Individual Student Assignment

One quarter of a student’s course grade is based on personal performance. The basic chores are:

• Write code for a well-known, simple problem. Typically, the code at most a few dozen lines.
• Test the code on a range of input sizes, collecting and plotting the running times.
• Use the test data to find a function that well approximates the data.
• Analyze the code’s syntax to develop a mathematical model of the algorithm’s time complexity: A big-$O$ analysis.

This term’s personal software development and analysis project is on Gaussian elimination.

Gaussian Elimination

Gaussian elimination is a method for computing the solution vector $\vec{x}$ of a given system of linear equations. 

$$A\vec{x} = \vec{b}$$

First, a reduction step transforms the augmented matrix $[A|\vec{b}]$ into upper-triangular form. Then, a second solving step produces the solution $\vec{x}$ by back-substitution.

Let’s get on with the code. Some types and helper functions may need to be defined. The reduction and solver steps need to be constructed. And, a main routine is needed as an entry point.

Here is the structure of the code. It is written in noweb style.

1a \( \langle 1a \rangle \equiv \)
\( \langle \text{Row, Column, and Matrix Types} \rangle \)
\( \langle \text{IO conversion functions} \rangle \)
\( \langle \text{The Gaussian reduction step} \rangle \)
\( \langle \text{The Gaussian solver step} \rangle \)
\( \langle \text{Main module} \rangle \)

Define the Row and Column types to be lists of double precision floating point numbers. Define the RMatrix type to be a list indexed by rows. Define the CMatrix type to be a list indexed by columns.

1b \( \langle 1b \rangle \equiv \)
\( \text{type Row} = [\text{Double}] \)
\( \text{type Column} = [\text{Double}] \)
\( \text{type RMatrix} = [\text{Row}] \)
\( \text{type CMatrix} = [\text{Column}] \)

\* Thanks to Lucky’s Notes for the structure of the code.
The Reduction Step

The reduction process used in Gaussian elimination transforms a matrix into upper-triangular form. Pictorially, for a small example.

\[
\begin{bmatrix}
P & \cdots & | \\
\cdots & \cdots & | \\
\cdots & \cdots & |
\end{bmatrix}
\mapsto
\begin{bmatrix}
1 & \cdots & | \\
0 & P & \cdots & | \\
0 & \cdots & |
\end{bmatrix}
\mapsto
\begin{bmatrix}
1 & \cdots & | \\
0 & 1 & \cdots & | \\
0 & 0 & P & \cdots & | \\
0 & 0 & \cdots & |
\end{bmatrix}
\mapsto
\begin{bmatrix}
1 & \cdots & | \\
0 & 1 & \cdots & | \\
0 & 0 & 1 & \cdots & | \\
0 & 0 & 0 & 1 & |
\end{bmatrix}
\]

The reduction steps for the first column are:

- Normalize the first row by scaling it by \(1/p\), where \(p\) is the value in the first row, first column. Assume \(p\), the pivot is not 0.
- Repeatedly, multiply the normalized first row by \(a\), \(b\), \(c\) and subtract the result row from second, third, and fourth rows.

To reduce a matrix to upper-triangular form repeat these reduction steps iteratively across the remaining rows and columns of matrix.

\[\text{gaussianReduce} :: \text{RMatrix} \to \text{RMatrix}\]

\[
gaussianReduce \text{ matrix} = \text{foldl} \ text{reducerow} \ text{matrix} \ [0..\text{length matrix}-2] \text{ where}
\]

\[
\text{reducerow} :: \text{RMatrix} \to \text{Int} \to \text{RMatrix}
\]

\[
\text{reducerow} \text{ m} \ r = \text{let}
\]

\[
(\text{Pick the row to reduce by, its pivot, and normalize this pivot row 2b})
\]

\[
(\text{Construct a function that reduces other rows 2c})
\]

\[
(\text{Apply the reduction function to rows below the pivot 3a})
\]

\[
(\text{Piece the matrix back together 3b})
\]

- Pick out the \(r\)-th row of matrix \(m\), using \(!!\), Haskell’s indexing operator.
- Pick out the \(r\)-th value in the \(r\)-th row and call it \(p\), the pivot. To keep things simple, assume these pivot elements along the main diagonal never equal 0.
- Finally, normalize the row by dividing each element in it by the pivot \(p\). The Haskell idiom that does this is to map the function \(x \mapsto x/p\) across the row. Call the normalized row \(row'\).

\[\text{Pick the row to reduce by, its pivot, and normalize this pivot row 2b} \equiv\]

\[
\text{row} = \text{m} \ (!! r)
\]

\[
\text{p} = \text{row} \ (!! r)
\]

\[
\text{row'} = \text{map} \ (\lambda x \to x/p) \text{ row}
\]

Now, to reduce another row, say \(nrow\), by \(row'\) apply the function

\[
nr \ast a - b
\]

(where \(nr\) is the \(r\)-th element in \(nrow\))

to each entry \(a\) in \(row\) and \(b\) in \(nrow\). The Haskell idiom for this is to \(\text{zip}\) the function with the values in \(row'\) and \(nrow\).

\[\text{Construct a function that reduces other rows 2c} \equiv\]

\[
\text{reduceonerow} \text{ nrow} = \text{let} \ nr = \text{nrow} \ (!! r) \text{ in} \ \text{zipWith} \ (\lambda a \ b \to nr\ast a - b) \text{ row'} \text{ row
}
Now, we can map the `reduceRow` function across all rows in matrix below the pivot row.

\[
\begin{align*}
& (\text{Apply the reduction function to rows below the pivot 3a}) \equiv \\
& \text{nextrows} = \text{map reduceonerow} \ (\text{drop} \ (r+1) \ m)
\end{align*}
\]

And finally, piece the results back together. Concatenate the first \( r \) rows from \( m \), row \( \text{row}' \) and the reduces \( \text{nextrows} \).

\[
\begin{align*}
& (\text{Piece the matrix back together 3b}) \equiv \\
& \text{in take} \ r \ m \ ++ \ [\text{row}'] \ ++ \ \text{nextrows}
\end{align*}
\]

To turn `gaussianReduce` into a standalone program, a `main` module, an `entry` point, must be established. IO can be tricky. The structure of the input must be known, and that must be translated into the structure of the Haskell function that implements the program.

Let’s agree, for the purpose of these notes, that the program is executed with input redirected from `standard input`, like so

```
./Gauss < data.txt
```

Assume the contents of `data.txt` is a string of lines separated by newline characters like so:

\[
\begin{array}{ccccccc}
& a_0 & a_1 & a_2 & \ldots & a_{0(n-1)} & b_0 \\
& a_{10} & a_{11} & a_{12} & \ldots & a_{1(n-1)} & b_1 \\
& \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
& a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \ldots & a_{n-1,n-1} & b_{n-1}
\end{array}
\]

Let’s agree to use the Haskell `idiom` `raw <- getContents` to input all of `data.txt` into one `String` called `raw`.

- The `lines` function breaks `raw` up into a list of strings `[String]`, separated at the newline. Call the result `rows`.
- Mapping the `words` function over `rows` breaks each string in `rows` list of `Strings`. These values need to be converted from `String` to `Double`.

\[
\begin{align*}
& (\text{Main module 3c}) \equiv \\
& \text{main :: IO ()} \\
& \text{main = do} \\
& \quad \text{raw <- getContents} \\
& \quad \text{let rows = lines raw} \\
& \quad \text{let rows' = map words rows} \\
& \quad \text{let matrix = map stringsToDoubles rows'} \\
& \quad \text{print $ matrix ++ gaussianReduce matrix}
\end{align*}
\]

\[
\begin{align*}
& (\text{IO conversion functions 3d}) \equiv \\
& \text{stringToDouble = read :: String -> Double} \\
& \text{stringsToDoubles :: [String] -> [Double]} \\
& \text{stringsToDoubles xs = map stringToDouble xs}
\end{align*}
\]
Test the Reduction Step

The noweb source file Gauss.nw is here.

- Running notangle Gauss.nw > Gauss.hs generates the Haskell code.
- Running noweave -index -delay Gauss.nw > Gauss.tex generates a \LaTeX\ file that is included in a wrapper \LaTeX\ file, that produces this document.

You may not have the noweb tools. You can retrieve the Haskell code from a link in the first step of this part of the assignment.

1. Download these files at the links: Gauss.hs and Gauss.tex.
2. Install the Glasgow Haskell Compiler and use its interpreter ghci to load Gauss.hs. Check the reduction code is correct on some simple cases, for instance,
   - *Main> gaussianReduce [[]] – the empty matrix
   - *Main> gaussianReduce [[1]] – a \(1 \times 1\) matrix that needs no normalization
   - *Main> gaussianReduce [[2]] – a \(1 \times 1\) matrix that needs normalization
   - *Main> gaussianReduce [[1,2]] – the equation \(2x = 1\)
   - *Main> gaussianReduce [[1,2],[2,3]] – an inconsistent system
   - *Main> gaussianReduce [[1,-2,1,4],[2,3,-1,5],[3,1,4,7]] – a picked out of the air example

3. Write code to generate \(n\) lists of random Doubles of length \(n + 1\) that represents a linear system \(Ax = b\), where \(A\) is an \(n \times n\) matrix and \(b\) is an \(n \times 1\) vector.

4. Run your data generation code to generate several data files of varying sizes \(n \times n + 1\).

5. Compile the Haskell source Gauss.hs with ghc profiling options, see the ghc User’s Guide on Profiling for instructions on this.

6. Execute Gauss on your data files, and collect running times.

7. Plot the running times. Find a curve that approximates the data. The \(R^2\) value for the approximation should be close to 1.

Analyze the reduction step

A big-\(O\) time complexity can be computed for each function in the code. For instance, the anonymous function \((\text{\textbackslash a \ b -> n\textbackslash r*a ~ b})\) has time complexity \(O(1)\). The time cost to map it over a list of length \(n + 1\) is \(O(n)\).

What are the big-\(O\) time complexities for the functions below. Explain your reasoning for each function. In particular, identify the size and type for the input and output of each function.

- stringsToDoubles
- map stringsToDoubles
The Gaussian Solver

The solving step in Gaussian elimination uses back substitution to solve for the values in \( \vec{x} \) in order \( x_{n-1}, x_{n-2}, \ldots, x_{0} \). Write Haskell code that implements the solver step.

Assignment, Part 3: Complete the Gaussian elimination algorithm by implementing the solver step. (25 points)

\[
\begin{align*}
&\text{(The Gaussian solver step 5)} \equiv \\
&- \text{gaussianSolve :: RMatrix} \rightarrow \text{[Double]} \\
&- \text{Your code goes here}
\end{align*}
\]

Test the Solver Step

Test your gaussianSolve code following steps similar to those outlined in testing gaussianReduce.

Analyze the solver step

What is the big-O time complexity of your gaussianSolve code? Explain your reasoning. In particular, identify the size and type for the input and output of gaussianSolve and internal functions it calls.

Assignment, Part 4: Perform a mathematical analysis of the solver algorithm. (25 points)

References

