Fall 2018 ME459 Final Project Report
University of Wisconsin-Madison

Nonlinear Simultaneous Equations Solver Based on Broyden's Method

Dmitrii Turygin

December 18, 2018

Abstract

Regenerative heat exchanger design code requires to solve a system of nonlinear non-differentiable simultaneous equations. This functionality was implemented using nested monotonic equation solvers, which can handle only one function of one variable. The code was slow, difficult to maintain, scaled poorly and did not take into account the fact that multiple variables can affect the value of one function. The new algorithm uses Broyden's method and aims to overcome the issues of the original approach. However, variables in these equations are physical quantities and have physical limits, which makes it challenging to solve such system of equations.

Contents

1. Problem statement

As a research assistant at Solar Energy Lab, I worked on regenerative heat exchanger design code that calculates the physical dimensions and performance metrics of the regenerator based on its thermodynamic model and target parameters. Design code needs to solve a system of nonlinear non-differentiable simultaneous equations. Equation solver was originally implemented using nested monotonic equation solvers, which can handle only one function of one variable. System of equations was split into five separate equations of one variable by picking single variable that seemed to play the largest role. Those equations were then solved in a hierarchy. First, last four variables would be fixed, and first variable would be changed until the first equation was solved. Then, first variable was fixed, and second variable was changed once to try and get the value of a second function closer to its desired value. After that last four variables would be fixed again, and first variable would be adjusted until first equation was solved. The process would then slowly propagate, starting to affect third, fourth and fifth variables, until the whole system was solved.

First two issues are speed and scalability. Nested structure of monotonic equation solvers is inefficient, because solution is discarded 99% of the time. If a value of the fifth variable is wrong, all the extensive work of precisely solving four equations down the hierarchy is unnecessary, yet it is required. Moreover, monotonic equation solver requires two initial guess values to start the iterative solution process, which means that thermodynamic model must be called twice to evaluate all the functions. For every guess value, the solver one step down in the hierarchy needs two guess values of its own and so on. Just this preparation work requires 2^N calls to thermodynamic model, where N is the number of nested solvers. This has significant impact on speed and limits the practical number of equations in a system, which was the main problem that I ran into, as complexity of the model increased.

Second issue is robustness to initial guess values. Since at any given point only one equation is being solved and only one variable can be adjusted, incorrect guess values of one solver make it impossible to solve the equations down the hierarchy, because of physical limits of the variables. For example, diameter can be set so large, that even the smallest reasonable length produces conductance value that is too large.

Third issue is difficulty of code maintenance due to large overhead. Five solvers calling each other in a specific order, spread out variables and functions evaluations, ten guess values – all this makes it difficult to change order of the solvers or to introduce new ones.

To resolve problems of the original approach, algorithm using Broyden's method was implemented. Broyden's method provides an iterative way for finding roots of a system of nonlinear non-differentiable simultaneous equations. Thermodynamic model is complicated and relies on interpolated values from lookup tables, which makes obtaining a closed-form expression for a derivative of a function impossible. Broyden's method uses Jacobian to iteratively achieve successively better approximations of the roots. The functions can be of multiple variables, which would improve robustness to initial guess values. Moreover, Jacobian is not calculated at every iteration, but rather is successively improved, which cuts down on calls to thermodynamic model and improves the performance of the code.

2. Solution description

The general algorithm looks as follows [1]:

1) Calculate Jacobian J_0 using finite-difference approximation. Second order approximation was used.
$$f' = \frac{f(x - \Delta x) + f(x + \Delta x) - 2f(x)}{\Delta x^2}$$
2) Calculate J_n^{-1} .
3) Calculate x_{n+1} from x_n, $f(x_n)$ and J_n^{-1}.
$$x_{n+1} = x_n - J_n^{-1} f(x_n)$$
4) Update J_n using "good Broyden's method", knowing $\Delta x = x_{n+1} - x_n$, $\Delta f = f_{n+1} - f_n$.
$$J_{n+1} = J_n + \frac{\Delta f - J_n \Delta x}{\|\Delta x\|^2} \Delta x^T$$
5) Go to step 2.

The main issue with Broyden's method is that it tends to take large Δx steps that quickly drive x_{n+1} beyond reasonable limits. To combat this, first, function and variable values were normalized. Then, influence of certain variables on certain functions was artificially removed by setting appropriate entries in Jacobian to zero at steps one and four. Moreover, an intermediate step was added between steps three and four. If after step three x_{n+1} goes beyond reasonable limits, Jacobian is updated using finite-difference approximation from step one. Then x_{n+1} is recalculated and procedure continues.

Three functions were created, and one was rewritten in RegeneratorModel.cpp. Function setPoint() transfers entries from x_n vector into appropriate model variables and denormalizes them. Function evaluate() calls thermodynamic model to calculate values of functions at a specified x, normalizes them and stores them in f_n. Function jacobian() calculates Jacobian using second order finite-difference approximation of partial derivatives of functions. This function calculates partial derivatives only along the diagonal (x_i has the strongest influence on f_i) to improve stability on the first step. Function getDesignSolution() was rewritten and contains the algorithm itself. For matrix and vector algebra, C++ library called Eigen was used. C++ library called Spdlog was used to write data from test runs into CSV file.

The most difficult and time-consuming part was tuning the algorithm to achieve stability. I started with one variable and one function and slowly worked my way up, tweaking which variables can affect which functions, figuring out the best ways to normalize function and variable values, and changing variables. For example, originally, two functions $UA_{target} - UA_{calc}$ (conductance of the heat exchanger in kW/K) and $dP_{target} - dP_{calc}$ (pressure drop through heat exchanger kPa) depended and diameter and length respectively. This worked well when variables were changed one at a time, but for Broyden's methods $Volume$ and $Aspect\ Ratio = D/L$ are used instead to achieve better stability.

3. Overview of results. Demonstration of your project

Overall, results are promising. The new algorithm produces correct solutions. Figure 1 shows that for wide range of design targets algorithm converges within 80 iterations. In three cases, where the number of iterations is zero, the algorithm did not produce a solution. The new solver is much faster as can be seen in Table 1.

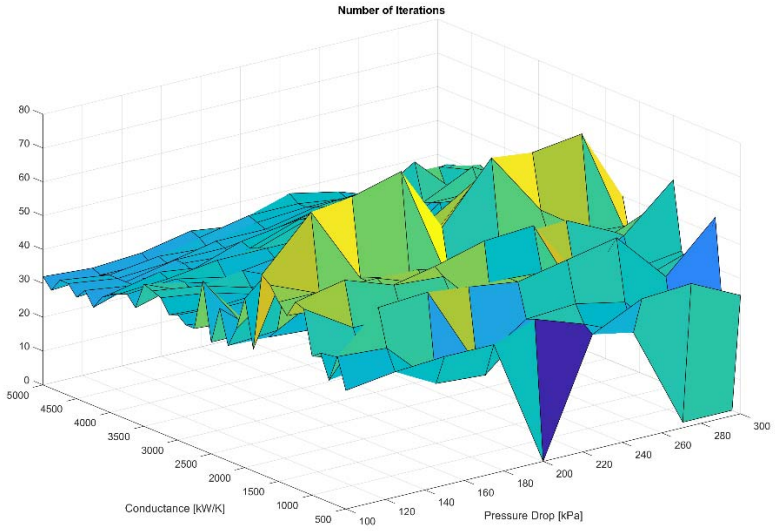

Figure 1. Performance of the solver based on Broyden's method for different design targets.

Table 1. Speed comparison (time to produce a solution).

	Average Time (ms)	Maximum Time (ms)
Monotonic equation solvers	6.2853	17
Broyden's method	0.2973	1

Algorithm is also fairly robust to changes in guess values, however there are plenty of cases when it does not converge.

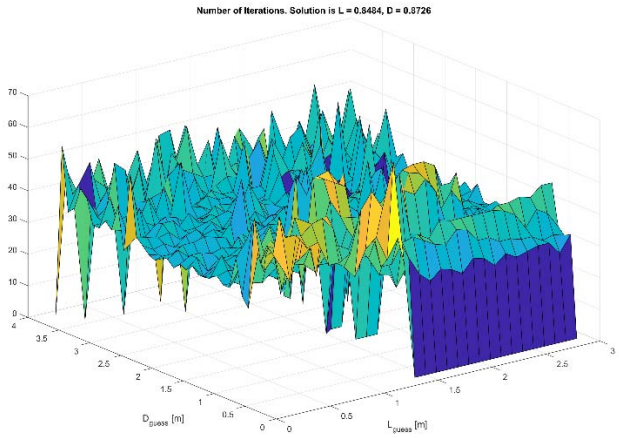

Figure 2. Robustness to initial guess values of the solver based on Broyden's method.

Unfortunately, Broyden's method is not as plug-and-play as was expected. Even after simplifying the problem down to five functions and five variables, and tuning the solver, Figure 2 shows that still there is much to be desired in terms of stability. Potential causes and solutions of stability issues will be discussed in Section 5.

Nevertheless, the new algorithm is very fast, and the gap between it and the old algorithm will only keep growing as the number of equations grows. Code is also much easier to maintain, and it has potential to be much more robust to initial guess values. Nested monotonic equation solvers algorithm is limited to certain aspect ratios.

4. Deliverables:

All the code is in my GitHub repository:

git clone https://github.com/tvdmitrii/ME459_Project_Regenerator.git

There are six branches, but master and develop are the most important ones. The other four are modifications that produce csv files used to construct plots and only serve the purpose of showing that I actually ran the tests. Master implements the original solver and develop implements the new solver based on Broyden's method. Documentation can be found in docs/html/annotated.html. I included into documentation only files that I wrote myself. All the code written specifically for this project is in develop in RegeneratorModel.cpp and main.cpp. Code can be compiled with CMake, which also moves required lookup table files into the build directory. Executable expects conductance in kW/K as the first parameter and pressure drop in kPa as the second. Reasonable range for conductance is 500-7000 and is 100-350 for pressure drop.
I did not write a lot of new code. Most of the time was spent making the solver work, so my commits are not spaced uniformly.

5. Conclusions and Future Work

Overall, Broyden's methods works well and addresses all the problems of the original implementation. However, stability of the algorithm needs to be improved. Algorithm tends to take very large steps and drive variables beyond physical limits. The steps are reasonable at first, but become large and rapid around convergence point, which tends to make diameter or length negative and break the thermodynamic model. I believe that the next step is adding "viscous" damping, which would be proportional to the change in x_n vector. It would help prevent such catastrophic changes and significantly improve stability. If this works out, I would want to make this solver a proper standalone class.

References

[1] *Broyden, C. G. (October 1965). "A Class of Methods for Solving Nonlinear Simultaneous Equations". Mathematics of Computation. American Mathematical Society.* ***19*** *(92): 577–593.*