpected ways.Therefore it was decided in early 2022 to perform a complete reimplementation of the current backend (tablo, gemsim, TG-program) in c++.

We use modern programming paradigms to create a more structured code that will facilitate maintenance and development speed for future features. The code design is modular, which will permit the use certain features of the GEMPACK software as lone standing libraries. On one hand this allows for shared functionality between the Delphi frontend and the backend code, thus avoiding duplicated code. One the other hand this structure opens the possibility to employ GEMPACK functionality from other programming languages such as python or R without the need to recode in the native language. While the rewrite will replicate most functionality, novel algorithms are used in the underlying code allowing for more versatility and faster execution times. With the current stage of the rewrite tablo parsing is 10-20 times faster than previously and gemsim style execution times are on par with current TG-programs. While hard to predict, we aim to have a fully functional prototype by the middle of 2023 and a release candidate middle 2024. We will present a progress report and hope to obtain user feedback on the ideas presented.

1 Introduction

The development of the GEMPACK software package started in the early 1980's. Over the past 40 years it has become a standard tool for CGE modelers next to GAMS. A key part of the success of GEMPACK is the design of the tablo language, making it particularly easy for economists to implement their model in an easy to understand way, closely resembling the underlying economic equations. At the same time GEMPACK manages to hide away the complex algebra and bookkeeping required to solve the models implemented in tablo.

Since its start, the GEMPACK source code has been contiguously extended supporting more and more advanced features in the tablo language as well as adding more elaborate solution techniques. In order to understand the scale of the development, it helps to look the timeline of features added to the software package.

- Release 4.2.02 April 1991
  - The first to look like modern GEMPACK. It included TABLO, TG-programs (though not yet GEMSIM), sparse solving techniques, Header Array files and multi-step calculations
- Release 5 starting from April 1994
  - Added GEMSIM
  - Added TABLO-like statements in CMF files
  - Set and element labeling on HAR files
  - Systematic sensitivity analysis
  - SET complements
  - Assertions
  - New sparse matrix solver
- Release 6 starting from October 1998
  - Set unions and intersections
  - Streamlined CMF files
  - Range checks for coefficients
- Release 7 starting from October 2000
  - Subtotals
  - Set Mappings
  - Shocks from HAR files
- Release 8 starting from October 2002
  - Complementarities
  - Tab file condensation statements
  - Homotopy
  - Product sets and projection mappings
- Release 9 starting from April 2005
  - Postsim processing
  - Sparse Header arrays
- Release 10 starting from May 2008
  - Improved reuse of pivots
  - Sparse HAR format

  - RAS matrix

- Release 11 starting from October 2011
  - Replaceable parameters in CMF
  - Index in set condition
  - Threading in LU solve

- Release 12 starting from 2018
  - New LU analysis
  - Rewritten gemsim evaluator
  - Automatic homogeneity testing
  - Runge-Kutta integrators

The list above is not complete but highlights the most important additions to the program over time. It does not show changes made to the structure of the underlying code (e.g. moving from Fortran77 to Fortran90, converting to freeform code,...). While all of these additions are useful, it can be seen, that the speed of development became slower and slower as time went on. Moreover, the newer features are very localized in scope, i.e. they are not touching the core programming of the code. The source of the problem is, that with each new implemented feature, the complexity of the source code grew and only minor adjustments to the underlying code structure were made to accommodate the increased complexity. The reason to avoid major adjustments of the code base was to maintain a fast development pace and allow for frequent new features as every adjustment to the programming logic requires a lot of time which does not produce tangible results and careful testing. In the early life of a software this strategy is not detrimental as the overall complexity is low. However as the software grows, this leads to software brittleness. According to Wikipedia: 'software brittleness is defined as the increased difficulty in fixing older software that may appear reliable, but actually fails badly when presented with unusual data or altered in a seemingly minor way'. Unfortunately, the backend of GEMPACK has reached the point, where this has become a major problem and several development ideas failed as the integration in the program proved too difficult (i.e. breaking seemingly unrelated parts of the code). Examples of these involve increasing the limit of the number of nonzeros in the LU to 64bit integers (a seemingly easy task), adding a aggregation function,... While it would be possible to work through all these problems the time required finding the linkage points and debugging the software would be very long compared to the impact of the added feature. Furthermore, even after testing we can not be sure, that no other 'hidden' connections exist which are not covered by our tests.

In the time from 2020 to 2021 an attempt was made to refactor the existing code and systematically reduce the complexity and step wise upgrade the source code. Unfortunately, the structure of the code proved so rigid, that certain parts of the refactoring could not be done in small (localized) steps but would have required the rewrite of large parts of code at the same time. These changes them self would have required even more adjustments in other places, such that in the end the refactoring was not possible without adjusting most of the existing

code at once. Even under the best circumstances this would be a very difficult task and limits the flexibility in the ways code can be modernized. Therefore, it was decided in 2022 to perform a complete rewrite of the backend from scratch.

2 General ideas of the rewrite

Our intention is, that the completed rewrite will have as little impact as possible on the user side. The main focus is to create a code that is easily extendable and will allow for rapid development speed again. In order to achieve this, we defined the following design goals:

The most important aim in the rewrite of the GEMPACK backend is to decouple the different components that are needed to interpret, compile and run the tablo language. In the current code global variables to connect all the different parts. While this approach simplifies the access to data structures, it creates hidden dependencies. I.e. changes to a global variable in one part of the program can have consequences in unrelated parts. In the rewrite we aim for an object oriented approach in which only the object itself can alter its data hence any modification to the data can easily be traced through the program.

The second part of the rewrite aims to have the program operate in a modular way. I.e. we attempt to minimize the connections between different parts of the backend. One such example would be that HAR file I/O will not rely on external data structures and operate independently of changes to the source code.

Thirdly, we aim for an object oriented approach where similar structures are subclasses from a common base class. The importance of this is, that many of the decision trees in the current source can be avoided clarifying the program flow and again compartmentalizing data access.

Finally, we want to maximize the reuse of functionality. In the current code many procedures exist in different versions depending on the details they are called with. This means, that changes to one of those routines need to be made in each and every of the related routines to ensure consistent behavior. Using function templates and polymorphism, such distinctions between routines performing near identical tasks can be completely avoided. We found that in order to achieve these goals, Fortran proved to be suboptimal as it's object oriented features are limited and templating is not intrinsic. Instead, we decided to perform the rewrite in c++ as an established programming language which provides all the required features and in addition has a vast variety of third party libraries (e.g. solver libraries, graph libraries,...).

None of this should have any impact on the workings as far as the user is concerned. However, these choices have implications on how the software can be used. The decoupling of functionality will permit to compile the code into independent libraries, that can be called from other programs. As the rewrite is done in c++ it is trivial to expose the API for certain parts of the code. This will permit, to make GEMPACK functionality available to other programs or programming languages such as R, Python,...

We aim to replace the very basic algebra processor in GEMPACK with a more powerful computer algebra system. When analyzing the output of GEMPACK (e.g. after substitutions) users often find lengthy statements. These emerge because GEMPACK current algebra processor only mechanically substitutes expressions without simplification e.g. GEMPACK produces (C +

1)$B/(CB+B)$ but could be simplified to 1 using a more elaborate CAS. Implementing such a more elaborate computer algebra system will provide a an easy way of increasing GEMPACKs performance. Moreover, as part of the CAS we aim to implement a way to identify common linear algebra operations (most importantly matrix multiplications) and execute those via high performance blas libraries which run up to 100 faster than the currently generated code.

One change that we are currently investigating, that will affect the way the program operates, is the use of a JIT compilation instead of using ahead of time compilation (ltg) for TG programs. This would effectively remove the need to distinguish between TG-program and gemsim as only a different evaluate function for the expressions will be called. Furthermore, at the current state of the rewrite shows massive time reductions in the parsing of the tablo code. With this speed up, there is little gain in requiring tablo as a separate step and the tablo functionality could be folded into the execution part of the program. Therefore, the final product could be a single executable working directly on tab files and performing simulations. The gain from the development side is, that we can get rid of communication layer (serialization of data) between tablo and gemsim/TG-program. The gain for the user side is faster execution speed and no waits for ltg when working on model development. A side benefit from this strategy has to do with computer security considerations. More and more organizations employer stricter and stricter software security policies. The most problematic policy for GEMPACK users is, to only permit the execution of whitelisted programs. The latter is especially problematic as each compilation of a model into a TG-program creates a new executable that would need whitelisting. As this is impractical, users will have to rely on gemsim or have are required to run their simulation on special computers not connected to the company network. If the JIT idea works as expected, these problems can be completely avoided.

3 Current State of the Rewrite

3.1 HAR I/O

So far we have implemented reading capabilities for HAR files, with writing capabilities in development. The HAR file library is completely independent of the rest of the code and can be used by any application. The library makes extensive use of templates to reduce the complexity due to the different datatypes used in input and output. In this way, a simple interface is obtained that has the form:

```
1    HARFile myHAR=HARFile(FileName,HARFile::ACCESS::READ);
2    auto metaInfo= myHAR.getMetaData(HeaderName);
3    auto it=myHAR.getDataIterator<outType>(HeaderName);
4    outType buffer[bufSize];
5    while(auto nRead=it->getNextData(buffer,bufSize)) {
6        //Do something with buffer
7    }
```

The template argument of the data iterator in line 3 needs to be a valid conversion from the data type on file. E.g. int can be converted to float, string data

can not be turned into float and will throw an exception at run time. It should be straightforward to adapt this simplistic interface to be interoperable with any outside system. In our case, this interface can be used by viewhar. In this way, any new types of header arrays (e.g. 64 bit integers or double precision data) won't require viewhar to implement its own new reader routines but simply link against the updated reader library.

3.2 Tablo Parser

The first step in parsing any programming language is tokenization. Beforehand, the tokenizer was hardwired to global data structures and relied on file input. We split the procedure into an abstract TabloInputStream class, which permits the implementation of different concrete classes to allow for input from different sources (e.g. file, character string or cmd input). The Tokenizer accepts any TabloInputStream and performs an on demand tokenization. In this way, the main window from analysGE (where user can type their own expression) could create an input stream of its own to directly interface with backend routines. This would greatly help to simplify analyseGE not having to maintain its own tablo parser and gemsim like evaluator.

The new tablo parser creates a scoped abstract syntax tree in which the statements are stored. All statements have their own class derived from a common base class which exposes all functions accessible from the outside. In this way, the details of the parser are fully contained and higher level routines do not need to care about the specifics.

Beforehand, GEMPACK used a state machine for the statement parsing. This works well, if only few states exist. Unfortunately over time the complexity of the language has grown making a handwritten state machine hard to understand. In the new algorithm we decided to split the parsing procedure into the preamble and the algebraic expression part. For the preamble we use a simple recursive descent parser which is extremely suited to the problem as it can be directly derived from the EBNF form of the tablo syntax. As soon as the recursive descent parser reach the point where algebraic expressions start we switch to an top down operator precedence parser (Pratt parser). This parser is well suited for the tablo language as the parsing logic is simple and its behavior can be coded to be statement driven as naturally all operators and terminals have their own class. Using this parsing algorithm the expressions are directly turned into an expression tree. This structure has many advantages compared to the original code in which operator and symbol stacks have been used. One key advantage that an expression tree is the natural form on which computer algebra systems operate.

As stated above, once the code is parsed and its syntax is checked, the statement and all its information is stored in the AST. The last step of the tablo parsing is the semantic checking (cross referencing statements for consistency). During this pass, we use the AST as a lookup table for all symbols required for the consistency check and store pointers to them for easy access. In the old code, a state machine was also used for the semantic checking. As stated above, this made the process very opaque and all possible checks had to be performed in situ when the state was reached. This led to very complex logic. In the new code, we split the checks by their nature and perform multiple passes through the statement/expression only testing for very specific properties. E.g.

- check that all symbols are defined before being used in this statement
- check that only permitted types are used (e.g. Mapping or Coefficients in Formulae).
- check that all indices are used correctly (i.e. index set matches coefficient declaration).

There are many more of these checks but as it can be seen, each of these checks is easy to write and it is easy to follow the checking logic used. Furthermore, if new features are introduced in tablo it is easy to add additional tests or integrate the existing tests into the new feature. Due to the use of the state machine, this piece wise checking was not possible in the old code.

Currently we have completed most of the tablo parser (not all checks are complete).

4 Computer Algebra System

We have started implementing a better computer algebra system working on our expression tree. So far we can perform basic simplification of algebraic expressions as well as take derivatives. While this does not sound too impressive, feeding in the produced condensed equations from the old code many duplications and simplifications are already recognized reducing the computational cost of evaluation. We are currently working on the linear algebra recognition and sum simplifications. Furthermore we are looking into data reuse, i.e. precomputation of results that can be reused. This is currently limited to single expressions but can potentially extended to identify common terms across the whole tablo code.

5 Gemsim and JIT Evaluator

In the old code the startup of the simulations required a preliminary pass (set evaluation) followed by the first formula pass. Due to data dependent sets, formulas were potentially calculated twice in this procedure. By creating a dependency graph from the abstract syntax tree and pruning it, we were able to provide an improved way of determining the necessary statements requiring evaluation in the different passes. Furthermore, using a dependency graph will potentially allow use to employ multiple cores at the same time to evaluate independent expressions simultaneously. From the top level of the code the evaluation sequence is a simple walk through the pruned dependency tree of interest (e.g. formula stage, update stage, equation stage, postsim stage,...)

The evaluators them self are currently work in progress but are close to completion. So far we developed an intermediate representation of the expressions, that can either be turned into a gemsim style evaluation or produce LLVM IR code for the use in JIT. We are currently working on completing the gemsim functionality for all operators and functions in the Tablo language as well as their LLVM IR counterpart. Once this is completed, we have to create the corresponding interface to be callable as part of the statement evaluate function. The key part here is to collect all the required pieces of data and provide a uniform interface that can be understood by both the JIT and the gemsim

evaluator. As soon as this is completed we will be able to produce the first benchmarks on data processing jobs.

The next stage of the evaluator will have to deal with submatrix processing for the equation stage. While the involved algebra is similar, the output needs to be placed in the correct position of the sparse representation of the LHS matrix. We are currently contemplating to evaluate the submatrices as dense vectors first and afterwards applying a series of sparse Kronecker products to sort the result into the LHS matrix representation. While this might appear as a potential overhead the increased memory locality of the operations should more than compensate for the additional computation.

6 Summary and Conclusions

The current state of the rewrite appears to fulfill many of the desired properties. A considerable amount of the development has been spent on developing the underlying data structures to be simple to use and flexible. This paid of in the rewriting effort when actually coding the tablo functionality. For example, developing the framework for the parser took more than 3 month. Implementing the syntax checks with the structures in place was done in less than 1 month. The fast pace was possible because the involved structures help minimize coding mistakes and the mistakes made were easy to spot. Seeing the ease with which all the different statement types, functions and operations could be implemented at this point is a good indicator that new feature will be equally easy to integrate.

As a preliminary result we can compare the speed of the tablo parser. We find that with the new data structures and algorithms, the speed is increased tenfold. While the speed of parsing might not be of great importance for user side applications, it opens possibilities for future developments. As stated above, the reduced time is short enough, that a distinction between tablo parsing and model execution is not crucial anymore and will allow a streamlined program flow for the user. A second avenue that could be explored is the integration of parsing in the tabmate editor. With the speed up, tabmate could run the tablo parser on a frequent bases (once a second or every two seconds) and provide real time highlighting of problems while developing new model code. The realtime highlighting of coding problems is a common feature in various programming IDE's and might prove convenient for GEMPACK users.

From the experience of working with the new code base, it looks like we are on the right track with the chosen program design. Many feature still need implementation, however the increased ease with which the currently available code supports the implemented extension is a positive indication for future additions. The benefit for the GEMPACK user base is, that new ideas and algorithms are implemented on the fly. Thus, GEMPACK development is not paused during the time of the rewrite. Once a release ready candidate is available (potentially only covering the most important subsets of features), we aim to make it available alongside the original version for user as an alternative. In this way, if a model only requires the implemented features, users will start benefiting from the new code.