MIT 18.338 Term Project: Numerical Experiments on Circular Ensembles and Jack Polynomials with Julia

N. Kemal Ure

May 15, 2013

Abstract

This project studies the numerical experiments related to averages of Jack Polynomials on Circular Ensembles. Main aim of the project is to produce efficient numerical implementation of sampling from circular ensembles and evaluation of Jack polynomials. Julia computing language is utilized for this purpose. Numerical results reinforce the statements of the theorem and it is shown that especially parallel implementation of Julia provides superior computation speed compared to MATLAB and non-parallel Julia implementations.

1 Background

This section provides the basic facts about circular ensembles [1] and Jack polynomials [2].

1.1 Circular Ensembles

Circular ensembles are simply measures on spaces of unitary matrices (i.e. complex square matrices U such that $U^*U = UU^* = I$, where U^* is the conjugate transpose and I is the identity matrix). Circular ensembles are parametrized by $\beta > 0 \in \mathbb{R}$ and the most commonly encountered circular ensembles are,

1. Circular Orthogonal Ensemble (COE), $\beta = 1$, symmetric unitary matrices
2. Circular Unitary Ensemble (CUE), $\beta = 2$, unitary matrices
3. Circular Symplectic Ensemble (CSE), $\beta == 4$, unitary quaternion matrices

Let $U(n)$ represent group of $n \times n$ unitary matrices and let U be a random matrix from drawn according to the Haar measure on this group. Let, $(\lambda_1, ..., \lambda_n)$ be the eigenvalues of the random matrix U. Since U is a unitary

matrix, it's eigenvalues lie on the unit circle thus, $\lambda_k = e^{i\theta_k}$ for some $\theta \in [0, 2\pi)$. It is a known result from random matrix theory that the probability density function of $\theta = (\theta_1, ..., \theta_n)$ is given as

$$p(\theta) = \frac{1}{Z_{n,\beta}} \prod_{1 \leq k < j \leq n} |e^{i\theta_k} - e^{i\theta_j}|^\beta, \tag{1}$$

where $Z_{n,\beta}$ is a normalization constant.

1.2 Jack Polynomials

Jack polynomials are a set of multivariate polynomials parametrized by a parameter $\beta > 0$. Let $P_k(n)$ the space of symmetric homogeneous polynomials of degree k in n variables. For a fixed β, Jack polynomials are indexed according to κ (a partition of k) and they serve as a basis for $P_k(n)$.

There are many different definitions of Jack polynomials, the one that is more useful to us is given with their connection regarding to circular ensembles. Macdonald [3] showed that,

$$\int_{[0,2\pi]^n} J_\kappa^\beta(e^{i\theta_1}, ..., e^{i\theta_n}) \overline{J_\lambda^\beta(e^{i\theta_1}, ..., e^{i\theta_n})} \prod_{j<k} |e^{i\theta_j} - e^{j\theta_k}| d\theta_1 ... d\theta_n = \delta_{\kappa,\lambda}. \tag{2}$$

Eq. 2, in conjunction with p.d.f. in 1 shows that Jack Polynomials orthogonalize the circular ensembles. Let X be an square matrix with eigenvalues $(\lambda_1, ..., \lambda_n)$, and define $J_\kappa^\beta(X) = J_\kappa^\beta(\lambda_1, ..., \lambda_n)$. Thus for a fixed β and two partitions $\kappa \neq \lambda$, we can state that,

$$\mathbb{E}[J_\kappa^\beta(U)\overline{J_\lambda^\beta(U)}] = 0, \tag{3}$$

where the expectation is taken over the random unitary matrix U according to the probability measure defined in Eq. 1. The basic aim of this project is to evaluate the expectation in 3 by Monte-Carlo simulations to verify that expectation indeed converges to zero as the number of samples used in the simulation increases. It is clear that in order to evaluate this expression we need two subroutines that,

1. Sample a random unitary matrix and find its eigenvalues
2. Evaluate the Jack function on these eigenvalues

In the next two sections we give the description of the theoretical properties that are used in the efficient numerical implementation of these subroutines based on the works [4, 5].

2 Sampling from Circular Ensembles

We will focus our attention to a specific class of unitary matrices called unitary upper Hessenberg matrices, which are "almost" upper triangular matrices with all zero entries above below it's first superdiagonal. Here is an example of an unitary upper Hessenberg matrix,

$$\begin{bmatrix} 0.5693 + 0.094i & -0.0042 + 0.0132i & 0.2468 + 0.7785i \\ 0.8168 & 0.0014 - 0.0097i & -0.2614 + 0.5153i \\ 0 & 0.9999 & 0.9999 \end{bmatrix}$$

In the work [4], Ammar et. al showed that there is one to one correspondence between the Schur parameters $\gamma_j \in \mathbb{C}, j = 1, ..., n$ and $n \times n$ upper unitary Hessenberg matrices. The Schur parameters are a finite sequence of complex numbers such that

$$|\gamma_j| \leq 1, 1 \leq j < n, |\gamma_n| = 1. \tag{4}$$

For given Schur parameters $\{\gamma\}$ we can construct the corresponding unitary upper Hesseberg matrix as,

$$H(\{\gamma\}) = G_1(\gamma_1)...G_{n-1}(\gamma_{n-1})\tilde{G_n}(\gamma_n), \tag{5}$$

where G is the Givens reflector [6], which also represents the generalization of this formula to an arbitrary β . Thus to sample a uniformly random unitary upper Hessenberg matrix, we can simply generate a random sequence of Schur parameters in Eq. 4, and then use the formula in 5 to compute the unitary upper Hessenberg matrix.

3 Efficient Evaluation of Jack Polynomials

The Jack polynomials can be expressed in monomial basis through summation over semi-standard Young tableaux (SSYT),

$$J_\lambda^\beta = \sum_{T-SSYT} f_T(\beta) x^T,$$

however this methods is computationally very inefficient. Demmel and Koe [5] developed and algorithm that recursively evaluate the Jack function based on the principal of dynamic programming. Recursion works as,

$$J_\lambda^\beta(x_1, ..., x_n) = \sum_{\mu \leq \lambda} J_\mu^\beta(x_1, ..., x_{n-1}) x_n^{|\lambda/\mu|} c_{\mu\lambda}, \tag{6}$$

where $c_{\mu\lambda}$ is a coefficient that depends on both partitions λ and μ. As it can be seen form the recursion, algorithm first evaluates the Jack functions of $n-1$ variables and uses linear combinations of these polynomials to calculate Jack polynomial in n variables. Koev provided an implementation of the algorithm at his website (http://math.mit.edu/ plamen/software/jack.m) in MATLAB language.

4 Results

4.1 Sampling Unitary Matrices

The algorithm that samples unitary upper Hessenberg matrices (described in Section 2) was implemented in the JULIA language. The code is available in the appendix.

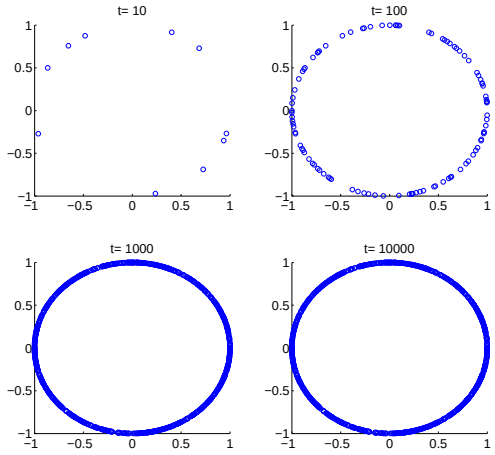

Figure 1: Eigenvalue spread of random unitary Hessenberg matrices for different samples sizes.

Fig. 1 displays the sampled eigenvalues of random unitary matrices (with $n = 3$) on the complex plane for different sample sizes (10,100,10^3 and 10^4). The figure shows that as the sample size increases eigenvalues of the sampled matrices cover the entire unit circle. These figures indicate that the algorithm succeeds in generating random matrices with eigenvalues on the unit circle.

4.2 Monte-Carlo Evaluation of Eq. 3

The recursive Jack polynomial evaluation algorithm described in Section 3 was implemented in JULIA language based on Koev's MATLAB implementation. Then the random matrices sampled with the code described above was fed to this process, and the outputs of the Jack functions were averaged over many samples and different partitions. Fig. 2 shows that as the number of samples increase, the absolute value of expectation in Eq. 3 decays to zero.

Fig. 3 plots the average number of samples required for expectation to converge within $\epsilon = 0.0001$ for different β values. It is shown that lesser number of samples is required for convergence as the value of β increases.

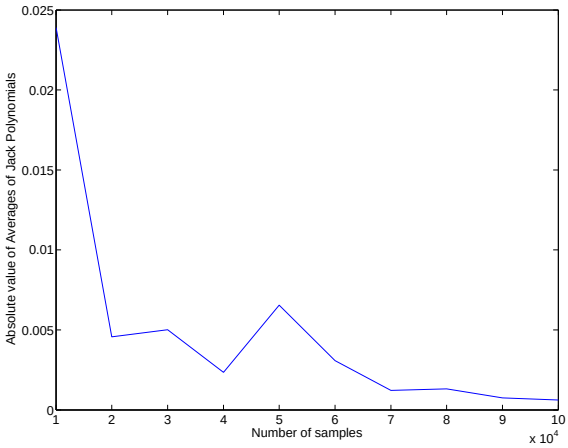

Figure 2: Absolute value of average of jack polynomials on circular ensembles vs. number of samples.

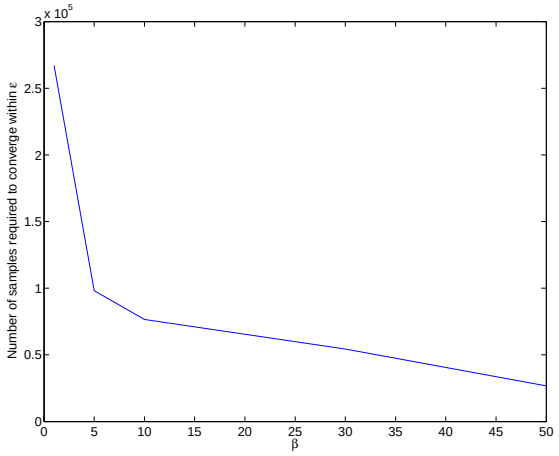

Figure 3: Average number of samples required for expectation to converge within $\epsilon = 0.0001$ for different β values.

4.3 MATLAB - Julia Comparisons

Table 1 shows the running time (in seconds) of MATLAB, Julia and Parallel Julia implementations for the numerical procedure described at the Results section for the calculation of Fig. 2 over different sample sizes. This table clearly indicates the computational power of Julia; for the larger sample values Julia is almost 20 times faster than the MATLAB implementation. Table also show that paralleling the code gives additional computation time saving.

Number of Samples	**MATLAB**	**Julia**	**Parallel Julia (4 Cores)**
500	5.1	1.7	1.3
1000	10.2	1.9	1.8
2000	20.3	2.4	1.9
5000	51.8	3.6	2.1
50000	516.4	25.6	3.9

Table 1: Comparison of running times (in seconds) of MATLAB, Julia and Parallel Julia implementations

References

[1] F.M. Dyson "The threefold way. Algebraic structure of symmetry groups and ensembles in quantum mechanics". J. Math. Phys. 3: 1199. (1962).

[2] Jack, Henry, "A class of symmetric polynomials with a parameter", Proceedings of the Royal Society of Edinburgh, Section A. Mathematics 69: 118, (1971).

[3] Macdonald, I. G., Symmetric functions and Hall polynomials, Oxford Mathematical Monographs (2nd ed.), New York: Oxford University Press, ISBN 0-19-853489-2, MR 1354144 (1995)

[4] Ammar, Gregory, William Gragg, and Lothar Reichel. "Constructing a unitary Hessenberg matrix from spectral data." In Numerical linear algebra, digital signal processing and parallel algorithms, pp. 385-395. Springer Berlin Heidelberg, 1991.

[5] Demmel, James, and Plamen Koev. "Accurate and efficient evaluation of Schur and Jack functions." Mathematics of computation 75, no. 253 (2006): 223-239.

[6] Forrester, Peter J., and Eric M. Rains. "Jacobians and rank 1 perturbations relating to unitary Hessenberg matrices." arXiv preprint math/0505552 (2005).

Appendix

JULIA codes

```
# Calculate emprical averafe of Jack polynomials
# on circular ensembles
require("circular.jl")
require("jack_eval.jl")
 require("samplejackaverage.jl")

n = 3
beta = 7
alpha=2/beta
t = 10000

k1=[ 1  1]
k2=[2]

xx = zeros(1,n)

jj1 = JackEvaluator(k1,xx,alpha)
jj2 = JackEvaluator(k2,xx,alpha)

v = 0 + 0*im

tic()

for i=1:t

    println("At Iteration ",i)
  v = v + samplejackaverage(jj1,jj2,n,beta)

end

println(abs(mean(v)))
toc()

# Parallel implementation of Circular Jack
@everywhere require("circular.jl")
@everywhere require("jack_eval.jl")
@everywhere require("samplejackaverage.jl")

@everywhere n = 3
@everywhere beta =50
@everywhere alpha=2/beta
```

```
#@ everywhere data = zeros(Float64,1,10)
#for j =1:10
t =100

@everywhere k1=[ 1  1]
@everywhere k2=[2]

@everywhere xx = zeros(Complex128,1,n)

@everywhere jj1 = JackEvaluator(k1,xx,alpha)
@everywhere jj2 = JackEvaluator(k2,xx,alpha)
@everywhere k =1
v = 0 + 0*im

tic()


#println(mean(v))

v = @parallel (+) for i=1:t
    samplejackaverage(jj1,jj2,n,beta)

end

println(abs(v/t))

#data[j] =abs(v/t)

#end
#end
#open(readall, "dataa.txt")
#writecsv("data.csv",data)

toc()

require("circular.jl")
require("jack_eval.jl")

function samplejackaverage(JE1,JE2,n,beta)

  e=eigvals(circular(n,beta))

    return     evaluate(JE1,e)*evaluate(JE2,e)'
end

# Code for circular ensemble, translated from A. Edelman's code
```

```
function circular(n,beta)

# Compute Schur Parameters (Verblunsky coefficients)
rn  = rand(1,n-1)         # Random numbers for schur magnitudes
rp  = rand(1,n-1)         # Random numbers for schur phases
e = (beta/2)*[1:(n-1)]    # The Theta_subscripts (v-1)/2

a = Array(Float64,1,n-1)
einv = 1./e
for i = 1:n-1
a[i] = sqrt(1-rn[i]^einv[i])    # Schur parameters from Theta
end
rho = sqrt(1-a.^2)        # Complementary Parameters
a = a .* exp(2*pi*im*rp)  # Random Phase

# Compute unitary upper Hessenberg
z = exp(2*pi*im*rand(1))
for j=1:n-1
    z=[1 zeros(1,j); zeros(j, 1) z]
    G=[a[j] rho[j];rho[j] -a[j]']
    z[1:2,:]=G*z[1:2,:]
end

return z
end

# 18.338 Project
# Convert Plamen Koev's MATLAB code to Julia

# Type Definition

type JackEvaluator

    x
    ja
    alpha
    Lp
    Lmax
    n
    lma
    kappa

    function JackEvaluator(kappa,xx,alpha1)

        x = xx
```

```
        alpha = alpha1
        n = length(x)
        Lp = sum(sign(kappa))
        lma = zeros(Lp,1)
        kappa = kappa[1:Lp]
        Lmax = kappa+1
        ja = Inf*ones(prod(Lmax+1),n) + 0*im*ones(prod(Lmax+1),n)

        new(x,ja,alpha,Lp,Lmax,n,lma,kappa)

    end


end


# Function Definitions

# The main Jack function evaluator
function evaluate(self::JackEvaluator,xeval)

    self.x =xeval
     f = 0.0 + 0.0*im

        if self.Lp>0

                self.lma[self.Lp] = 1

                for i = self.Lp-1:-1:1
                        self.lma[i] = self.lma[i+1]*self.Lmax[i+1]
                end

                # Initialize

                initialize(self,1,0);

                f = jack1(self,self.n,0,nmu(self,self.kappa),nmu(self,self.kappa
        else

                f = 1.0 + 0.0*im

        end

        return f
end
```

```
## Initialize Function

function initialize(self::JackEvaluator,k,l)

        if k<=self.Lp

                m =self.Lmax[k]-1

                if k>1

                        m = min(m,part(self,l,k-1))

                end

                for i =1:m

                        l = l + self.lma[k]
                        self.ja[l+1,1:n] = Inf * ones(1,n) + 0*im*ones(1,n)
                        initialize(self,k+1,l);
                end

        end

end


# Part Function


function part(self::JackEvaluator,pn,i)

        if i >length(self.lma)

                f = 0

        else
                if i == 1
                        f = floor(pn/self.lma[i])
                else
                        f = floor(mod(pn,self.lma[i-1])/self.lma[i])

                end
        end
        #println("Part output : ",f)
        return f
end
```

```
# Jack1 Function

function jack1(self::JackEvaluator,j,k,lambda,l)

    s = 1.0 + 0.0*im

    if 1<=j && l>0

        t = self.ja[l+1,j]

        if k==0 && t != Inf
            s = t

        elseif part(self,l,j+1)>0
            s = 0.0 + 0.0*im

        elseif j==1

            s = self.x[1]^part(self,l,1)*prod(1 + self.alpha*[0:part(self,l,1)-1

        else

            if k ==0
                i = 1
            else
                i = k

            end

            s = jack1(self,j-1,0,l,l)*AB(self,lambda,l)*self.x[j]^lm(self,lambda

            while part(self,l,i)>0

                if part(self,l,i) > part(self,l,i+1)

                    if part(self,l,i)>1

                        s = s + jack1(self,j,i,lambda,l-self.lma[i])
                    else

                       s = s+ jack1(self,j-1,0,l-self.lma[i],l-self.lma[i])*AB(s
                    end
                end

                i = i+1
```

```
            end

        end

        if k == 0

            self.ja[l+1,j] = s

        end

    end
    #println("Jack1 output : ",s)

    return s
end


# AB Function

function AB(self::JackEvaluator,l,m)

    f = 1

    for i =1:self.Lp
        for j=1:part(self,m,i)

        if l_t(self,l,j) == l_t(self,m,j)

            f = f/(l_t(self,m,j) -i + self.alpha*(part(self,m,i) - j + 1))

        else
            f = f/(l_t(self,m,j) -i + 1 + self.alpha*(part(self,m,i) - j ))
        end
       end
    end


    for i =1:self.Lp
    for j=1:part(self,l,i)
        if l_t(self,l,j) == l_t(self,m,j)

            f = f*(l_t(self,l,j) -i + self.alpha*(part(self,l,i) - j + 1))

        else
            f = f*(l_t(self,l,j) -i + 1 + self.alpha*(part(self,l,i) - j ))
```

```
        end
    end
    end
    #println("AB output : ",f)
    return f
end

# lm function

function lm(self::JackEvaluator,l,m)

    f = 0

    for i =1:self.Lp

        f = f+ part(self,l,i) - part(self,m,i)

    end
    #println("lm output : ",f)
    return f
end


# l_t function

function l_t(self::JackEvaluator,l,q)

    i =1
    f = 0

    while part(self,l,i) >= q

        f = f+ 1
        i = i+1

    end
    #println("l_t output : ",f)
    return f
end

# nmu function

function nmu(self::JackEvaluator,l)

    f = 0
```

```
    for i =1:self.Lp

            f = self.Lmax[i]*f

                if  i<= length(l)
                    f = f+l[i]
                end

    end
    #println("nmu output : ",f)
    return f
end
```