Calculus 3 for Computer Science Project

Will Farrington, Jeff Drasher, Mike Hirth

April 12, 2009

Contents

1 Introduction

Much of the code used within this assignment is Ruby version 1.9.x. Since it is, for most intents and purposes, a lesser known language than say, Java, we felt it necessary to explain bits of the language which might surface in the code we wrote for this project.

First and foremost, Ruby shares many similarites with, and is inspired by, Lisp, Smalltalk, and Perl (most likely in that order). It contains many of the niceties of functional programming that Lisp does, it is truly object oriented in the same way Smalltalk is, and it is a terse scripting language with powerful features like Perl is.

1.1 Syntax, Language Quirks, Etc.

Notations that might seem confusing unless one has a background in all three of these languages are such:

Basic Syntax Comments are preceded by a `#`. Strings are wrapped in either single- or double-quotes. Indentation is two spaces. Rather than using indentation-based block delimiters or curly braces, Ruby simply uses the end keyword.

Blocks Blocks are notations for anonymous functions in Ruby (like lambda in Lisp). They can be written a number of ways, such as:

```
# Assigning a lambda/proc to a variable, and then calling it
f = ->(x){ x + 1 }
f[2] #=> 3

# Array#map is one of several functions which takes a block as an argument
a = [1,2,3]
a.map { |i| i * i } #=> [1,4,9]

# This is the expanded form of the block, used for creating multi-line anonymous
    functions
a.each do |item|
  puts item
end
# would print 1, 2, and 3 on separate lines
```

Ranges Ranges are represented in Ruby with one of two operators, `..` or `...`. `0..10` is an inclusive range ($[0, 10]$) in a more mathematical notation, whereas `0...5` is exclusive ($[0, 5)$). Ranges can be iterated across.

Method Invocation In Ruby, method invocation has optional parens. Rather than using the form `instance.method(arg1, arg2)`, one can use the form `instance.method arg1, arg2`. In the case where an invocation doesn't have arguments, the parentheses are still optional.

Object Oriented Ruby, like Smalltalk, is object oriented down to the primitives of the language. This means that all things in Ruby are objects, and thus have methods that can operate on

them. This library was written to make use of this, monkey-patching functionality into the existing Matrix and Vector classes in Ruby.

Notation Ruby has a common nomenclature for expressing its classes and their methods. `Object#method` is the de facto standard among Rubyists, hence, that's the form we'll use here. Similarly, `#=>` is used to denote return values.

Further Notes Ruby's Matrix and Vector classes lack `#[]=` methods, therefore, we often convert these two datatypes to arrays and back again to perform matrix or vector arithmetic or other operations.

Hopefully, that should clear up any misconceptions or confusion before addressing the actual code at hand. That said, all three parts of this report do make use of both some standard libraries in Ruby, as well as extensions upon them.

- http://www.ruby-doc.org/core/classes/Array.html
- http://www.ruby-doc.org/core/classes/Matrix.html
- http://www.ruby-doc.org/core/classes/Vector.html

1.2 Common Code

Additionally, we wrote an abstraction layer into some of these classes via monkeypatching in order to add some common functionality:

Listing 1: Common Code for All Three Parts

```
class Vector
  private
  def sign(x)
    return 1 if x > 0
    return -1 if x < 0
    return 0
  end
end

class Matrix
  def pretty_print
    str = ""
    self.to_a.each do |row|
      row.each do |i|
        if i.to_i >= 0
          str << " "
        end
        if ("%.3f" % i).to_f == i.to_i
          str << "#{i.to_i}     "
        else
```

```
          str << "%.3f " % i
        end
      end
      str << "\n"
    end
    puts str
  end

  def inf_norm
    self.to_a.map do |a|
      a.map do |ar|
        ar.abs
      end.inject(&:+)
    end.sort[0]
  end

  def is_lower_triangular?
    triangular(self.column_vectors)
  end

  def is_upper_triangular?
    triangular(self.row_vectors)
  end

  private
  def triangular(vecs)
    for i in 0...vecs.length
      vec = vecs[i].to_a
      unless i <= 1
        return false unless vec[0...i].all? { |n| n == 0 } and vec[i..-1].all? { |n| n != 0
            }
      end
    end
    return true
  end
end
```

1.3 About This Document

This document was typeset in LaTeX. It uses the *color* and *listings* packages for the code formatting. The source code for this document is available at http://github.com/wfarr/calc3-for-cs/report/report.tex.

2 Part One

The purpose of Part One of the project is to solve the typical $A\vec{x} = \vec{b}$ equation, with A being a Hilbert matrix. A Hilbert matrix is a square matrix whose elements follow the form

$$H_{ij} = \frac{1}{i + j - 1}$$

Here's an implementation in Ruby:

Listing 2: Hilbert Matrix Implementation

```
class Matrix
  def self.hilbert(n)
    m = Matrix.zero(n).to_a
    m = m.each_index.map{|row| m[row].each_index.map{|col| 1 / (row + col + 1)}}
    return Matrix.rows(m)
  end
end

Matrix.hilbert(4) #=> Matrix[[1/1,1/2,1/3,1/4], [1/2,1/3,1/4,1/5], [1/3,1/4,1/5,1/6],
    [1/4,1/5,1/6,1/7]]
```

Often times, simplifying a single matrix A into two or more "nicer" matrices (in the case of these algorithms, LU or QR) can make solving the equation $A\vec{x} = \vec{b}$ easier. Such algorithms introduce the potential for error, namely because they are modified forms of the original matrix.

2.1 LU Decomposition

LU Decomposition uses matrix multiplication to reduce a matrix A into two matrices, L (a lower triangular matrix) and U (an upper triangular matrix).

2.1.1 Explanation of the Algorithm

The algorithm for doing so is fairly simple in and of itself:

1. Starting with the first column, find the first non-zero entry below the diagonal. Let this entry be considered x. Let that column's diagonal element be y.
2. Multiply an Identity matrix, with the location of the entry x set to the value $-\frac{x}{y}$. This matrix is L_n.
3. The resulting matrix is the new A for the next iteration.
4. Repeat these steps until the resulting A is upper triangular. At this point, A becomes U.
5. To find L, multiply $L_1^{-1}L_2^{-1}...L_n^{-1}$.
6. Substitute A with LU in the equation $A\vec{x} = \vec{b}$ and solve.

2.1.2 Implementation of the Algorithm

Listing 3: LU Decomposition

```
class Matrix
  def lu_decomposition
    return nil unless self.square?
    n = self.row_size
    a = self
    l_n = []
    cvs = a.column_vectors.map { |v| v.to_a }
    for k in 0...cvs.length
      for j in 0...cvs.length
        l_new = Matrix.identity(n).to_a
        if l_new[j][k] == 1 || j < k
          next
        end
        l_new[j][k] = - (cvs[k][j] / cvs[k][k])
        l_n << l_new
        a = Matrix[*l_new] * Matrix[*cvs.transpose]
        cvs = a.column_vectors.map { |v| v.to_a }
      end
    end
    l_final = l_n.map { |m| Matrix[*m].inverse }.inject(&:*)
    u_final = a
    return l_final, u_final
  end
end
```

The algorithm first begins with an essential check: the method `self.square?` determines if the matrix is a square matrix, and returns true if it is. LU Decomposition can only be done on square matrices, thus, the method returns nil when given a non-square matrix. Next, the algorithm defines A (written as `a` in the code because `A` would've been a Constant rather than a variable) to be the instance of `self`. To iterate across the columns efficiently, we use `Matrix#column_vectors`, which returns an array of column vectors. This array is then `map`ped over to convert the vectors into arrays. The end result is that `cvs` is an array of arrays representing the columns of `self`.

The actual computation lies in the nested `for` loops. For each iteration, an `l_new` matrix is created and converted to an array. If the current values of `j` and `k` are above the diagonal, then the algorithm skips to the next iteration. Next, `l_new[j][k]` is set to $-\frac{x}{y}$, as above in the algorithm's description. A new `a` is made as the product of `l_new` and `cvs.transpose` (the same matrix as A). The last step of each iteration is rebuilding `cvs` based off of the newest `a`.

Finally, L and U are assigned and returned. While `u_final` is straight-forward, `l_final` is a bit more complicated. `l_n.map { |m| Matrix[*m].inverse }` returns an array of inverted matrices from the original array of arrays (of arrays). The one bit of syntactic sugar in that line is the use of `Matrix[*m]`. In this case, `*` is acting as the glob operator, essentially inserting all the content of the array it's called on rather than simply inserting the array itself.

This is necessary because `Matrix[...]` takes a list of rows (in the form of arrays) as its argument. Finally, this new array is passed `Array#inject`, which applies a given block to all elements

of an array and returns the result. In this case, the injection is making use of a feature in Ruby 1.9 called `symbol_to_proc`, which allows for passing the method the `:*` symbol and automatically converting it into a proc/lambda. Thus, the result of the injection is to multiply all the results of the map together, in order.

2.1.3 Results and Analysis

The error introduced by the algorithm is effectively 0.

2.2 Householder Reflections

Lots of stuff about Householder Reflections.

2.2.1 Description of the Algorithm

Snafu.

2.2.2 Implementation of the Algorithm

Listing 4: QR Decomposition via Householder Reflections

```
class Matrix
  def householder
    return nil unless self.square?
    current_iteration = self
    init_dim = self.row_size
    h_list = []
    cv = current_iteration.column_vectors[0]
    h = (cv.find_householder_reflection - Matrix.identity(cv.size)).expand_to_dimensions(
        init_dim,init_dim) + Matrix.identity(init_dim)
    h_list << h
    current_iteration = h * current_iteration
    for i in 0...self.row_size
      cv = current_iteration.get_column_vector(i+1)
      break if cv.size < 2 || current_iteration.is_upper_triangular?
      h = (cv.find_householder_reflection - Matrix.identity(cv.size)).expand_to_dimensions(
          init_dim,init_dim) + Matrix.identity(init_dim)
      h_list << h
      current_iteration = h * current_iteration
    end
    q,r = h_list.inject(&:*), current_iteration
    return q,r
  end

  def expand_to_dimensions(x,y)
    curr_x, curr_y, a = self.row_size, self.column_size, self.to_a
    a.each_index do |row|
      for i in 0...(y - curr_y)
        a[row] = a[row].insert(0,0)
      end
    end
    for i in 0...(x - curr_x)
```

```
      a = a.insert(0,Array.new(y){0})
    end
    return Matrix.rows(a)
  end

  def get_column_vector(x)
    return Vector.elements(self.column(x)[x..-1])
  end
end

class Vector
  def find_householder_reflection
    a = self.to_a
    a = a[0] if a[0].is_a?(Array)
    a[0] = a[0] + sign(a[0]) * self.r
    u = Vector[*a]
    norm_u_sqrd = u.r**2
    uut = u.covector.transpose * u.covector
    h = Matrix.identity(uut.row_size) - (uut * (2 / norm_u_sqrd))
    return h
  end
end
```

2.2.3 Results and Analysis

2.3 Givens Rotations

Lots of stuff about Givens Rotations.

2.3.1 Explanation of the Algorithm

Snafu

2.3.2 Implementation of the Algorithm

Listing 5: QR Decomposition via Givens Rotations

```
class Matrix
  def givens
    return nil unless self.square?
    n = self.row_size
    a = self
    g_n = []
    cvs = a.column_vectors.map { |v| v.to_a }
    for i in 0...cvs.length
      for j in 0...cvs.length
        next unless j > i
        g = Matrix.identity(n).to_a
        c = cvs[i][i] / Math.sqrt(cvs[i][i]**2 + cvs[i][j]**2)
        s = -cvs[i][j] / Math.sqrt(cvs[i][i]**2 + cvs[i][j]**2)
        g[i][i], g[j][j] = c, c
        g[j][i], g[i][j] = s, -s
        g = Matrix[*g]
```

```
        g_n << g
        a = g * a
        cvs = a.column_vectors.map { |v| v.to_a }
      end
    end
    q,r = g_n.map { |m| m.t }.inject(&:*), a
    return q,r
  end
end
```

2.3.3 Results and Analysis

3 Part Two

Clusterfuck.

3.1 Jacobi Method

Lots of stuff about Jacobi Method.

3.2 Gauss-Seidel Method

Lots of stuff about Gauss-Seidel Method.

4 Part Three

Less of a clusterfuck.

4.1 Leslie Matrices

Also stuff.

4.2 Power Method

Stuff galore.

4.3 Answering the Question of Population Trends

4.3.1 Question 1

The problem asked to interpret the data in the matrix and discuss the social factors that influence those numbers. The answer to this question was derived from an analysis of the given Leslie matrix.

In the given Leslie matrix the survival rate in the different age groups and the fecundity both vary. The fecundity can be seen when analyzing the first row of the matrix. In the age groups 0-9 and 50 and over, the average birth of females from that age class is zero because women are not physically fertile. In the ranges 10-19 and 20-29 the birth rate is low because women generally have the most children in their thirties, which is apparent in the .9 per capita average in that age group.

Looking at the data in each of the columns of the matrix, the fraction of surviving individuals varies due to several factors. The number in each column represents the surviving fraction that makes it to the next group. Newborns of age 0-9 have a fraction of .7 possibly due to infant deaths which occur due to birth complications and medical problems. The survival rates increase in the groups of 10-19 and 20-29, particularly because individuals are the healthiest during this "prime" of their life. The rate slowly decreases after 30-39 and over the next few age groups until it drops .4 in the 70-79 age group in which health complications due to old age cause more deaths.

4.3.2 Question 2

The problem asked to calculate the population distribution, total population, and fraction of change of the total population in 2010, 2020, 2030, 2040, and 2050.

The approach to this problem was to use the Leslie Matrix Model and multiply the Leslie Matrix by each previous population distribution, starting in 2010 and using the given distribution for 2000.

The population distribution is shown in the code ¡pt3.rb:74-79¿. The total population of the city in question for each year was as follows:

2010: 173490.0
2020: 182815.0
2030: 221212.4
2040: 265050.467
2050: 338794.2932

With the fractional change of the total population:
2000 → 2010: 22.18%
2010 → 2020: 5.37%
2020 → 2030: 21.00%
2030 → 2040: 19.82%
2040 → 2050: 27.82%

By implementing the formula given in the problem, $x(k+1) = A\vec{x}(k), \quad k = 0, 1, 2, 3, 4, 5$ and $\vec{x}(0)$ the given population distribution, the population distribution could be calculated for each decade increase. This, more simply put, is a multiplication of the Leslie matrix with the previous population distribution, implemented here ¡pt3.rb:74-79¿, where $\vec{x}_0$ is the $\vec{x}(0)$ population distribution. In order to find the total population of one year, a simple summation of the elements of each of the population distribution matrices (multiplied by 10^5) shown in ¡pt3.rb:83-90¿ gives the total population. The percent increase of the population was calculated by taking the difference in total population divided by the previous population, for each decade as shown: ¡pt3.rb:105-109¿.

4.3.3 Question 3

The problem asked to write a program implementing the Power method to calculate the largest eigenvalue of a $n \times n$ matrix where the method stops after 8 digits of accuracy, and to use the program to calculate the largest eigenvalue of the Leslie matrix.

The problem was approached by developing a Power method algorithm in Ruby that took in an $n \times n$ matrix that iterated until 8 digit decimal precision was achieved. After testing to see if the program returned correct eigenvalues for any square matrix, it was passed the Leslie matrix in order to find the largest eigenvalue.

After running the program using the Leslie matrix ¡pt3.rb:69¿, the method returned the eigenvalue 1.28865626235758.

Implementing the power method in Ruby only required the creation of one method because of some shared coding with part 1 including the `Vector#inner_product` method to calculate the dot product, used in part of the algorithm. The general formula for the power method can be found in the web notes by Laszlo Erdos for *Linear Algebra: Numberical Methods in Numerical Computation of Eigenvalues*(p 62).

where $\vec{w}$ and $\vec{u}_0$ are randomly chosen nonzero vectors. As shown in ¡pt3.rb:9-14¿ the power method program implemented uses a $\vec{w}$ vector of $[1, 0, 0..., 0_n]^t$ and a $\vec{u}_0$ vector of $[1, 1, 1..., 1_n]^t$ both with the number of entries equal to n of the $n \times n$ Leslie matrix. These are the vectors we

used in class to compute eigenvalues with the Power Method but they could be any vectors as long as they are not zero. The coding of the algorithm for the actual iterations of the method can be seen in the while loop in ¡pt3.rb:35-49¿. There are two general parts to calculating the eigenvalue. First, $\vec{u}_1$ is calculated by ¡pt3.rb:14¿ and later $\vec{u}_{(n+1)}$ with the same code but over many iterations in the while loop, represented in the code, after certain conversions, as $\vec{u}_{new}$. This is modeled by $\vec{u}_{(n+1)} = A\vec{u}_{(n)}$ on page 62 of the web notes by Erdos, where in the code $\vec{u}_{new}$ is $\vec{u}_{(n+1)}$, $\vec{u}_{prev}$ is $\vec{u}_n$ and A is a, the given matrix.

The second part involves the eigenvalue calculated using the inner products according to the limit exampled earlier. The code is first implemented here: ¡pt3.rb:22,23¿ and later on over multiple iterations, where `lambda` is the approximated eigenvalue and `lambda_prev` is saved for precision calculation. The break statement of ¡pt3.rb:48¿ terminates the loop after the difference in lambdas is less than 10^-8, guaranteeing that precision. The first iteration does not have a lambda to compare so it occurs outside and before the loop, creating some repetition of code. When converting matrices to arrays in order to multiply the inner product a small problem is presented by the conversion in Ruby. When using the function `Matrix#.to_a` the given matrix is converted to an array of arrays containing numbers instead of just an array of numbers. To take care of this, a new array is created and filled with the elements of the old array shown here ¡pt3.rb:44-46¿. In the code the first calculation of $\vec{u}_{(n+1)}$ is `u_new_matrix` and in Ruby is of the type `Matrix`. The array conversion is denoted as `u_new_array` and finally the array of numbers which can actually be used is $\vec{u}_{new}$. Once lambda is calculated as a float point of precision to 8 decimals it is returned by `Matrix#power_method`.

The next part of the problem asked to analyze the eigenvalue obtained for the Leslie matrix and discuss what this means about the growth of the population in the long run.

Obtaining the eigenvalue of 1.28865626 reveals the eventual behavior of the population of the given city. The eigenvalue is greater than 1 which means the population will grow. If it was less than 1, the population would over time become extinct and if it was equal to 1 then the population would remain stable. This can be shown using the theorem covered in chapter 4 of Erdos's notes (p 49):

> Theoreom 4.1
>
> For a square matrix A, $||A^k||$ eventually goes to 0 if and only if, all eigenvalues of A satisfy $|\lambda| < 1$
>
> When looking at the iterative model given in the question
>
> $$\vec{x}(k) = A^k\vec{x}(0)$$
>
> it is possible to see that a norm that converged to zero with a large enough k would eventually give a population of zero. This would happen if the largest eigenvalue was less than 1, and it can be drawn that because our eigenvalue computed for the Leslie matrix is greater than 1 the population must grow at that rate.

Therefore we arrive at the equation

$$\vec{x}(n+1) = \lambda\vec{x}(n)$$

for large values of n, where λ is the dominant or largest eigenvalue.

This shows that as n gets larger $\vec{x}(n+1)$ is a scalar multiple of $\vec{x}(n)$ and if that scalar, the computed eigenvalue, is positive the population will grow by that growth rate, as shown in our given case of the population of a city.

4.3.4 Question 4

The problem asked to decrease the birth rate of the second age group by half in 2020 and then predict the population for 2030, 2040 and 2050, discussing again the meaning of the eigenvalue for this new matrix over a long period of time.

This was approached by changing the original Leslie matrix to reflect the new change in birth rate and then applying the previous method of calculating using the total populations and fractional change for the years of 2030, 2040, and 2050.

The newly computed eigenvalue is 1.16790279. The new populations of the following years are:

2020: 182815.0
2030: 194542.4
2040: 224593.967
2050: 262190.9132

and the percentage of change in populations are

2020 → 2030: 6.41%
2030 → 2040: 15.45%
2040 → 2050: 16.74%

These values were arrived at by first altering the Leslie matrix as seen here ¡pt3.rb:113¿ where the birth rate of the second age group, which was previous at 1.2 was decreased to 0.6. This birth rate changes in 2020 so the population distribution in 2010 and 2020 remains the same as presented in the data. In calculating the distribution in 2030, the calculation used before still applies but includes the new Leslie matrix. The population distribution in 2030 is equal to the new Leslie matrix times the population distribution in 2020. In Ruby this looks like ¡pt3.rb:123-125¿, where `x_3new, x_4new, x_5new` are all of the new distribution matrices. Once these are calculated the total populations can be found by using the same method earlier, which was to add each of the numbers in the population distribution ¡pt3.rb:132-137¿. Finding the percent change was also the same as previously done.

The eigenvalue is computed by using the Power Method on the new Leslie matrix, using a simple call to the method ¡pt3.rb:147¿

This eigenvalue is still a positive number greater than 1, showing a population that will still grow positively and eventually level off at a factor of 1.167902791647756 for the new birth rate, which is somewhat lower than the previous rate. The population totals show the growing population diminishes due to the halved birthrate of the second age group in 2020 and by 2050 where the population would have been 338794.2932 it is now 262190.9132. However, the population will not diminish because it's dominant eigenvalue is still greater than 1.