Joshua Itwaru
jni2102
Professor Dan Rubenstein
Web/Mobile/App Programming
Term Project

App Name: Food Simplex
Code line count: ~2600

This project overview will be broken into multiple parts:

1) Project purpose/overview
2) Mathematical/Algorithmic overview and analysis
3) Technical/Architectural Overview
4) Program execution from beginning to end
5) Personal Experience

Project Purpose/Overview

You're on a strict bodybuilding regimen and that means, in addition to having strict rules in the gym, you have strict rules in the kitchen. All of your macros for every meal are planned out carefully and to the dot – for lunch, you must have exactly 500 calories, 45 grams of protein, 30 grams of crabs, and 7 grams of fat – no way around it. The problem is, given the food you have in your fridge, how do you figure out what proportions of each food you can eat to achieve that specific macro set? It might seems like simple mathematical process, but it turns out to be fairly complicated and computationally demanding. The Food Simplex app solves this problem and tells you what proportions of some given foods you should eat to achieve a specific macro set.

Mathematical/Algorithmic Overview and Analysis

We take the user's inputted food as a vector containing the food's macros (calories, protein, fat, carbs) – we take all of the inputted food vectors and join them as columns of a matrix A. Then we take the desired macro list as a vector and set that as our vector b. With this format, we can now see that finding the right proportions of each food is equivalent to solving for x in Ax=b – that is, what number should I multiple each column (food item) in matrix A so that each row (representing a single macro from each food) adds up to the corresponding entry in b (the desired macro)? Thus, step 1 is solving a system of linear equations.
**Note: explaining every aspect of the mathematical/algorithmic process can make this paper *very* lengthy. For detailed explanations on algorithmic processes and run time, please see the comments in the code. The code has been thoroughly commented. The following will be general overviews of the algorithms.

The input: a system Ax=b
The desired output: a strictly non-negative solution vector x that satisfies Ax=b

Here, we go on the journey to solving for this vector x.

1.1 Solving a System of Linear Equations

The steps to solving a system of linear equations (Ax=b) are:

1. Convert matrix A into Reduced Row Echelon form (RREF). RREF is an upper triangular matrix form from which the solution vector can be easily solved for via backsubstitution. Conversion to RREF takes $O(n^3)$ time, as every row and every column must be traversed. Even at best, for any general matrix, this time can only be divided in 2, which still results in $O(n^3)$ time. The time can be cut to linear time for a diagonal or k-diagonal matrix, but this requires a matrix of special form – this web app takes all general matrices.
2. Solve for the vector x by backsubstitution. Backsubstition, in my implemantation, takes n(n-1)/2 time by never unecessarily crossing over to the left side of the matrix diagonal, but even still, this results in $O(n^2)$ time (which is the general worst case time).
3. Solve for the nullspace vectors. Technically, the nullspace vectors should be solved mathematically, but I use a trick due to an observation that the nullspace vectors lie in the free columns of the matrix. Take the following example:

$$\begin{matrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 4 \end{matrix}$$

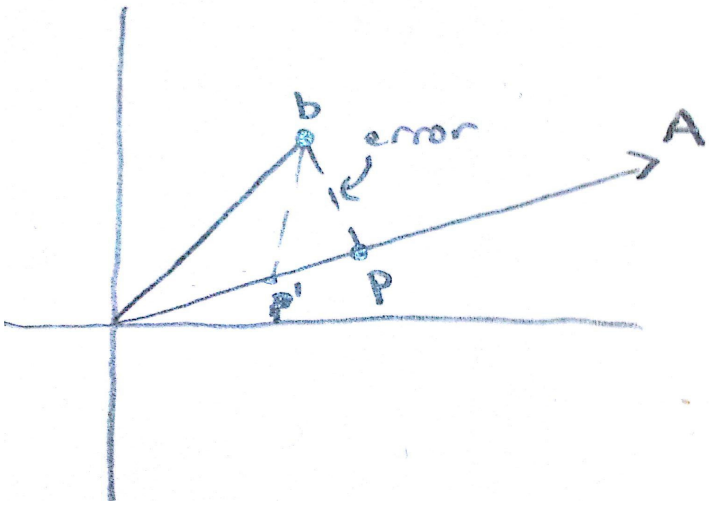
   In this example, the nullspace vector is $(-5, -2, -4, 1)^T$ which happens to be the negative of each component in the last column (the free column). Verify for yourself that this solution works (multiplying the above matrix by this vector produces a zero vector). Solving for the nullspace vectors with this trick runs in O(n) time (finally!)

In general, solving a system of linear equations takes $O(n^3)$ time unless you have a special matrix (i.e. diagonal, symmetric, etc). There are no known algorithms to alleviate this general run time for any general matrix.

If there turns out to be no solution to x, then we attempt to perform a Least Squares Approximation.

1.2 (optional) Solving Least Squares Approximation

Take the following 2 dimensional graph:



Notice that the b vector is not in line with the A vector. If I asked for some combinations or vectors in the line A to get b, I would not get a solution, because no combination of vectors in A could result in a vector in the direction of b. I can't solve this problem; but instead of giving up and running with my tail between my legs, I could try to find a solution vector *p* on the line A that minimizes the error between b and A. *p* is called the "projection of b onto A" and it minimizes the distance between the head of b and A (notice that any other length of *p*, such as *p'*, would increase the distance between the head of b and the line of A). This example is in 2 dimensions, but the idea holds for any dimension. The idea is, if you can't solve Ax=b, then project b onto A and try to solve for that. This is most used for fitting data points to parabolas and such things, but since the theory also applies in the case of this web app, I decided to use it here. Through an unbelievable amount of magical mathematics, solving this problem boils down to solving $A_{trans}x_{hat}=A_{trans}b$ as a system of linear equations where "trans" means "Transpose". Since this is basically solving a system of linear equations, the run time is $O(n^3)$, and cannot be mitigated, as explained earlier.

If solving this system of equation yields a non-negative x vector, then we have to do something else – it doesn't make physical sense to eat negative proportions of a food. If the vector x comes with some nullspace vectors, though, then we have something to work with. In a nutshell, nullspace vectors multiply with A to produce zero, so adding any combinations of nullspace vectors to x changes the value of x *while preserving the property Ax=b* – again, more beautiful linear algebra magic. The new equation that we want to solve is:

$$x + a_1N(A)_1 + ...+ a_nN(A)_n >= 0$$

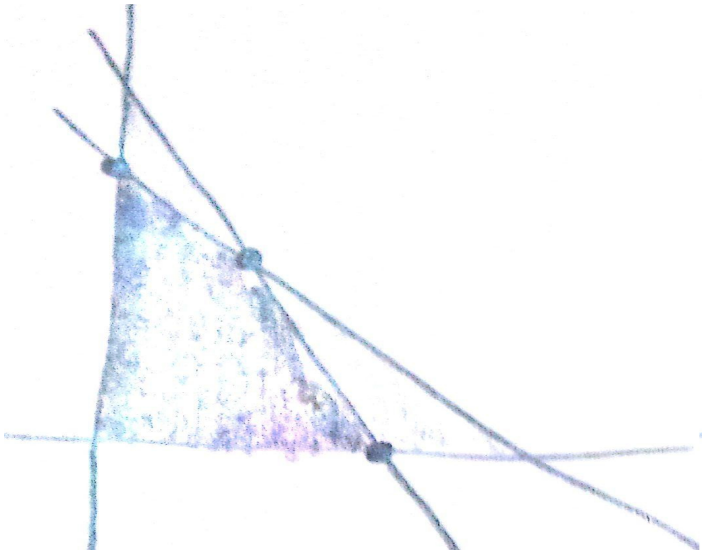
where x is a vector, $N(A)_i$ is a vector, and a_i is a scalar. We now set out to solve for these coefficients a_i.

1.3 The Simplex Method

We have some inequality:

$$x + a_1N(A)_1 + ...+ a_nN(A)_n >= 0$$

and we want to find the coefficients of the nullspace vectors that satisfy the constraint that the resulting vector is non-negative. This is called the "feasible region" problem – the solution is in a region of points within some constraining boundaries in a graph. Take the following 2 dimensional graph:

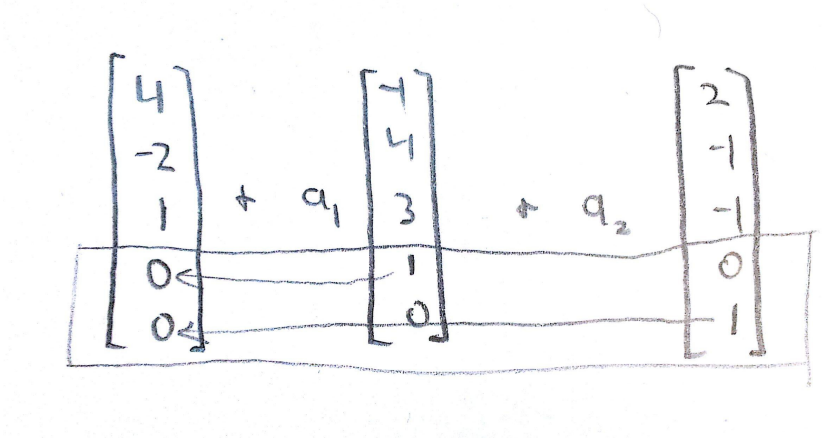


The darkly shaded region is a feasible region – every point in that region satisfies the given constraints, and every point outside of this region (including the lightly shaded regions) violates at least one constraint. The points that maximize the value of the objective function are one (or more) of the corner points of the of the feasible region, as shown.

For a 3 dimensional problem, the feasible region would be a polyhedron – maybe a cube or a sphere or something. After that, graphically finding these coefficients is impossible. The problem is, every nullspace vector adds another dimension to the problem, so an equation with *n* nullspace vectors has a dimension of *n*. The simplex method, in a nutshell, traverses the endpoints of the polytope, in any dimension, and correctly terminates when it has found the point on the graph that maximizes the objective function.

Since this is not a generic objective function maximization problem, I don't actually have an objective function to maximize. I figured out a neat way workaround. Each row of the inequality at the beginning of this subsection refers to a proportion of a specific food, and is also a constraining inequality (must be greater than zero), so I could use each row as a separate objective function, or combine the inequalities from specific rows to emulate “maximizing multiple specific food items”. With this little workaround, I can “sort of” emulate maximizing specific foods. I say “sort of” because if 2 rows (one that I want to maximize and one that I don't) have the same constraining equations, and the objective function is just a single equation, the simplex method has no way of telling apart the rows...all it knows is the objective function.

I also take advantage of the observation that no coefficient can be negative (this seems obvious, but it's not). See the diagram below:

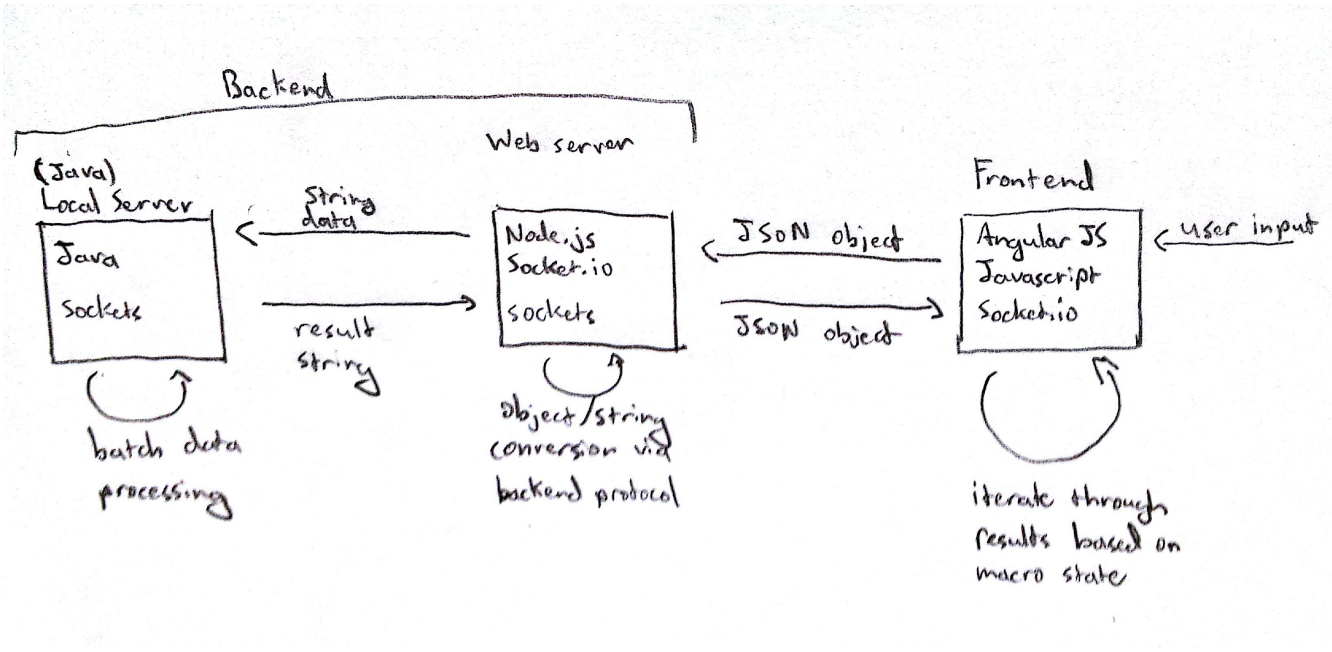


This diagram is representative of all systems that this web app will encounter. The x vector will have zeros for every nullspace vector, and each nullspace vector will have a 1 corresponding to a unique zero on the x vector, which means that if any coefficient is negative, then some component in the x vector is guaranteed to be negative. Because of this assumption, I'm able to skip some precautions and checking in building the Tableaus (which are the data structures used to perform simplex operations).

The simplex method is complicated so please see the code for the process – the code is documented. The worst case run time of the simplex method is exponential, and there are no known polynomial algorithms or heuristics that alleviate this run time. However, some German guy named Borgwardt demonstrated that the average run time is polynomially bounded.

Technical/architectural overview

Data flow diagram:



This web app has 3 main components: the front end, the web server, and the backend data processor.

1.1 The Frontend

The frontend is everything that the client sees and interacts with – everything inside the web browser. I use a Javascript-based framework called AngularJS, instead of regular old unflavored JavaScript, because AngularJS is a powerful framework that allows one to do things like dynamically insert html elements in response to user activity. To connect to the server, I use a module called Socket.io. Socket.io isn't natively supported by AngularJS, so I built a wrapper for it, which was neat. Socket.io is akin to Socket programming in Java or C++, except that it is event-driven. The main function of the frontend is to collect user data and send this user data to the server.

1.2 The Node Server

For the server technology, I used Node.js, which is, interestingly enough, written in JavaScript. I use Socket.io on the Node.js server to integrate communication between Node.js and the frontend AngulaJS. The reason I chose Node.js is because it's light (I only have around 20 lines of boilerplate code in my Node.js server file) and I knew that I would do most of the heavy computations in Java. The Node.js server primarily serves web pages to the user, converts data between String form and JSON form via Protocol, and routes information between the frontend AngularJS and the backend server/data processor, Java, which is next. Node.js has built-in socket functionality, so I took advantage of this to

communicate with the Java backend.

1.3 The Java Server

Technically, I have 2 servers, because the Java backend communicates locally with Node.js via Web Sockets. The Java side performs all of the heavy and substantial computations, and it is also here where the Protocol for communication between Java and Node.js is defined.

A Day in the Life: Program Execution, From Beginning to End

You're a user. You go to the Food Simplex app online, input some food data and your desired numerical macro values (if you don't care at all about certain macros, you can put any random non-negative number), and choose which foods you want to maximize. You click “Get Proportions”. AngularJS collects and sorts all of the user-inputted information into a single JSON object and sends that object to the Node.js server through a Socket.io interface.

At the Node.js server, Node.js receives the JSON object and writes the JSON object to a local Java program via a Web Socket. Node.js sends messages to Java through a Protocol (written by me) that itself and Java have mutually agreed to use. Node.js converts each field of the JSON object into a String and writes each line of the String to Java, interjected with Protocol messages so Java knows exactly what kind of message to expect and how to handle the incoming message.

At the Java program is where the bulk of the work happens. JavaServer receives the input from Node.js through Web Sockets. After it has received indication from Node.js that it has indeed received the entire message, via the agreed Protocol, JavaServer passes the information to NodeProtocol, which determines the message type from the message header, using the Protocol. In this case, this message will be a request to find a solution to the frontend problem of finding the exact food proportions to achieve a specific macro set. NodeProtocol extracts the data from the Node.js message as Matrices (Matrix objects). The Java program uses the NCR class to cycle through all combinations of possible macro fields (i.e. one solution involving only calories and protein, one involving only calories and fat, one involving calories, protein, and fat, etc.) and produces Matrix systems of Ax=b for each combination (since there are 4 macros, there will be 15 different systems to solve for, which excludes the possibility of considering no macros). The mathematical operations on these matrices will now begin with the MatOps class.

MatOps first attempts to find a solution by solving the system of linear equations Ax=b, where A is the food matrix and b is the desired-macros vector, to yield a vector x and the nullspace of A. If this fails to produce these things, then we perform a Least Squares Approximation to find a 'close' solution. If this fails still to produce these things, then there is no solution. Let's assume, however, that we did not fail, and we retrieved a solution vector x and a nullspace. If this solution turns out to be non-negative, then we've found a solution. But, of course, let's assume that our solution is *not* non-negative. We t hen pass this solution set to the TwoPhaseSimplex method to find some combination of nullspace vectors that, when added to our vector x, produce a non-negative vector that still correctly produces our desired macro set. If the TwoPhaseSimplex method fails to find a basic feasible solution (that is, a solution that correctly produces the desired macro set and adheres to the non-negativity constraints) then we have no solution. Let's assume that the TwoPhase Simplex found a solution. MatOps returns this solution to the JavaServer, and the JavaServer sends this solution to NodeProtocol, where NodeProtocol packages this solution in compliance with Protocol as a String to be sent to Node.js. After all 15 systems are solved in

this manner, NodeProtocol sends this package, now in String form, back to JavaServer, and JavaServer sends this String back to Node.js through a Web Socket.

When Node.js receives the message from Java, it decodes the message based on Protocol, and converts the message into a JSON object, based on, once again, Protocol. Node.js then sends the packaged JSON object back to AngularJS on the frontend through the Socket.io interface.

When AngularJS receives this JSON object, it displays the solution concerning all macros by default. Since all 15 possible solutions are cached on the frontend, you can immediately view the solution concerning any macro combination by checking off the relevant macro boxes and clicking “Update solution”. At this point, the you see a basic feasible solution, and proceed to pack your lunch before heading to the gym to become the next Arnold. You're welcome.

Personal Experience

At the beginning of the semester, I had no idea what to do for a project. In fact, I spent over a month juggling between project ideas. I originally started with an idea similar to Khalid's (the food/club website thing) but I didn't really know where to go with it. Then I moved to the “Entrepreneur's App”, which was a spinoff of the NP-Complete problem, independent set, and spanning tree stuff. I put the project ideas on hold because the semester was progressing quickly, and starting on the machinery of the web app.

Instead of using a full server suite like Rails of Django, I wanted to try something new and interesting with regards to web programming, so I decided to use Java to do my backend computations and Node.js as my web server (I think it paid off – I feel like writing this in some web language would have been hell). I had no idea how to integrate the two, though. I originally thought of writing information from Node.js to a file, and then having Java read the file, then having Java write a solution back to the file, and have Node.js read that solution from the file – but I didn't know how to get the timing right. Then I decided to try interprocess communication through a named pipe, but that was actually harder and very system dependent. After stumbling around for a while, I found that I could use sockets for interprocess communication.

Resuming the app ideas, I had no clue what to do, but day, this idea literally just hit me in the head – this “Bodybuilding Food App”. I'm not sure how – it was so completely random, but I liked it.

The algorithm idea was going to be to solve a system of linear equations to find exact food proportions, and do some other problems such as the knapsack problem and the rod cutting problem (they can be seen in my “Optimizer.java” class, which I abandoned) to make a multi-feature app. The problem was though, I later realized that solving the system of equations could lead to a solution containing negative food proportions, which doesn't make any sense. I had *no* idea how to figure the problem out. With the linear algebra that I knew how to do, I managed to figure out how to solve the problem graphically for 1, 2, and even 3 dimensions, but I had no idea how to handle more, let alone code the solution. After extensive internet research, I discovered the field of Linear Programming, and further, I discovered that the problem I was trying to solve was actually called the “feasible region” problem. I learned it by watching lectures (courtesy of IIT Madras, India) and studying online. After learning the theory, implementing it was another beast altogether.

As for the web specific stuff, instead of using JSON libraries to handle data manipulation, I decided to write my own protocols and handle the objects myself, just to see what it was like. At the very least, I

learned that proper protocol is *super* important, and bad intercommunication protocol can really destroy the efficacy of your app as a whole.

In general, while frustrating and seemingly hopeless at times, writing this app was a very rewarding experience. I'm proud of this app, and I am glad that this class gave me a chance to create something so cool.