A Methodology for porting CUTEst optimization problems to Julia

P. Nalluri, J. Hückelheim, P. Hovland, S. Narayanan, and M. Menickelly, 5 August 2021

Nonlinear programs (NLPs) are groups of equations designed to maximize or minimize the value of a given function. They are especially useful when modeling real-world problems in physics and engineering. Due to the complexity of these programs, researchers often rely on repositories of well-studied, pre-existing functions to hone NLP optimizers before applying them in research. One such repository, CUTEst (a Constrained and Unconstrained Testing Environment on steroids) [Adv. Mod. Opt. 10, 1 (2008)], contains three-thousand useful nonlinear programs. However, many of CUTEst's functions are written in older, obscure internal file formats, such as the Standard Input Format (SIF), and are dependent on a compiler in Fortran, a coding language no longer in popular use. CUTEst's dependency on its older SIF and Fortran components means it is unable to be used exclusively in a modern coding language. To streamline and modernize the process of using CUTEst, we formulated an approach to read the SIF files and develop a Julia implementation of these NLPs. Initially, we attempted to automate the process; however, due to unpredictable variation in the commands and syntax used across SIF files, coding an accurate parser quickly became time-consuming. Manually decoding the NLPs proved to be simpler – as every unexpected syntactic variation did not have to be integrated into the parser. Through manual decoding, the majority (80%) of the unconstrained nonlinear programs were hand-parsed. The remaining 20% were not decoded due to irregularities in their SIF files (undefined variables, non-SIF syntax, etc). In this work, a suite of CUTEst nonlinear programs was ported to a modern coding language, increasing the ease at which CUTEst functions can be applied to relevant computational research.

CUTEst (a Constrained and Unconstrained Testing Environment on steroids)[1,2] is a test set of nonlinear programs (NLPs) that is used to hone the accuracy and efficiency of optimizers. Because using CUTEst in Julia requires a Fortran compiler, Julia wrapper, and parsing an obscure internal file type (SIF) to read NLPs[3], developing a Julia module that can perform as comprehensively and well as the SIF-Fortran-wrapper infrastructure is key. This paper focuses on the progress achieved in creating the Julia module thus far and discusses future directions in which module development can proceed. Below are discussed the disadvantages of the SIF-Fortran-wrapper infrastructure, the process of decoding SIF files, sets of successfully and unsuccessfully ported CUTEst problems, features of the developed Julia module, directions for further work on the module, and the benefits of a pure-Julia CUTEst implementation.

I. AN INTRODUCTION TO THE CUTEST JULIAWRAPPER

A. Background: Julia wrapper use

The CUTEst problem set of nonlinear programs (NLPs) is accessible in Julia via the CUTEst.jl package. This package allows the user to call specific CUTEst NLPs and output their objective function value, gradient, and Hessian matrix at a particular input vector. In addition, the package is equipped with a selection tool that enables the user to sort problems by their type (constrained vs unconstrained, problem dimension, etc.).[4,3] Various other tools are included in the CUTEst.jl package; however, the capabilities mentioned above are among the most useful when testing the accuracy of optimizers. This is because many optimizers depend on objective function values, gradients, and Hessian matrices when determining the minimum or maximum feasible solution for a given nonlinear program. Moreover, an optimizer can be built to only work with certain types of problems (unconstrained problems, those with dimension below some $n \in \mathbb{R}$, etc.).

The internal process to call a CUTEst problem via the Julia interface is somewhat complicated. Once the user inputs a command to call a given NLP, the corresponding Standard Input Format (SIF) file must be decoded by a Fortran compiler. The compiler then provides the output associated with the inputted command.[4]

B. Disadvantages of using the CUTEst Julia wrapper

Though the Julia wrapper for CUTEst is much more convenient to use than the Fortran interface, various issues with the usability of CUTEst still exist.

Firstly, the SIF file type seems to only exist within the CUTEst test set and its previous generations (CUTE, CUTEr). Because it is an internal file type, there are few resources widely available that provide documentation for the file type.[4] Among these few is the SIF Reference Document written by the developers of CUTEst and its previous versions[5]; however, this documentation requires extensive study, given the unintuitive setup of SIF files. Moreover, the SIF Reference Document does not contain all the variations that can occur within a SIF file. Thus, once learning to read SIF files, a user must attempt to decode each function of interest. If a researcher is interested in the mathematical formula for an NLP, they therefore must spend an immense amount of time on simply decoding a SIF file, and still may be unsuccessful in doing so.

Additionally, the user is required to download a Fortran compiler in order to use CUTEst.jl, or any other interface to the CUTEst test set.[4] If the SIF files were ported to another language, the need for extra software to compile them would be obsolete. Moreover, Fortran is not only a fairly outdated programming language, but also only can operate in single and double precision – creating the possibility for error in its calculations.[4] Many modern programming languages, including Julia, can operate in arbitrary precision, which removes much of the error due to floating point arithmetic.

Given the issues that abound with the usability of CUTEst, it is pertinent to develop a version of CUTEst that operates entirely within Julia.

II. SIF DECODING METHODOLOGY

The background knowledge for decoding the SIF files came from the SIF Reference Document created by Conn, Gould, and Toint.[5] Once a significant amount of time was spent learning how to parse the SIF files, two methods were pursued to decode them: an automated parse or manual parsing. The former proved to be quite time-consuming, as considerable variation in the syntax and commands used within SIF files was present. Manual decoding, however, progressed quite rapidly – resulting in approximately 230 functions decoded in three weeks, as shown in Figures 1,2.

III. SUBSETS OF SUCCESSFULLY DECODED CUTEST PROBLEMS

A. Summary of results of decoding

Within the CUTEst set exist a total of 1490 nonlinear programs, 268 unconstrained and 1222 constrained.[2] Decoding the unconstrained problems was prioritized, as their SIF files were significantly shorter than their constrained counterparts. To determine whether the SIF file for a problem was decoded correctly, the problem was coded in Julia and unit tested against its counterpart sent through the Fortran compiler. As a result, 231 unconstrained NLPs were successfully decoded and ported to Julia.[6] The remaining 37 unconstrained problems contained various idiosyncrasies within or related to their SIF files, which made parsing them unsuccessful. The Julia CUTEst Ports GitHub repository contains all successful and unsuccessful ports of CUTEst unconstrained functions.

B. SIF file characteristics that prevented successful parsing

The characteristics that prevented parsing for the SIF files of these remaining 37 unconstrained problems are elaborated upon in Figure 3 and discussed below.

Approximately 13% of unconstrained NLPs matched their SIF files perfectly, but their outputs did not match those of their counterparts sent through the Fortran compiler. These discrepancies could have arisen if the methodology used to port functions to Julia was not comprehensive enough to cover some uncommon SIF command/syntax variations.

Another 5% of unconstrained problems contained additional Fortran code at the end of their SIF files. This code was not written in the SIF format, but rather used more general Fortran commands.[3] These problems were not translated be- cause an in-depth working knowledge of Fortran syntax would be required to transcribe the Fortran section to Julia. Moreover, these Fortran sections often called functions from other files – meaning that the NLPs with additional Fortran code were not entirely defined within their SIF files.[2] Thus, porting them to Julia with only their SIF files as data sources was not possible.

The remaining unconstrained NLPs that were unable to be decoded had a medley of errors. Approximately 1.5% had corresponding SIF files of size 300 KB, rendering manual decoding infeasible.[2] The last 0.3% of unconstrained NLPs contained undefined variables in the SIF files – meaning certain constants or parameters would be used without being assigned a numerical value.[2]

IV. FEATURES OF JULIACUTESTMODULE.JL

After decoding the SIF files, the Julia module JuliaCUTEstModule.jl was developed containing 231 successfully decoded Julia ports of SIF file functions.[6] The commands necessary to utilize the functionalities of the module are similar to those needed for the CUTEst Julia wrapper dependent on the SIF files and Fortran compiler.[2,7] This ensures a smooth transition between using the older Julia wrapper and the new Julia module for the user.

JuliaCUTEstModule.jl depends neither on the SIF files nor on a Fortran compiler. Instead, it contains each decoded NLP within a Julia function. The user can call an NLP to check whether it is included in the module, the size of its input vector, or its objective function value at a particular input vector. The commands to do so use similar syntax and input types to the older CUTEst Julia wrapper.[2,7,6]

V. FURTHER DIRECTIONS FOR JULIACUTESTMODULE.JL

Presently, JuliaCUTEstModule.jl contains a fraction of the CUTEst NLPs and only supports

some of the older SIF-Fortran-dependent Julia wrapper capabilities.[2,6] However, with further development, JuliaCUTEstModule.jl could become a functional replacement for the older Julia wrapper, CUTEst.jl.

Among the most useful capabilities of CUTEst.jl, were its features that computed the value of an objective function, gradient, and Hessian matrix at an inputted vector. The first of these is already supported by JuliaCUTEstModule.jl.[2] The latter two possibly could be achieved with automatic differentiation (AD), which tasks the computer with finding derivatives of various orders, rather than requiring the user to explicitly code computations for each derivative into every NLP function.

Another useful feature of CUTEst.jl is its "selection tool", which allows users to sort NLPs by type.[7] A version of this tool in JuliaCUTEstModule.jl may be possible to replicate, once more functions are successfully ported to Julia.

Lastly, decoding the remaining unconstrained and constrained NLPs will create a more comprehensive problem base for researchers to test optimizers. With the addition of the above functionalities, the native Julia JuliaCUTEstModule.jl implementation can become a full replacement of CUTEst.jl and the SIF-file/Fortran-code infrastructure on which it depends.

VI. ADVANTAGES OF A PURE-JULIA CUTEST IMPLEMENTATION

In addition to bypassing the SIF files and Fortran compiler with JuliaCUTEstModule.jl, other advantages of a pure-Julia implementation include the ability to compute in arbitrary precision and the capacity to expand mathematically past what is offered by the SIF files and CUTEst.jl. With a full native Julia CUTEst implementation, capabilities such as finding higher-order derivatives or linearly combining NLPs can be added to JuliaCUTEstModule.jl much more easily. Additionally, certain automatic differentiation tools that require explicit NLP Julia code can be tested and perfected on JuliaCUTEstModule.jl once it is fully developed. Lastly, a user can easily add new optimization problems to JuliaCUTEstModule.jl, streamlining the process of testing an optimizer by combining other test sets with CUTEst. This allows the user to develop their own broader test set, without having to code their own SIF files or being reliant on the release of a new generation of CUTEst.[6]

VII. CONCLUSION

In this work, 15% of nonlinear programs (NLPs) in the CUTEst test set have been ported to Julia. These NLPs are packaged into the Julia module JuliaCUTEstModule.jl, which is able to determine whether an NLP included in the module, the required size of its input vector, or its objective function value at a particular input vector.[6] Only some capabilities that the SIF-Fortran-dependent Julia wrapper CUTEst.jl supports have been replicated in JuliaCUTEstModule.jl[2,6]; however, with further development, such as computing gradients with automatic differentiation tools, the pure-Julia JuliaCUTEstModule.jl will be able to replace the SIF-Fortran-CUTEst.jl infrastructure. This will streamline the process of working with CUTEst NLPs and allow development of novel computational features unsupported by CUTEst.jl.

VIII. ACKNOWLEDGEMENTS

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships (SULI) program. The research was conducted at Argonne Nation Laboratory under the mentorship of Dr. Paul Hovland, Dr. Jan Hückelheim, Dr. Sri Hari Krishna Narayanan, Dr. Michel Schanen, and Dr. Matt Menickelly. Advice on scientific communication was given by Dr. Bob Boomsma.

IX. APPENDIX

Appendix: GitHub Repository with Developed Code

The below GitHub repository contains all the successfully decoded Julia ports for this work. The Julia module developed is under file name

JuliaCUTEstModule.jl. The link to the repository is https://github.com/panalluri/CUTEst-Julia-Ports.

X. REFERENCES

[1]A. Neculai, "An Unconstrained Optimization Test Functions Collection," Advanced Modeling and Optimization 10, 1–16 (2008).

[2]A. Siqueira et al, (2020), https://github.com/JuliaSmoothOptimizers/CUTEst.jl.

[3]N. Gould. et al, (2020), https://github.com/ralna/CUTEst.

[4]D. Orban, N. Gould and P. Toint, "CUTEst: A Constrained and Unconstrained Testing Environment with safe threads," Computational Optimization and Applications 60, 1–15 (2014).

[5]N. Gould, A. Conn and P. Toint, (2018), https://www.numerical.rl.ac.uk/lancelot/sif/sif.html.

[6]P. Nalluri, (2021), https://github.com/panalluri/CUTEst-Julia-Ports.

[7]A. S. et al, (2020), https://github.com/JuliaSmoothOptimizers/NLP Models.jl.

XI. FIGURES

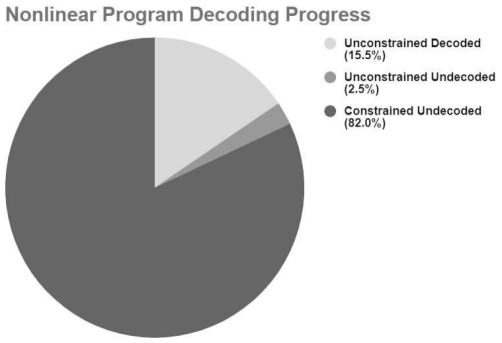

FIG. 1. Nonlinear Program Decoding Progress

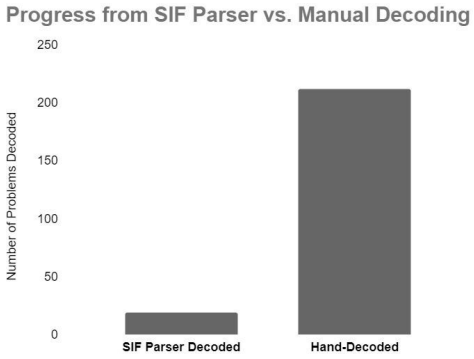

FIG. 2. Automated vs Manual Decoding Progress

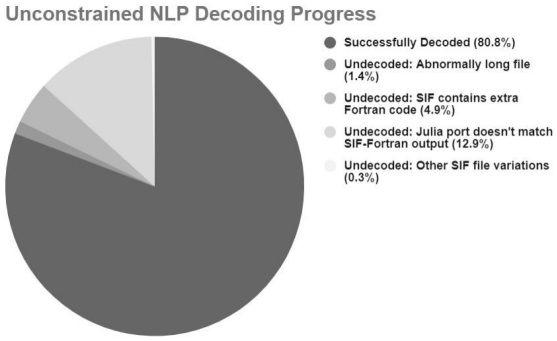

FIG. 3. Unconstrained Nonlinear Program Progress and Error Distribution