ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Intership Notes

FÉLIX BERGERON
1976613

SUMMER OF 2022

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Contents

2 Documentation - 25/08/2022

Disclaimer: This is a draft and the code that is presented contains comments and commented code lines that aren't necessarily relevant. The latest version of the code can be accessed at **https://github.com/Bergolint/STAGE.git**

2.1 The function emulator

For a system of vector and matrices, the function outputs something that looks like this :

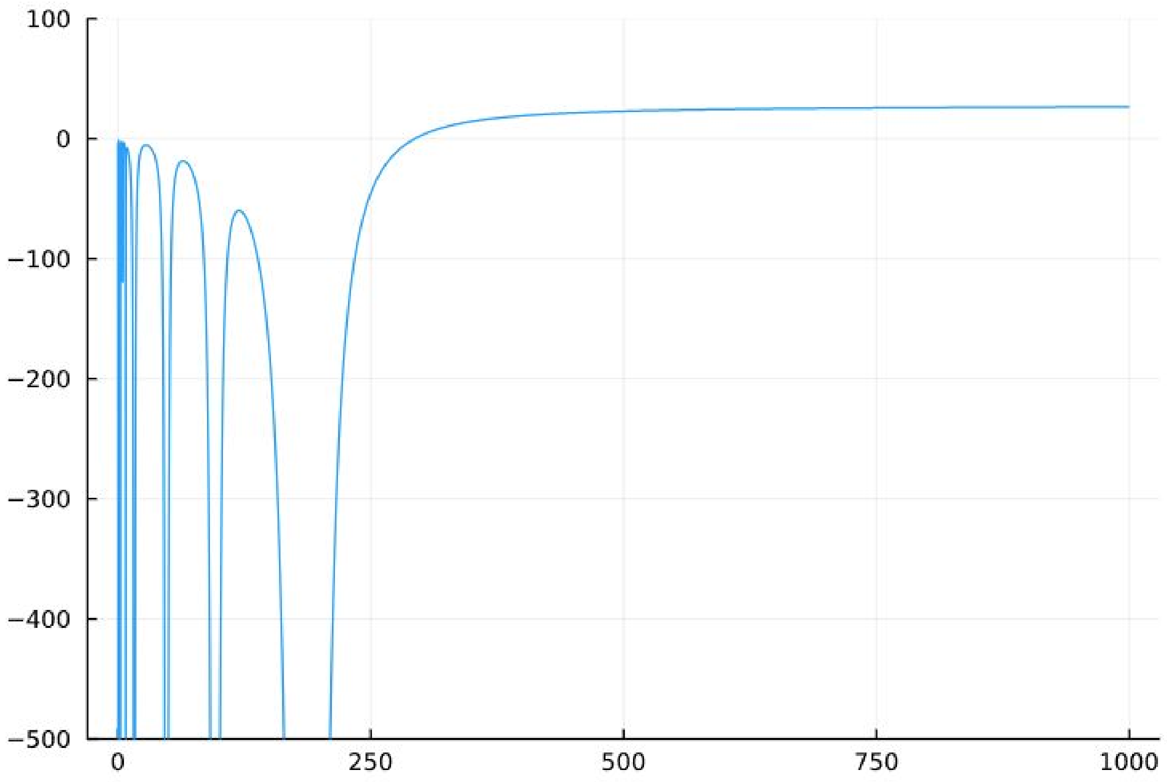

Figure 1: Output function

This emulates the function that the root finding algorithm is eventually going to be used on. The code is simply :

```
function f(x)
    t = inv(A_0 + x*A_1)*(s_0+x*s_1)
    f_0 =  2*real(adjoint(t)*s_1) - adjoint(t)*A_1*t
    return f_0[1]
```

```
end
```

Here A_0 is an indefinite hermitian $n \times n$ matrix and A_1 is a positive definite hermitian $n \times n$ matrix. s_0 and s_1 are n length vectors.

2.2 Random Matrix/Vector generator

In order to test if the root finding algorithm is preforming well, it is useful to be able to create many situations on which we can test our algorithm. This is why it is important to generate these random systems. Here is the code that is used to do just that. It outputs random matrices and vectors of dimensions n.

```
module Random_Matrix

using LinearAlgebra
export indefinite, definite

function indefinite(n) # n is the size of the output matrix
    bool = false
    while bool == false
        n_p = 0 #number of positive eigenvalues
        n_n = 0 #number of negative eigenvalues
        #construct the matrix
        A_0 = rand(n,n)
        Ap = A_0 + adjoint(A_0)
        L = Diagonal(rand(-10:10, n,n))
        global A = Ap + L
        #check eigenvalue to see if is it indefinite
        eig = eigvals(A)
        for i in eig
            if i < 0
                n_n += 1
            elseif i > 0
                n_p += 1
            end
        end
        if n_p == 0 || n_n == 0
            bool = false
        else
            bool = true
        end
    end
    return A
end
```

```
function definite(n)
    R = rand(n,n)
    L = LowerTriangular(R)
    A = adjoint(L)*L
    A = A+ 0.01*I #peak disparity factor
    return A
end

end
#end of random matrix generator module


n = 100
A_0 = Random_Matrix.indefinite(n)
A_1 = Random_Matrix.definite(n)
s_0 = rand(n,1)
s_1 = rand(n,1)
```

Notes:

1. The 0.01 adjustment factor in the definite function is there to modulate the spread of the peaks of the emulated function. A small value of this factor results in a bigger spread of the peaks (because the peaks are the eigenvalues and we are taking the inverse of the matrix to build the emulated function).

2.3 Padé approximant algorithm

This algorithm was simply copied from the paper by V.K.Murthy titled "Systolic algorithm for rational interpolation and Padé approximation". Here is the code written in Julia with modifications and some qol functionalities:

```
function Pade(x, y; N = 500, xl = 0.0, xr = xmax, rebuild_with = [])
    #Padé approximant algorithm
    x = x
    r = y
    l =  length(x)
    R= zeros(l)
    X= zeros(l)
    P= zeros(l)
    S= zeros(l)
    M= zeros(l)

    for i in range(1,l) #first loop
        R[i] = r[i]
        X[i] = x[i]
    end

    for j in range(1,l)#second loop
        P[j] = R[j]
        for s in range(j+1, l)
            S[s] = R[j]
            R[s] = R[s] - S[s]
            if R[s] == 0
                #println("Huston, we have a problem, ABORT.")
            else
                M[s] = X[s] - X[j]
                R[s] = M[s]/R[s]
                if j-1 == 0 # to avoid indexing at the P[0] position
                    R[s] = R[s]
                else
                    R[s] = R[s] + P[j-1]
                end
            end
        end
    end

    function rebuild(x)  #rebuild the approximation from the little blocks
        A = zeros(l)
        B = zeros(l)
        A[1] = P[1]
        B[1] = 1
        A[2] = P[2]*P[1] + (x-X[1])
        B[2] = P[2]
        for i in range(2, l-1)
            A[i+1] = A[i]*(P[i+1] -P[i-1]) + A[i-1]*(x-X[i])
            B[i+1] = B[i]*(P[i+1] -P[i-1]) + B[i-1]*(x-X[i])
        end
        if isinf(abs(A[l])) == true || isinf(abs(B[l])) == true || isnan(abs(A[l])) == true
            throw(Error) #not sure what to do when this happens yet, problems occur when N e
        else
            return A[l]/B[l]
        end
    end
    local px
    if isempty(rebuild_with)== true
        px = [i for i in range(xl, xr, N)]
        approx = map(rebuild, px)
    else
        px = rebuild_with
        approx = map(rebuild, px)
    end
    return (px, approx)
end
```

Notes:

1. For some reason, the Padé approximant sometimes predict peaks (very narrow ones) where there isn't any on the actual function. To see them, it is required to use very small step uniform sampling, otherwise they are unnoticeable. The sampled points don't even suggests that peaks could be there but it still happens. Perhaps this is a bug. Maybe the Padé algorithm wasn't rewritten correctly (perhaps the indexing of some of the building blocks ?). This can cause issues when conducting tests on the Padé approximant (convexity test can give false negative, etc.). As of right now, the best way to circumvent this is to sample with big steps so that we miss these abnormal peaks. This way it is unlikely that a convexity check on the Padé approximant is going to fail due to these peaks. These peak could also be a feature. Sampling close to them seems to make them disappear, so perhaps nothing needs to be done about them.

2. When using more than 336 points to build the Padé approximant (this value may be different depending on the context ... I havn't done enough tests to completly understand the behavious of this problem), the Padé approximant sort of breaks. This is due to how the Padé is computed, with ratios of quantities (see the Padé expression in the systolic algorithm paper). These quantities become very large, you get a very big numerator and a very big denominator (on the order of 10^{300}), which breaks once the power exceeds the computer's limit. This issue can be fixed rather easily, but since it is unlikely we'll need so many points to build a Padé, this isn't that big of a deal. Also, the issue is linked to evaluating the Padé near the last values from which it was constructed. Let's say a point a $x = 200$ is used (while using a large number of points to build the Padé $N \approx 300$), evaluating the Padé approximant at a value close or above $x = 200$ will cause the problem talked about above, big powers that the computer can't handle. So be careful when evaluating a Padé far from where it was constructed, this issue might arise even when using few points (I haven't test this).

3. The rebuilding of the Padé approximant should be made into a separate function to avoid having to evaluate the points to build the Padé everytime we need to use the approximation.

2.4 Root finding algorithm

As of now, the algorithm used is the bisection method. It was used out of convenience without any real thoughts behind it. It would probably be a good idea to implement a different algorithm and choose it carefully.

2.4.1 Bisection method

Here is the bissection method coded in Julia :

```
#x1: starting point to the left
#x2 : starting point to the right
#: Maximum error we want between the actual zero and the approximated one
#N: Maximum number of iteration before returning a value

    function bissection(x1,x2,,N)
        local fxm
        xm = (x1 + x2)/2
        compt = 0
        while abs(x2-x1)/(2*abs(xm)) >  && compt < N
            xm = (x1 + x2)/2
            ans = Pade(big_x_sampled[end],big_y_sampled[end],rebuild_with=[x1,x2,xm])
            ys = ans[2]
            fx1 = ys[1]
            fx2 = ys[2]
            fxm = ys[3]
            if  fx1*fxm < 0
                x2 = xm
            elseif fxm*fx2<0
                x1 = xm
            end
            compt +=1
        end
        return (xm, fxm)
    end
```

Notes:

1. The code was modified to access the Padé approximant constructed from the already sampled points. The issue is that for every evaluation of the function, the Padé approximant is rebuilt and reevaluated. This is because of the way the Padé approximant code was written. This could be made much more efficient.

2.5 Convexity tests

Two functions have been written to do convexity tests. The first one was convexity_test, the second one was peak_finder.

2.5.1 Convexity test

Because of the way I coded this test, it is likely as inefficient as it can get. There is a triple embedded loop that looks at every points to check if a point lies above or below a line formed by two other points. This results in a lot of evaluations. Also, for some reason, the test sometimes fails when it shouldn't. Seeing as the peak_finder test seems to work better and isn't as expensive, I did not bother to try and fix the issues with this test.
Here is the code :

```
function convexity_test(x,y, display_plot = false)
    local xl = 0
    local i = 1
    while i < length(x) #while loop to easily skip iteration once we've found a point of non
        for j in range(i+2, length(x))
            for k in range(i+1,j-1)
                p1 = (x[i], y[i])
                p2 = (x[k], y[k])
                p3 = (x[j], y[j])
                slope = (p3[2] - p1[2])/(p3[1] - p1[1])
                intercept = p1[2] - slope*(p1[1])
                Y = slope*p2[1] + intercept
                if display_plot  == true
                    small_x = [i for i in range(p1[1], p3[1],10)]
                    small_y = [(slope*i + intercept) for i in small_x]
                    plt = plot(px,py, ylim=(ymin,ymax), legend = false)
                    #plot!(x,y, markershape = :circle, color = :black)
                    plot!(small_x, small_y)
                    plot!([p1[1]], [p1[2]], markershape = :circle, color = :green)
                    plot!([p3[1]], [p3[2]], markershape = :circle, color = :blue)
                    plot!([p2[1]], [p2[2]], markershape = :circle, color = :red)
                    display(plt)#necessary to display the graph (equivalent to plt.show() in
                    #sleep(0.1)  #delay to allow for the graph to display
                end
                if Y > p2[2]
                    println("sampled point  ",p2[2])
                    println("pedicted point ", Y)
                    println("___________")
                    xl = x[k]
                    i+=1
                end
            end
        end
        i+=1
    end
    return xl
end
```

2.5.2 Peak finder

This test is simple. It simply evaluates the slope for each consecutive set of points. Convexity is likely broken once :

1. the slope changes sign.
2. the absolute value of the slope at one point is greater than the previous slope but is smaller than the next slope.

Once one of those criteria is respected, the value of the point at which the convexity breaks is added to the list called *peaks*. The list of peaks is then utilized by other functions.

Here is the code :

```
function peak_finder(xs, ys)  #finds the peaks within the sampled point
    slope_criteria = 2 #this needs to be more dynamic, to account for the amount of sampled
    peaks = []
    slopes = []
    for i in range(2, length(xs))
        added = false #verify if a peak has been added twice
        slope =  (ys[i]-ys[i-1])/(xs[i]-xs[i-1])
        push!(slopes, slope)

        # println(i)
        # println(slope)
        # println(slopes)
        # Test : This one seems like a bad test to conduct, doesnt narrow down the position
        # if slope > slope_criteria
        #     push!(peaks,xs[i])
        #     # println(slope)
        #     # println(" ")
        # end

        #Test 1: Checks a sign change in the slope
        if i > 2

            if (slopes[i-2]\slopes[i-1]) < 0 && ys[i-2] < -5 #change of sign of the slope, i
                push!(peaks, xs[i-1]) #note: we take xs[i-1] instead of xs[i-2] because the
                added = true
            end
        end
        #Test2 : Checks if the slope before and after a point stopped growing (indicating th
        if i > 3
            if abs(slopes[i-3]) < abs(slopes[i-2]) && abs(slopes[i-2]) > abs(slopes[i-1]) &&
                push!(peaks, xs[i-1])
            end
        end
    end
    return peaks
end
```

Notes:

1. There is a criteria that may seem odd in the if statements. The $ys[] < -5$ criteria is something that may not be required. Originally the peak_finder test was not used as a simple convexity check, but rather as a peak finder ! The test would, if it weren't for the $ys[] < -5$ criteria, return region of the function that were not necessarily peaks. This criteria can and should then probably be removed since peak_finder is the convexity check test that is currently being used.

2.6 Initial sampling - First test

This function initiates the whole algorithm and samples a few points left to right in a binary search-like way until it finds a value of the function below 0. Once a negative value is found, a few more points are taken. These points may be unnecessary, more testing is needed. The code also verify that the Padé approximant builded from these sampled points doesn't predict peaks into the region that is above the x axis. If peaks are predicted, the function currently doesn't do anything. Additional steps could be added.

Here is the code :

```
function First_test(x_start)
    xs = Any[x_start]
    ys = Any[f(x_start)]
    xr = x_start
    local xl = 0
    xn = x_start/2
    #check if our starting point gives us a value above 0
    ###### Need completing #####
    if ys[end] <= 0
        while ys[end] <=0
            x_start = 2*x_start
            push!(xs, x_start)
            push!(ys, f(x_start))
        end
    else #if our starting point was adequate, samples from right to left
        while ys[1] > 0
            insert!(xs,1,xn)
            insert!(ys,1,f(xn))
            xn = xn/2
        end
    end
    #one last check to see if it’s still above zero later on
    x_start = 1.5*x_start
    push!(xs, x_start)
    push!(ys, f(x_start))
    if ys[end]<0
        println("Need to check this scenario")
    end
    #check how many points we already have sampled to determine
    #how much more sampling needs to be done
    #(we’re going to do one sampling in between every already sampled points)
    xns = Any[]
    for i in range(1, length(xs)-1)
        push!(xns, (xs[i+1]+xs[i])/2)
    end
    for i in range(1, length(xns))
        for j in range(2, length(xs))
            if xns[i] < xs[j] && xns[i] > xs[j-1]
                insert!(xs, j, xns[i])
                insert!(ys,j,f(xns[i]))
            end
        end
    end
    #add a random point for test purposes (for the above test)
    # insert!(xs,3, 90)
    # insert!(ys,3,40)
    # insert!(xs,4, 95)
    # insert!(ys,4,70)
    #Use the padé to see if peaks are predicted in the area that is supposed to be convexit
    x_pade, y_pade = Pade(xs,ys)
    peaks = peak_finder(x_pade, y_pade)

    ########################
    ##### Some investigating needs to be done to determine what causes
    ##### convexity_test to give false positives when checking the padé_approx
    ####  This issue doesnt arrise with Peak_finder
    # convexs1 = convexity_test(xs,ys)
```

```
    # convexs2 = convexity_test(x_pade, y_pade, true)
    # println(peaks)
    # println(convexs2)

    #determine which region is above the x axis
    above_region = xs
    for i in range(1,length(xs))
        if ys[i] < 0
            above_region = xs[i:end]
        end
    end
    #check if there is a predicted peak wihtin the "above" sampled region
    if isempty(peaks) == false
        for i in range(1, length(peaks))
            if peaks[i] > above_region[1] && peaks[i] < above_region[end]
                println("There is a predicted peak in the above region, do something about i
            end
        end
    end
    #Determines the right and left bound in which more sampling might be needed
    xr = above_region[2]
    for i in range(1,length(xs))
        if ys[i] < 0
            xl = xs[i]
        end
    end
    return xs, ys, xl, xr
end
```

2.7 Sampling loop - Big Booiii

This is where most of the tests and sampling is done. First, First_test is called to sample a few points. The next sampled points are determined by doing a convexity check using the function Peak_finder on the Padé approximant. The function is then sampled at the x value that corresponds to the last element of the peaks that were returned by peak_finder (which is the predicted peak furthest to the right). The sampling keeps going until a stopping criteria is met. Currently, the stopping criteria is that for one set of sampled points to another, the difference between the two Padé approximation (built from these set of points) decreases twice in a row. This means that for each additional sampled points, our approximation is getting better and better. The algorithm is still in pre-pre-[..]-alpha, so this might not be the best stopping criteria, but for the limited tests that were conducted it was performing adequately. Here is the code :

```
xs, ys , xl, xr = First_test(xr)
global x_sampled = xs
global y_sampled = ys
global compt = 0


function big_boiii(display_plot = false)
    local plt
    if display_plot ==true
        plt = plot(x_sampled,y_sampled, markershape = :circle, color = :green, linewidth = (
        plot!(px,py)
    end
    #the stopping critera is that the error between padés need to diminish
    #for two consecutives  sampling
    while stop_crit == false
        global xs, ys, compt
        local index
        index = 1
        pade_x, pade_y = Pade(x_sampled, y_sampled, N =5000)
        peaks = peak_finder(pade_x, pade_y)
        #println(peaks)
        #sample near the peak but far enough from last sampled point (inbetween both values)
        # lastpeak = peaks[end]
        # plastpeak = peaks[end-1]
        # next = 0
        # ####finds what sampled point is closest to the suspected peak
        # #### in order to sample close to it (inbetween)
        # for i in x_sampled
        #     if i > lastpeak
        #         next = i
        #         break
        #     end
        # end

        # ####this needs cleaning up (xn1, xn2,xn3), might be too much to sample 3 times eac
        # #### also these variables names are confusing
        # xn1 = (lastpeak+next)/2 #this needs cleaning up
        # xn2 = (plastpeak+next)/2
        # xn3 = lastpeak
        # xs = [xn3, xn1]
        # ys = [f(xn3),f(xn1)]
        xn = last(peaks)
        xs = [xn]
        ys = [f(xn)]
        ###### Adds the new sampled values in order to the already sampled ones (in ascendin
        ###### This could be made into it’s own function ...
        for i in range(1, length(xs))
            for j in range(2, length(x_sampled))
                if xs[i] < x_sampled[j] && xs[i] > x_sampled[j-1]
                    insert!(x_sampled, j, xs[i])
                    insert!(y_sampled,j,ys[i])
                    index = j
                end
                if xs[i] < x_sampled[j] && j ==2
                    insert!(x_sampled, 1, xs[i])
                    insert!(y_sampled,1,ys[i])
                    index = j
                end
                if xs[i] > x_sampled[j] && j==length(x_sampled)
                    insert!(x_sampled, j+1, xs[i])
                    insert!(y_sampled,j+1,ys[i])
                    index = j
                end
            end
        end
        #checks if this sampling has changed something about our evaluation
        pade_stop_criteria(x_sampled[index:end], y_sampled[index:end],xn)
        #plots the different padés (if needed)
        if display_plot==true
            for i in range(1,length(big_x_sampled))
                local pade_x, pade_y
                N=5000
                pade_x, pade_y = Pade(big_x_sampled[i],big_y_sampled[i],N=N,xl = xn)
                plot!(pade_x,pade_y)
                plot!(x_sampled,y_sampled, markershape = :circle, linewidth = 0, color = :gr
            end
            display(plt)
            sleep(0.3)
        end
        compt+=1
    end




    #We can narrow the position of the zero using our sampled points
    println(x_sampled)
    for i in range(1, length(x_sampled))
        sleep(1)
```

```
        #println(x_sampled[end-i+1])
        if y_sampled[i]<0
            xl = y_sampled[i]
        end
        if y_sampled[end-i+1] < 0
            println(x_sampled[end-i+1])
            xr = x_sampled[end-i+2]
        end
    end
    println(xl," ", xr)
    #Finds the zero using the last Padé approximant (bissection method)
    zeros = bissection(xl,xr,10^(-10),100)
    #println("The real root is : ", 65.52353857114213)
    return zeros[1]
end
```

Notes:

1. A stopping criteria based on the position of the last root needs to be implemented. If additional sampling doesn't change the position of the zero by a significant amount (value to be determined), the sampling should stop.
2. A sampling method that is informed by the estimated slope could be implemented instead of binary search.
3. Implement a Monte Carlo simulation to test the success rate of the algorithm for various $n \times n$ systems and saves the system for which the algorithm failed. This implies that an inefficient method to determine for sure the position of the last zero needs to be implemented. A likely candidate would be a right to left small step sampling.

2.8 To do / Bugs / potential issues

1. A stopping criteria based on the position of the last root needs to be implemented. If additional sampling doesn't change the position of the zero by a significant amount (value to be determined), the sampling should stop.
2. A sampling method that is informed by the estimated slope could be implemented instead of binary search.
3. Implement a Monte Carlo simulation to test the success rate of the algorithm for various $n \times n$ systems and saves the system for which the algorithm failed. This implies that an inefficient method to determine for sure the position of the last zero needs to be implemented. A likely candidate would be a right to left small step sampling.

4. Global and local variables are sort of a mess. I went with what worked, the code works ... But most functions use the same name to treat important variables. Some of those variables also have a global and local scope. Perhaps this could raise issues. One problem that could arise is that from an iteration to the next, a variable isn't recalled due to some "if" statements not triggering, which would have the last iterations value still associated to the variable. It happened a few times, but usually it did break the program and was easy to identity. Some of these errors might still be lurking. An easy fix would be to put some of these functions within their own modules.