In [2]:

```
using LinearAlgebra, BenchmarkTools, ApproxFun, ComplexPhasePortrait, TaylorSer

```

In this Julia notebook we implement the algorithms discussed in the main text of the project. First off is the Guassian redution algorithm. Note that we break up reduction to row-echelon and reduced row echelon form in to two different functions and then combine them inside the 'invert' function. We do this for clarity and to make analysis less convoluted, however, we could get a better performance out of our code if we simply implemented the entire algorithm in a single invert function, thus eliminating some extra function calls that cause overhead.

In [3]:

```
function row_ech(A; Aug::Bool=false) #This is a function that reduces A to row-
    A = float(A)
    if Aug
        #determining the dimensions  of mxn matrix to be reduced
        m = size(A,1)
        n = size(A,2)÷2

        else
        m = size(A,1)
        n = size(A,2)
        end

    for lever = 1:(n-1)


        if A[lever, lever] == 0
            ##Here we make sure that the lever is non-zero by swapping rows
            for j = lever+1:m
                if A[j, lever] != 0
                    row = A[lever,:]
                    A[lever, :] = A[j, :]
                    A[j, :] = row
                    break
                    end
                end
            end

        if A[lever, lever] == 0
            ##If every entry on a column is zero we move on to the next column
            continue
            end


        for i = (lever+1):m
            """This loop performs the process of Gaussian elimination until
            every entry below A[i,i] is zero for all i i.e. matrix is upper tri
            multp = -A[i,lever]/A[lever,lever]
            A[i,lever:end] = A[i,lever:end] + multp * A[lever,lever:end]

            end
        end
    return A
    end
```

Out[3]:

```
row_ech (generic function with 1 method)
```

In [4]:

```
function reduced_row_ech(A; Aug::Bool=false)
    #This function takes in a matrix in row-echelon form and takes it to rre fo
    A = float(A)
    if Aug
        #determining the dimensions  of mxn matrix to be reduced
        m = size(A,1)
        n = size(A,2)÷2

        else
        m = size(A,1)
        n = size(A,2)
        end

    for lever = n:-1:2

        if A[lever, lever] == 0

            for j= lever:-1:1
                if A[j, lever] != 0
                    row = A[lever, :]
                    A[lever, :] = A[j, :]
                    A[j, :] = row
                    break
                    end
                end
            end
        if A[lever, lever] == 0
                continue
                end
        for i = lever-1:-1:1
            multp = -A[i, lever]/A[lever, lever]
            A[i,lever:end] = A[i,lever:end] + multp*A[lever,lever:end]

            end
        end
    for i = 1:m
        if A[i,i] != 0

            A[i,:] = (1/A[i,i])*A[i,:]
            end
        end
    return A
    end
```

Out[4]:

```
reduced_row_ech (generic function with 1 method)
```

In [5]:

```
function invert(A)
    aug = row_ech([A I], Aug = true)
    aug = reduced_row_ech(aug, Aug = true)
    return aug[:, size(A,2)+1:end]
    end
```

Out[5]:

invert (generic function with 1 method)

In [6]:

```
function exp_sqr(x,n::Integer)
    """This function calcuates the n-th power of x using the exponentiation by
    bin_str = string(n, base = 2)
    sqrs = zeros(typeof(x[1,1]), size(x,1), size(x,2))
    sqrs_arr = fill(sqrs, length(bin_str))
    sqrs_arr[1] = x
    x_exp = I
    if n%2 != 0
        x_exp = x
    end

    for i = 2:length(bin_str)

        sqrs_arr[i] = sqrs_arr[i-1]*sqrs_arr[i-1]
        if reverse(bin_str)[i] == '1'
            x_exp = x_exp * sqrs_arr[i]
            end

        end
    return x_exp
    end

```

Out[6]:

exp_sqr (generic function with 1 method)

In [7]:

```
function horner_eval(A, F)
    F = reverse(F)
    S = F[1]*A + F[2]*I
    for i = 3:size(F,1)
        S = A*S + F[i]*I
        end
    return S
end
```

Out[7]:

horner_eval (generic function with 1 method)

In [1]:

```
function trapz_rule_N(f, A, N)
    #=This function approximates f(A) using N terms in the trapezium
    rule to approximate the Cauchy integral definition. f must be analyitc on
    the unit circle and γ will be a circular contour such that it encapsulates
    R = opnorm(A,2)

    θ = 0:2π/(N):2π

    Γ = (R+0.1)*exp.(im*θ) #this is the contour of integration

    Γ_prime = im*Γ
    fun_arr = f.(Γ)
    Id_arr = [I*γ for γ in Γ]
    mat_arr = [Id - A for Id in Id_arr]
    invert_mat_arr = [invert(K) for K in mat_arr]

    summand_arr = [Γ_prime[i] * fun_arr[i] * invert_mat_arr[i] for i = 1:N]




    return 1/(im*N+1)* sum(summand_arr)
    end
```

Out[1]:

trapz_rule_N (generic function with 1 method)

In [8]:

```
function affine(a, N)
   #=Helper function. essentially assigns a variable, call it t, which the Tayl
    f can then be expressed in=#
    return a + Taylor1(typeof(a), N)
    end
```

Out[8]:

affine (generic function with 1 method)

In [9]:

```
function taylor_app_N(f, A, N)
    #=This function calculates f(A) using the Taylor expansion definition to th
    t = affine(0.0, N)
    taylor_approx = f(t)


    return horner_eval(A, taylor_approx.coeffs)
    end

```

Out[9]:

```
taylor_app_N (generic function with 1 method)
```

In [14]:

```
A = rand(ComplexF64, 5,5)
ρ = opnorm(A, 2)
A = A/ρ
f = z -> exp(A)

f_A = trapz_rule_N(f,A, 10000)
opnorm(f_A - f(A))
```

Out[14]:

```
0.00019353895626936324
```

In [91]:

```
f = z -> exp(z^2)
#=Here we are just finding the error of the Cauchy method for N = 1:1000:100000
f_A = f(A)

yy = []
xx = 1:1000:100000
for i in xx
    f_AN = trapz_rule_N(f, A, i)
    append!(yy, opnorm(f_A - f_AN,Inf))
    end

```

In [94]:

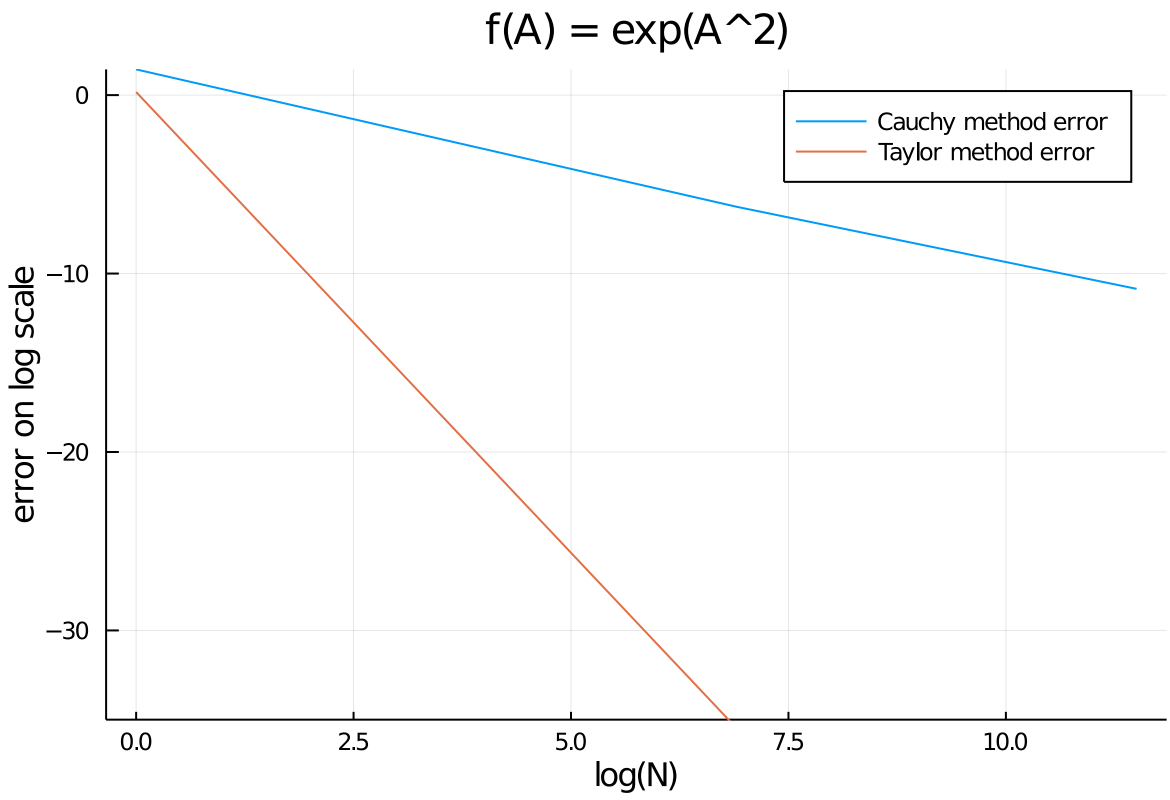
```
yy_taylor = []
#=And for the Taylor method...=#
for i in xx
    f_AN = taylor_app_N(f, A, i)
    append!(yy_taylor, opnorm(f_A - f_AN,Inf))
    end

```

In [95]:

```
#=Plotting the results=#
plot(log.(xx), hcat(log.(yy), log.(yy_taylor)), title = "f(A) = exp(A^2)",
    label = ["Cauchy method error" "Taylor method error"], xlabel = "log(N)", y
ylims = (-35, Inf))
```

Out[95]:



In [306]:

```
function give_rand_toeplitz(n)
    #=This function returns a random nxn toeplitz matrix with 1's on the main d
    a = rand(ComplexF64, n-1)
    r = rand(ComplexF64, n-1)
    A = zeros(ComplexF64, n,n)
    for i = 1:n
        for j = 1:n
            if i - j < 0
                A[i,j] = a[-(i-j)]
            elseif i - j > 0
                A[i,j] = r[i-j]
                else
                    A[i,j] = 1
                end
            end
        end
    return A
    end


```

Out[306]:

```
give_rand_toeplitz (generic function with 1 method)
```

Below is an attempted implementation of the Trench algorithm. There is a logic error in the code which I could not figure out.

In [307]:

```
function trench_invert(A)
    #=This function *should* invert a toeplitz matrix with non-singular princip
    there is a logic error in the code which I could not find=#
    n = size(A,1)
    R = vcat(reverse(A[1, 2:end]), 1, A[2:end, 1])

    X = zeros(Float64, size(R))
    B = zeros(Float64, n, n)

    λ = 1-R[n-1]*R[n+1]
    X[n] = λ
    X[n-1] = -R[n+1]; X[n+1] = -R[n-1]

    for i = 1:n-2

        η = -R[n-i-1] - transpose(X[n+1:n+i])*R[n-i:n-1]
        γ = -R[n+i+1] - transpose(R[n+1:n+i])*X[n-i:n-1]

        X[n+1:n+i+1] = vcat(X[n+1:n+i]+(η/λ)*X[n-i:n-1], η/λ)
        X[n-i-1:n-1] = vcat(γ/λ, X[n-i:n-1]+ (γ/λ)*X[n+1:n+i])

        λ = λ - (η*γ/λ)
        @show 1/λ


        end

    g = reverse(X[1:n-1])
    e = X[n+1:end]
    B[1,1] = 1
    B[1,2:end] = e
    B[2:end,1] = g
    Λ = (g.*transpose(e) .- (reverse(e).*transpose(reverse(g))))


    for i = 2:n
        for j = 2:n
            B[i,j] = B[i-1, j-1] + Λ[i-1, j-1]

            end
        end



    return B/λ
    end



```

Out[307]:

```
trench_invert (generic function with 1 method)
```