pset5

September 7, 2017

1 18.06 Pset 5

Due Wednesday, March 15.

1.1 Problem 1

(From Strang, section 10.1, question 17.) Suppose A is a 12×9 incidence matrix from a connected (but unknown) graph.

1. How many columns of A are independent?
2. What condition on f makes it possible to solve $A^T y = f$. (Give an explicit formula, don't just say $f \in C(A^T)$ or similar.)
3. The diagonal entries of $A^T A$ give the number of edges into each node. What is the sum of those diagonal entries?

1.2 Problem 2

In class, we represented a graph by an incidence matrix. Another way to represent a graph by a matrix is to use an adjacency matrix.

For a graph with n nodes, the adjacency matrix A is a *square* $n \times n$ matrix, where $A_{i,j} = 1$ if there is an edge from node i to node j, and is zero otherwise.

For example, consider the example graph used in lecture: Its adjacency matrix is:

```
In [ ]: A = [0 0 0 1 0 0
             1 0 0 0 0 1
             0 1 0 0 0 0
             0 0 0 0 1 1
             0 0 0 0 0 1
             0 0 1 0 0 0]
```

1.2.1 part (a)

Consider a vector $x = (1, 0, 0, 0, 0, 0)$, which has a nonzero entry only in node 1. If we multiply this by A^T one or two times, we get:

```
In [ ]: x = [1,0,0,0,0,0]
        A' * x

In [ ]: A'^2 * x
```

- What is the smallest power n of A^T such that $x^T (A^T)^n x > 0$? Why?
- How does this change if we use A instead of A^T?

1.2.2 part (b)

The smallest power of A that gives an entry > 1 is A^4, for which the $(2,6)$ entry is equal to 2. This is because node 2 and 6 are connected by ???????? of length 4.

```
In [ ]: A^4
```

1.3 Problem 3: Nonlinear equations and Newton's method

18.06 is about linear systems of equations, but often real-world problems involve *nonlinear* equations. But linear algebra is still useful in such cases! Often, we solve nonlinear equations by *approximating* them by a *sequence* of *linear* equations. One common such technique is the multidimensional Newton's method. You will explore an application of that in this problem.

Turn in a printout of any code you write and plots you make along with your solutions.

Suppose we are solving the system of nonlinear equations:

$$x_1^3 - 3x_1x_2^2 + x_1 - 1 = 03x_1^2x_2 - x_2^3 + x_2 = 0$$

in two real variables x_1 and x_2. We can also write this as a vector equation $v(x) = 0$, where

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \qquad v(x) = \begin{pmatrix} x_1^3 - 3x_1x_2^2 + x_1 - 1 \\ 3x_1^2x_2 - x_2^3 + x_2 \end{pmatrix}.$$

The key to Newton's method is this: given a *guess* for x, we *approximate* our function v for *nearby* $x + \delta$ by a *linear* approximation, given by the Taylor expansion of v to first order. In particular, if

$$\delta = \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix},$$

the multidimensional Taylor expansion takes the form:

$$v(x + \delta) = v(x) + J(x)\delta + \cdots$$

where $J(x)$ is some matrix.

1.3.1 part (a)

Write down an explicit **equation for the entries of the matrix** $J(x)$ in the Taylor expansion of our function $v(x)$ above.

Recall from 18.02, that we can Taylor expand a function $f(x_1, x_2)$ as

$$f(x_1 + \delta_1, x_2 + \delta_2) = f(x_1, x_2) + \left.\frac{\partial f}{\partial x_1}\right|_{x_1,x_2} \delta_1 + \left.\frac{\partial f}{\partial x_2}\right|_{x_1,x_2} \delta_2 + \cdots .$$

If you apply this to each component of $v(x)$, you should get a formula for the matrix $J(x)$.

1.3.2 part (b)

Write a Julia function `J(x)` that, given a vector `x`, returns the matrix $J(x)$ using your formula from (a). For example, here is a function `v(x)` returning the column vector $v(x)$:

```
In [ ]: v(x) = [   x[1]^3-3*x[1]*x[2]^2+x[1]-1
                   3*x[1]^2*x[2]-x[2]^3+x[2]    ]
```

In a similar style, implement the function $J(x)$ returning a 2×2 matrix:

```
In [ ]: J(x) = [  ???    ???
                  ???    ???  ]
```

1.3.3 part (c)

To get a Newton step, we replace x by $x + \delta$, where δ is the solution to

$$v(x) + J(x)\delta = 0 \; .$$

That is, we approximate $v(x + \delta)$ by the first-order Taylor expansion, and use this to approximately solve $v(x + \delta) \approx 0$.

Write a Julia function `newtonstep(x)` that takes a vector x and returns the Newton step $x + \delta$ given by the solution to the first-order (linear!) equation above. (Recall that if you are solving $Ax = b$, the solution x in Julia is `A \ b`.) Use your functions `v(x)` and `J(x)` from (b)!

```
In [ ]: newtonstep(x) = some formula for x+δ
```

1.3.4 part (d)

It turns out that there are exactly 3 real solutions to $v(x) = 0$

$$x_1 = 0.6823278038280195, \; x_2 = 0$$

and

$$x_1 = -0.3411639019140098, \; x_2 = \pm 1.1615413999972526$$

If you give an initial guess of $(0.5, 0.1)$, you should see that the Newton steps rapidly converge towards the first solution.

Run the following code to interactively change the number of Newton iterations and see the results. **How many Newton iterations do you need to get $x_1 = 0.6823278038280195$ with 4 correct digits? How many for 10 digits?**

```
In [ ]: # Pkg.add("Interact") # uncomment this line if you haven't installed Interact yet
        using Interact
        @manipulate for iterations in slider(0:30, value=0)
            x = [0.5, 0.1]
            for i = 1:iterations
                x = newtonstep(x)
            end
            Text("x₁ = $(x[1]),  x₂ = $(x[2])")
        end
```

(If it doesn't converge to the right answer, you made a mistake in your steps above! **Go fix it before continuing!**)

1.3.5 part (e)

What solution the Newton steps converge to (or whether they converge at all!) depends strongly on our initial guess. If the initial guess is close to a solution, it will converge to that solution, but if the initial guess is far away then the behavior is much less predictable.

The following code tries the Newton iterations for different starting points (x_1, x_2), and uses a different color depending on which root was found. (It plots the exact solutions as stars.) **What do you notice about the pattern of colors, especially on the *boundaries* between the regions that converge to one solution or another?**

(You might want to change the range of x_1 and x_2 that are plotted to zoom in on the boundaries.)

```
In [ ]: # do repeated newton steps, starting at x, until
        # the solution converges or maxiters is reached,
        # returning the tuple: (solution x, number of iterations)
        function newton(x, maxiters=100)
```

```
        for i = 1:maxiters
            xnew = newtonstep(x)
            xnew == x && return (x, i)
            x = xnew
        end
        return (x, maxiters)
    end
```

In [ ]:
```
# Pkg.add("PyPlot")  # uncomment this line if you haven't installed PyPlot yet
using PyPlot
```

In [ ]:
```
x1 = linspace(-2,2,400) # 400 equally spaced points from -2 to 2
x2 = linspace(-2,2,400) # 400 equally spaced points from -2 to 2

# perform Newton iterations and plot the results for all (x1,x2)
results = [newton([x1,x2]) for x2 in x2, x1 in x1]
pcolor(x1', x2, map(r -> atan2(r[1][2], r[1][1]), results), cmap="Blues")

# plot stars for the locations of the exact solutions:
sols = [0.6823278038280195  0.0
       -0.3411639019140098  1.1615413999972526
       -0.3411639019140098 -1.1615413999972526]
plot(sols[:,1], sols[:,2], "r*")

axis("square")
xlabel(L"x_1")
ylabel(L"x_2")
title("color = Newton solution; stars = solutions")
```

It is also interesting to plot the number of iterations that were required to converge as a function of (x_1, x_2):

In [ ]:
```
pcolor(x1', x2, map(r -> r[2], results))
plot(sols[:,1], sols[:,2], "r*")
axis("square")
xlabel(L"x_1")
ylabel(L"x_2")
colorbar()
title("number of Newton iterations to converge")
```