Calculus 3 for Computer Science Project

Will Farrington, Jeff Drasher, Mike Hirth

April 11, 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.

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:

```ruby
# 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

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
```

```ruby
  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
```

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 matrices (in the case of these algorithms, LU or QR) and then solve. Such algorithms introduce the potential for error, namely because they are modified forms of the original matrix.

2.1 LU Decomposition

Lots of stuff about LU Decomp.

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
```

2.2 Householder Reflections

Lots of stuff about Householder Reflections.

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
```

2.3 Givens Rotations

Lots of stuff about Givens Rotations.

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
```

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.