Assignments for Students as Individuals

CSE 5211 Analysis of Algorithms, Instructor: William Shoaff

Spring 2018

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. [1] Some types and helper functions will be needed. The reduction and solver steps must 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.

[1] Thanks to Lucky's Notes for the structure of the code.

1a ⟨*Gauss* 1a⟩≡

```
⟨Row, Column, and Matrix Types 1b⟩
⟨IO conversion functions 3d⟩
⟨The Gaussian reduction step 2a⟩
⟨The Gaussian solver step 5⟩
⟨Main module 3c⟩
```

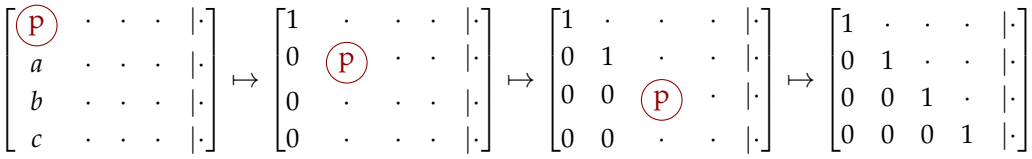
Define the `Row` and `Column` types to be lists of double precision floating point numbers. Define the `RMatrix` type to be a list indexed by rows. Define the `CMatrix` type to be a list indexed by columns.

1b ⟨*Row, Column, and Matrix Types* 1b⟩≡

```
type Row = [Double]
type Column = [Double]
type RMatrix = [Row]
type CMatrix = [Column]
```

The Reduction Step

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



To reduce a matrix to upper-triangular form repeat these steps iteratively across all rows of the `matrix`. For the first row:

1. Assume p, the `pivot`, is not 0 and normalize the row by scaling it by $1/p$.
2. Repeatedly, multiply the normalized row by a, b, c and subtract the result row from second, third, and fourth rows, respectively.

2a ⟨*The Gaussian reduction step* 2a⟩≡

```
gaussianReduce :: RMatrix -> RMatrix
gaussianReduce matrix = foldl reducerow matrix [0..length matrix-2] where
  reducerow :: RMatrix -> Int -> RMatrix
  reducerow m r = let
    ⟨Pick the row to reduce by, its pivot, and normalize this pivot row 2b⟩
    ⟨Construct a function that reduces other rows 2c⟩
    ⟨Apply the reduction function to rows below the pivot 3a⟩
    ⟨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.
- Normalize the row by dividing each element in it by the pivot p. The Haskell *idiom* that does this is to map the anonymous function $x \mapsto x/p$ across the row. Call the normalized row `row'`.

2b ⟨*Pick the* `row` *to reduce by, its* `pivot`*, and normalize this pivot* `row` 2b⟩≡

```
row = m !! r
p = row !! r
row' = map (\x -> x/p) row
```

Now, to reduce another row, say `nrow`, by `row'` apply the function

$$nr * a - b \qquad \text{(where nr is the r-th element in nrow)}$$

to each entry a in `row` and b in `nrow`. The Haskell *idiom* for this is to *zip* the function with the values in `row'` and `nrow`.

2c ⟨*Construct a function that reduces other rows* 2c⟩≡

```
reduceonerow nrow = let nr = nrow !! r in zipWith (\a b -> nr*a - b) row' nrow
```

Next, map `reduceRow` across all rows in `matrix` below the `pivot` row.

3a ⟨*Apply the reduction function to rows below the* `pivot` 3a⟩≡
```
nextrows = map reduceonerow (drop (r+1) m)
```

And finally, piece the results back together: Concatenate the first `r` rows from `m`, row `row'` and the reduces `nextrows`.

3b ⟨*Piece the matrix back together* 3b⟩≡
```
in take r m ++ [row'] ++ nextrows
```

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 the 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 < gauss.dat
```

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

$$\begin{array}{cccccccc} a_{00} & a_{01} & a_{02} & \dots & a_{0(n-1)} & b_0 & \backslash n \\ a_{10} & a_{11} & a_{12} & \dots & a_{1(n-1)} & b_1 & \backslash n \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \dots & a_{n-1,n-1} & b_{n-1} & \backslash n \end{array}$$

Let's agree to use the Haskell *idiom* `raw <- getContents` to input all of `gauss.dat` 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`.

3c ⟨*Main module* 3c⟩≡
```
main :: IO ()
main = do
  raw <- getContents
  let rows = lines raw
  let rows' = map words rows
  let matrix = map stringsToDoubles rows'
  print $ matrix ++ gaussianReduce matrix
```

3d ⟨*IO conversion functions* 3d⟩≡
```
stringToDouble = read :: String -> Double

stringsToDoubles :: [String] -> [Double]
stringsToDoubles xs = map stringToDouble xs
```

Test the Reduction Step

Assignment, Part 1: Complete the steps in this section. (25 points)

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,
   - `*Gauss> gaussianReduce [[]]` – the empty matrix
   - `*Gauss> gaussianReduce [[1]]` – a 1×1 matrix that needs no normalization
   - `*Gauss> gaussianReduce [[2]]` – a 1×1 matrix that needs normalization
   - `*Gauss> gaussianReduce [[1,2]]` – the equation $2x = 1$
   - `*Gauss> gaussianReduce [[1,2],[2,3]]` – an inconsistent system
   - `*Gauss> 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

Assignment, Part 2: Perform a mathematical analysis of the reduction algorithm. (25 points)

A big-O time complexity can be computed for each function in the code. For instance, the *anonymous* function `(\a b -> nr*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`

- `reduceonerow`
- `reduceRow`
- `take r m ++ [row'] ++ nextrows`
- `gaussianReduce`

Does the empirical run time data from experiments in you performed when testing `gaussianReduce` agree with the analytic analysis?

The Gaussian Solver

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

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.

5 ⟨*The Gaussian solver step* 5⟩≡

```
- gaussianSolve :: RMatrix -> [Double]
- Your code goes here
```

Test the Solver Step

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

Analyze the solver step

Assignment, Part 4: Perform empirical and mathematical analyses of the solver algorithm. (25 points)

1. What is the big-O time complexity of your `gaussianSolve` code?
2. Write a function that tests the accuracy of the computed solution $\vec{x}'$.
   (a) Use the L_∞ norm to measure and report solution accuracy.

$$\|b' - b\|_\infty = \max\left\{\left|b'_0 - b_0\right|, \left|b'_1 - b_1\right|, \ldots, \left|b'_{n-1} - b_{n-1}\right|\right\}$$

   where $b' = A\vec{x}'$.

   (b) Hilbert matrices H_n are famous examples of *ill-conditioned* matrices. Informally, this means solving $H_n\vec{x} = b$ accurately is difficult. The Hilbert H_5 is

$$H_5 = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

   In general, the value in row i, column j is

$$(H_n)_{ij} = \frac{1}{i+j+1}, \quad i, j = 0, \ldots, n-1$$

   Test your code on Hilbert matrices.

References

Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/.

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition.

The GHC Team (2014). *The Glorious Glasgow Haskell Compilation System User's Guide*.