assignment1

July 22, 2021

1 General Notes

By Ryan Cox

I find version control useful for me so I'm keeping my work in a git repo. You can find it here: https://github.com/Infinite-Improbability/math322-inverse-theory

My experience, as discussed, is in Python. However, I decided that if I'm going to be doing a lot of matrix work I'd use something with native support and better performance. MatLab is, of course, great with matrices but I prefer something a bit more open. Enter Julia, which is both fast and designed with linear algebra and scientific computing in mind. I've been interested in learning it, so here we go.

2 Q1 | Problem 1.1

We are measuring subsets of objects. The data is the measured masses of the subsets (mostly triplets of objects). The parameters are the masses of the individual objects. There are 100 data values and 100 parameters.

[48]:
```
# Lets find the matrix G such that d=Gm
#= NB: Julia's compiler apparently has better performance for functions than␣
 ↪top level code
We don't need the performance gains here but I'll do it anyway to develop the␣
 ↪habit.
Likewise, we'll use a sparse array. It won't matter for the relatively small␣
 ↪matrices we're working with here,
but it may help if the function is used for something large.

Using a dense matrix causes an out of memory error with a large matrix␣
 ↪(N=100000).
An alternate version of the below code involved insertions into a sparse␣
 ↪matrix. That takes ~5.7s with N=100000. It can be viewed on commit 4c7e7ae
The following version is harder to read but handles N=100000 in ~0.0088s.

Yes, I know all this optimisation wasn't necessary. But it was fun!
=#

using SparseArrays
```

```
"""Make NxN square matrix and fill it with triplets of a coefficent a.
In a row with index r, the cells [r, r], [r, r-1], [r, r-2] are filled.
Argument a multipes the entire matrix by scalar a.
For example TripletsMatrix(5, 1) produces the sparse matrix
   [1 0 0 0 0;
    1 1 0 0 0;
    1 1 1 0 0;
    0 1 1 1 0;
    0 0 1 1 1]"""
function TripletsMatrix(N::Int, a=1::Number)::SparseMatrixCSC  # a is probably␣
↪unneccessary (vs hardcoding a=1) except in some odd cases, but including it␣
↪is trivial.
    #= We are creating a matrix with COO, a coordinate list.
    For some k, row[k] gives row index, col[k] gives col index and val[k]␣
↪gives the value for that coordinate pair.
    We create the matrices in advance then increment over vector coordiantes␣
↪with k because it is reportedly faster than appending to matrices. =#

    if N < 2 error("N should be at least 2.") end

    col = Vector{Int64}(undef, 3*(N-1)) # 3*(N-1) should be the amount of␣
↪nonzero values <- 3 from the first two rows and 3*(N-2) from the rest.
    row = Vector{Int64}(undef, 3*(N-1))
    val = Vector{Float64}(undef, 3*(N-1))
    k = 1 # This tracks our location in the COO vectors
    for i = 1:N # Row index.
        for j in i-2:i # Column index, for non-zero values. We don't write to␣
↪any columns higher than i so this doesn't do anything weird in the last row.
            if j > 0 # To handle first two rows.
                col[k] = j
                row[k] = i
                val[k] = a
                k += 1
            end
        end
    end
    sparse(row, col, val) # Takes our vectors and turns them into a sparse␣
↪matrix
end;
```

[40]:
```
N = 100 # Number of objects
G = TripletsMatrix(N)

display(G) # Julia prints larges sparse matrices in this dot format that merely␣
↪highlights non-zero values. This is much more compact than the full matrix.
```

```
println()
println("""While this appears to show only two values in some places inspection␣
 ↪of those rows, as below, shows this is not the case.
    I was confused as to why, until a friend realised it uses Braille␣
 ↪characters to represent the structure, so it isn't a perfect representation.
 ↪""")
println("Row 4, non-zero values:")
println(G[4,:])
println()

# What % of G is 0?
# nnz returns number of stored values. This can include zero if you have told␣
 ↪it to store 0 somewhere! dropzeros can be handy to remove nonstructual zeros.
percent = (1 - nnz(G) / N^2) * 100
println(string(percent) * "% of G is zero")
```

```
100×100 SparseMatrixCSC{Float64, Int64} with 297 stored entries:
```

```
While this appears to show only two values in some places inspection of those
rows, as below, shows this is not the case.
I was confused as to why, until a friend realised it uses Braille characters to
represent the structure, so it isn't a perfect representation.
Row 4, non-zero values:
  [2  ]  =  1.0
  [3  ]  =  1.0
  [4  ]  =  1.0
```

```
97.03% of G is zero
```

3 Q2 | Problem 1.2

We are again measuring subsets of objects. The data is the height measurements of the subsets (stacks of objects). The parameters are the heights of the individual objects. This gives us 50 measurements and 50 parameters.

[3]:
```
#= We'll just create NxN matrix filled with our coefficent and make a lower␣
 ↪triangular matrix from it
I experimented with sparse matrices but they weren't a good fit here. You can␣
 ↪only really start using them after LowerTriangular so it just hurts␣
 ↪performance. =#

using LinearAlgebra

N = 50
G = LowerTriangular(fill(1, (N, N)))
display(G)

percent = (1 - count(!iszero, G) / N ^ 2) * 100
println(string(percent) * "% of G is zero")
```

```
50×50 LowerTriangular{Int64, Matrix{Int64}}:
 1                                  …
 1  1
 1  1  1
 1  1  1  1
 1  1  1  1  1
 1  1  1  1  1  1                    …
 1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1        …
 1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1  1  1  1      1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1  …   1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1  …   1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1      1  1  1  1  1  1  1  1  1  1
```

```
 1  1  1  1  1  1  1  1  1  1  1  1  1     1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1     1  1  1  1  1  1  1  1  1  1  1  1

49.0% of G is zero
```

4 Q3 | Problem 1.3

[4]:
```
z = Vector(LinRange(0, 10, 11)) # linear spacing, start=0, stop=10, # of␣
 ↪values=11
G = [ones(11) z z.^2 z.^3]
display(G)
```

```
11×4 Matrix{Float64}:
 1.0   0.0    0.0     0.0
 1.0   1.0    1.0     1.0
 1.0   2.0    4.0     8.0
 1.0   3.0    9.0    27.0
 1.0   4.0   16.0    64.0
 1.0   5.0   25.0   125.0
 1.0   6.0   36.0   216.0
 1.0   7.0   49.0   343.0
 1.0   8.0   64.0   512.0
 1.0   9.0   81.0   729.0
 1.0  10.0  100.0  1000.0
```

5 Q4 | Problem 1.4

[49]:
```
#= Unfortunately we aren't so lucky as to be able to directly reuse␣
 ↪TripletsMatrix. Or rather, we could, but the tweaking we'd have to do␣
 ↪after would be suboptimal.
This particular solution handles the endpoints by only overaging two values.
=#


"""Make NxN square matrix and fill it with triplets of coefficents so as to␣
 ↪average groups of three neighbouring points.
In non-boundary row with index r, the cells [r, r], [r, r-1], [r, r-2] are␣
 ↪filled with 1/3.
The first (last) row has been instead given 1/2 in the first (last) two columns␣
 ↪so as to average two parameters.
Argument a multipes the entire matrix by scalar a.
For example RunningAverageMatrix(5) produces the sparse matrix
   [0.5  0.5  0    0    0;
    0.3  0.3  0.3  0    0;
    1    0.3  0.3  0.3  0;
    0    0    0.3  0.3  0.3;
```

```
    0    0    0    0.5  0.5]
where 1/3 has been truncated to 0.3 for readability.
"""
function RunningAverageMatrix(N::Int, a=1::Number)::SparseMatrixCSC
    # Avoid a potential error
    if N < 3
        error("Arguement N should be at least two.")
    end

    # This is COO again
    T = 3 * N - 2 # 3*(N-2)+4 = 3*N-2 nonzero values <- 2 and 2 from the inital␣
↪and final rows and 3*(N-2) from the rest.
    col = Vector{Int64}(undef, T)
    row = Vector{Int64}(undef, T)
    val = Vector{Float64}(undef, T)
    k = 3 # This tracks our location in the COO vectors. We start at 3 because␣
↪we're skipping the first two values for now.
    for i = 2:(N-1) # Row index. We start with the second row
        for j in i-1:(i+1) # Column index, for non-zero values.
            col[k] = j
            row[k] = i
            val[k] = a
            k += 1
        end
    end

    # Hardcoding is ugly but it works. I didn't need to keep the vector indices␣
↪corresponding with the LTR order of elements in the matrix but I find it␣
↪more clear.
    col[1] = 1
    row[1] = 1
    val[1] = (3/2)*a
    col[2] = 2
    row[2] = 1
    val[2] = (3/2)*a
    col[T-1] = N-1
    row[T-1] = N
    val[T-1] = (3/2)*a
    col[T] = N
    row[T] = N
    val[T] = (3/2)*a

    (1/3) * sparse(row, col, val) # Takes our vectors and turns them into a␣
↪sparse matrix
end;
```

[50]:
```
N = 7
G = RunningAverageMatrix(N)
display(G)

percent = (1 - nnz(G) / N^2) * 100
println(string(percent) * "% of G is zero. This percentage increases with the␣
 ↪size of the matrix.")
```

```
7×7 SparseMatrixCSC{Float64, Int64} with 19 stored entries:
 0.5       0.5
 0.333333  0.333333  0.333333
           0.333333  0.333333  0.333333
                     0.333333  0.333333  0.333333
                               0.333333  0.333333  0.333333
                                         0.333333  0.333333  0.333333
                                                   0.5       0.5
```

```
61.224489795918366% of G is zero. This percentage increases with the size of the
matrix.
```

6 Q5 | Problem 1.5

We know that

$$\mathbf{S} = \mathbf{CF}$$

Typically $\mathbf{S}$ is $s \times c$, $\mathbf{C}$ is $s \times p$ and $\mathbf{F}$ is $p \times c$ where s is the number of samples, c is the number of chemicals and p is the number of end-member rocks.

Given the constraints in the problem $s = 1$ so $\mathbf{S}$ and $\mathbf{C}$ are row vectors and $\mathbf{F}$ is some matrix. $\mathbf{S}$ is the data, $\mathbf{C}$ and $\mathbf{F}$ are typically parameters. However, we know the compositions of the p factors is known we can remove them from the parameters and consider this information an auxilary variable. Then it becomes clear that by taking the transpose of the entire equation we get

$$\mathbf{S}^T = (\mathbf{CF})^T = \mathbf{F}^T\mathbf{C}^T$$

Where $\mathbf{S}^T = \mathbf{d}$, $\mathbf{F}^T = \mathbf{G}$ and $\mathbf{C}^T = \mathbf{m}$.

This assumes we know the composition of the end factors exactly.

7 Q6 | Problem 2.1

[7]:
```
using Distributions

u = Uniform()
println("Mean: " * string(mean(u)))
println("Variance: " * string(var(u)))
```

```
Mean: 0.5
Variance: 0.08333333333333333
```

Or manually (note use of bar for expected value - it renders better)

$$\begin{aligned}\bar{d} &= \int_{-\infty}^{\infty} dp(d)\mathrm{d}d \\ &= \int_{0}^{1} d\mathrm{d}d \\ &= \left[\frac{d^2}{2}\right]_0^1 \\ &= \frac{1^2}{2} - \frac{0^2}{2} = \frac{1}{2}\end{aligned}$$

and

$$\begin{aligned}\sigma^2 &= \int_{-\infty}^{\infty} (d-\bar{d})^2 p(d)\mathrm{d}d \\ &= \int_{0}^{1} (d-0.5)^2 \mathrm{d}d \\ &= \left[\frac{1}{3}(d-0.5)^3\right]_0^1 \\ &= \frac{1}{3}\left((1-0.5)^3 - (0-0.5)^3\right) \\ &= \frac{1}{3}\left(0.5^3 - (-0.5)^3\right) \\ &= \frac{1}{12} \approx 0.08333\end{aligned}$$

8 Q7 | Problem 2.2

$$\begin{aligned}E &= d^2 \\ d &= \pm\sqrt{E} \\ \left|\frac{\mathrm{d}d}{\mathrm{d}E}\right| &= \frac{1}{2\sqrt{E}}\end{aligned}$$

So using

$$p(m) = p[d(m)]\left|\frac{\mathrm{d}d}{\mathrm{d}m}\right|$$

we get

$$p(E) = \frac{1}{2\sqrt{2\pi E}} \exp\left[\frac{(\sqrt{E}-\bar{d})^2}{2\sigma^2}\right]$$

9 Q8

Through looking at a very recent (2021) article and then going back to the sources, I came accross *Estimating stellar mean density through seismic inversions* by Reese, Marques, Goupil, Thompson, and Deheuvels (https://doi.org/10.1051/0004-6361/201118156). The paper proposes ways to determine stellar mean density from measurements of stellar pulsation frequencies. In particular, it provides a framework for designing kernal based linear inversions that yield this density. It also presents three inversion techniques for this task and applies them to various test cases.

The data is discrete stellar pulsation frequencies. These are themselves derived from missions like COROT, which monitors variation in the brightness of stars caused by pulsation. The measurement of brightness and the subsequent conversion to frequencies are the first sources of error. In particular, surface effects may cause unrelated brightness changes. As best I can tell, Fourier analysis is used to take the measured spectrum and derive the frequencies of modes.

Dimensional analysis suggests that these frequencies scale with $\sqrt{GM/R^3}$ and $< \Delta v > \propto \sqrt{\rho}$ where $< \Delta v >$ is the large frequency seperation.

One method discussed, SOLA, obtains relative mean density variation $\delta\bar{\rho}/\bar{\rho}$ (a discrete parameter) by minimising a particular cost function (eq 16 of article). This serves to minimise the difference between an averaging kernal and a particular target function $T(x) = 4\pi x \frac{\rho}{\rho_R}$. This particular method avoids having to determine the entire density variation profile, a continous parameter. Note ρ_R is density from a reference model.

The authors did not appear to address questions of existence, uniqueness or instability. This may be because the SOLA method is well established and these questions have been addressed in prior works.

[ ]: