CIS 510 HW 1 Submission

Scott Lambert

10/12/2022

Problem 1

Include in *my_solvers.jl* a function *conj_grad*() that performs the conjugate gradient (CG) algorithm (see class notes for pseudocode) to obtain an approximate solution $\tilde{x}$ to the linear system $Ax = b$, where A is a positive definite matrix. *conj_grad*() should take in a positive definite matrix A, an initial guess x_0, a right-hand-side vector b, a tolerance ϵ and a maximum number of iterations $\text{iter}_{\max}$, and return an approximate solution $\tilde{x}$ to $Ax = b$, such that the relative error $err_R \leq \epsilon$, where

$$err_R = \frac{||A\tilde{x} - b||}{||\tilde{x}||}$$

and $||\cdot||$ is the 2-norm, i.e. $||x|| = \sqrt{x^T x}$ (i.e. you need to define the "stopping criterion" $err_R \leq \epsilon$).
Hint: one way to generate a positive definite matrix is through $A = I + B^T B$ as done previously. Also, a common choice for initial guess is the vector of all zeros. Take $\epsilon = 10^{-6}$, $\text{iter}_{\max} = 100$ and try making sure CG returns a reasonable guess for $N = 10, 100$ and $N = 1,000$.

Ideas for discussion: how did you assert that CG code was correct? Did you have to change any parameters of the stopping condition to achieve convergence? Ideas for figure: plot time to solution as a function of N or plot norm of the current residual $||Ax_i - b||$ as a function of iteration i.

Solution

Demonstration of correctness

See the end of this file for the code used for this problem.

I asserted the solution's correctness by plotting the relative error as a function of N (the system size) for various N. I first generated a random symmetric positive-definite matrix A via $A = B^T B + I$ where B is a random NxN matrix (`rand(N,N)`). I then computed a random N-vector x via `rand(N)` and a right-hand side b by multiplying x by A: $b = Ax$. Next I used `conj_grad` to solve $Ax = b$, and plotted the relative error $||x_{\text{CG}} - x_{\text{true}}||/||x_{\text{true}}||$ as a function of the system size N for three separate trials. I repeated this process three times for each value of N because each trial uses a random matrix. The errors are all in the 10^{-7} range, which makes sense given that the stop criterion was that the relative error be less than 10^{-6}. It is noteworthy that the relative error decreases as a function of N.

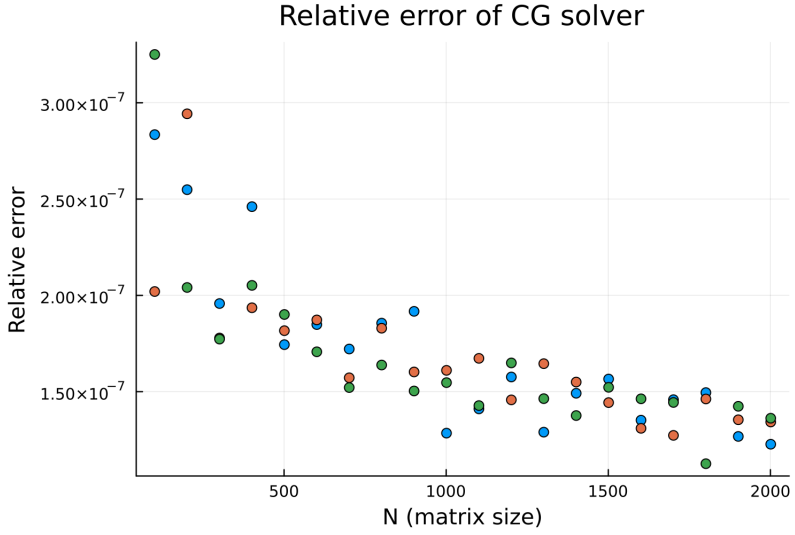

Figure 1: CG solver error for a range of N values. Each color represents a different trial at that value of N.

Performance

The plot below shows the performance of the CG solver as a function of N. For each value of N, three random, symmetric, positive-definite matrices were generated as for the error plot above. The `conj_grad` computation was timed and the times are plotted. Aside from some strange results near N = 1500, the performance appears to follow a smooth curve.

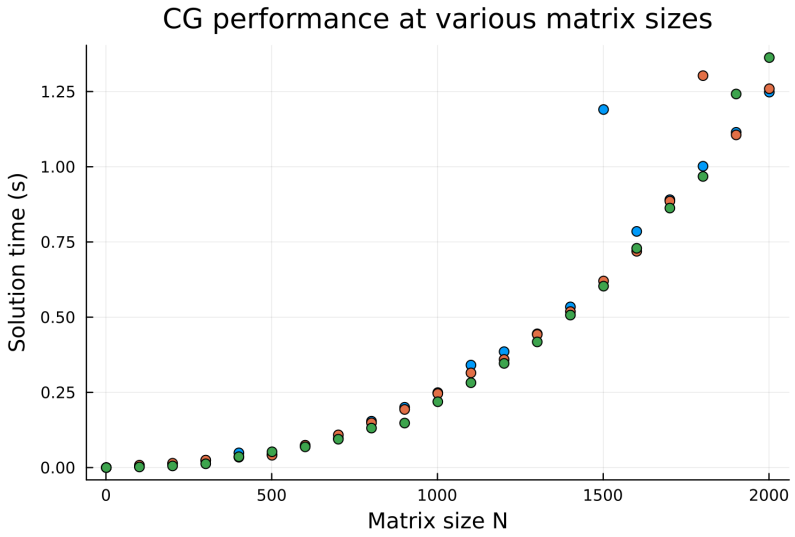

Figure 2: CG solver performance for a range of N values. Each color represents a different trial at that value of N.

Problem 2

Create a new script called *euler_methods.jl* that includes the functions *my_forward_euler*() (you may use/modify mine from class) and *my_backward_euler*()(write yourself) that solve a general IVP

$$y' = \lambda y$$
$$y(0) = y_0$$

for a scalar λ, on the time domain $0 \leq t \leq T_f$ using a time step Δt, where $f(t, y)$ is a known RHS (right-hand-side) function and y_0 is the initial value. Include tests of both functions. For example solve the IVP

$$y' = -3y,$$
$$y(0) = 17$$

on the temporal domain $0 \leq t \leq 4$ using a stable time step and plot the numerical approximation and exact solution $y(t) = 17e^{-3t}$ on the same figure and show they they align upon visual inspection. Do this for both forward and backward Euler.

Ideas for discussion: how did you determine a stable time step for each of the two methods? Ideas for figure: Include plots from example tests above.

Solution

Initial conditions and time interval

I solved the IVP

$$y' = -y$$
$$y(0) = 1$$

over the range $[0, 3]$ using the forward Euler and backward Euler methods. The exact solution is

$$y(t) = e^{-t}$$

Maximum stable timestep

The maximum stable timestep for the forward Euler method was shown in class to be

$$\Delta t_{\max} = -2/\lambda$$

For this problem $\lambda = 1$, so $\Delta t_{\max} = 2$. I chose a timestep of 0.1, which is significantly below the stability threshold.

Plots

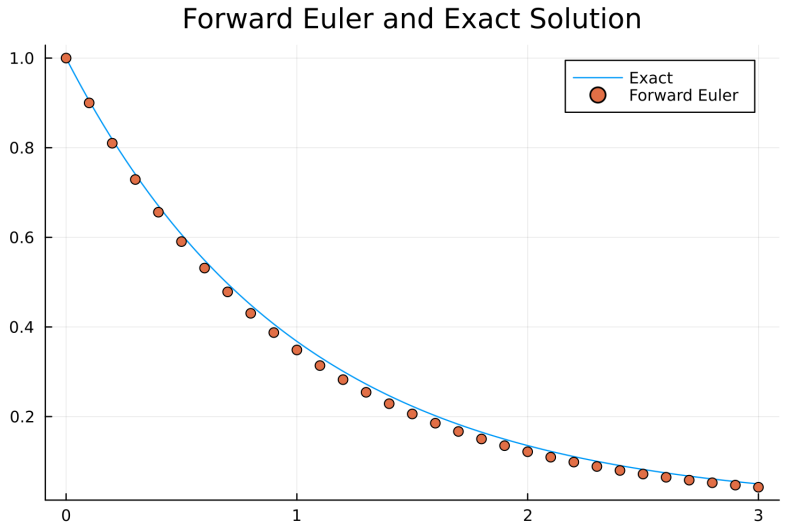

Figure 3: Forward Euler solution showing good agreement with the analytical solution. The numerical solution undershoots the analytical solution. This is to be expected for a solution that is concave-up, since for such a curve the tangent line always lies below the curve.

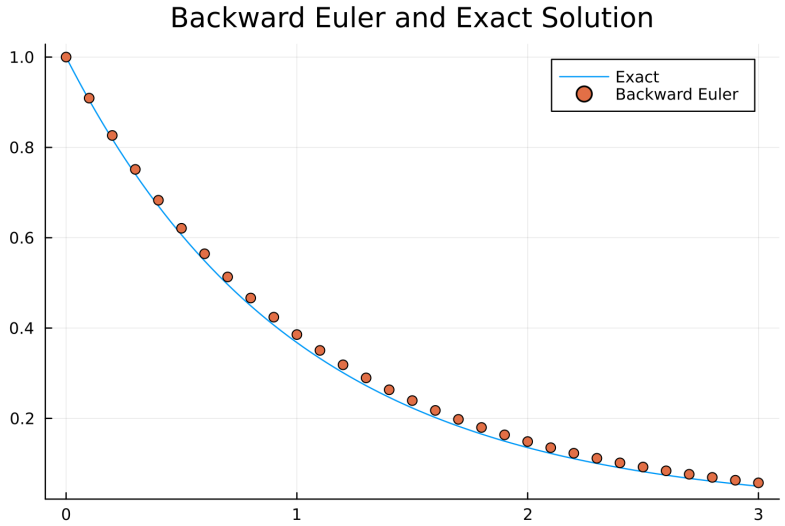

Figure 4: Backward Euler solution with good agreement. In this case the numerical solution slightly overestimates the analytical solution, because the method finds the next y-value by tracing along a line with the slope at that y-value, which is more positive than the slope at the current value.

Problem 3

Consider the $N \times N$ banded, sparse matrix

$$A = \begin{bmatrix} -2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & -2 & 1 & \ddots & & & & \vdots \\ 0 & 1 & -2 & 1 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 1 & -2 & 1 & 0 \\ \vdots & & & & \ddots & 1 & -2 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 & -2 \end{bmatrix}. \tag{1}$$

Write a Julia script called *dense_sparse_lu.jl* that:

(i) creates A in dense array format (e.g. initialize $A = zeros(N, N)$ and loop through rows and columns to fill in values). Use built-in (optimized) Julia functionality (i.e use Julia's LinearAlgebra package) to compute an LUP factorization (e.g. $F = lu(A)$), and then solve the system $Ax = b$ (e.g. $x = F\backslash b$) , where N = 1,000 and b is an $N \times 1$ vector of all 1's. Time the two computations (make sure to time twice and take second time, as first time includes compilation).

(ii) Repeat (i) but define A in sparse array format (do NOT call sparse(A) on dense A. why?) using the SparseArrays package. Comment on speed-up (compared to (i)) for different N.

Ideas for discussion: What matrix data structure (sparse versus dense is better in terms of computational time?)
Ideas for figure: Plot time against N for dense versus sparse matrix formats.

Solution

The matrix A is the symmetric discrete second derivative. Sample output from my function that constructs the matrix is below:

```
julia> ddMatrix(5)
5 5  SparseMatrixCSC{Float64, Int64} with 13 stored entries:
 -2.0   1.0
  1.0  -2.0   1.0
         1.0  -2.0   1.0
                1.0  -2.0   1.0
                       1.0  -2.0

julia> ddMatrix(5,make_sparse=false)
5 5  Matrix{Float64}:
 -2.0   1.0   0.0   0.0   0.0
  1.0  -2.0   1.0   0.0   0.0
  0.0   1.0  -2.0   1.0   0.0
  0.0   0.0   1.0  -2.0   1.0
  0.0   0.0   0.0   1.0  -2.0
```

When `make_sparse = false`, A is not generated by calling sparse() on a dense matrix since this would defeat one of the purposes of using a sparse matrix in the first place, which is to save memory. Instead, A is generated using the compressed sparse column (CSC) format (see source code for details).

Performance comparison with dense and sparse matrices

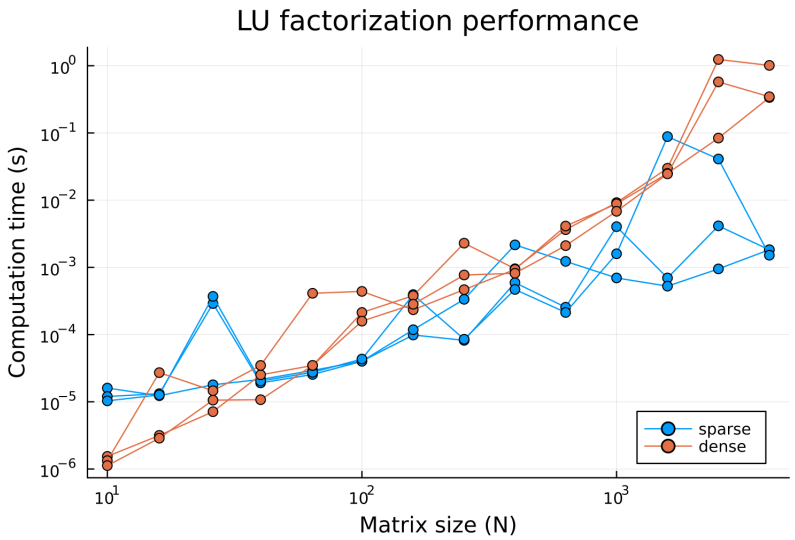

Figure 5: Time required to compute the LU factorization of the discrete second derivative of size N. The three curves of each color are for the three different trials with the associated matrix storage method (dense or sparse).

The figure above shows the time required to compute the LU factorization of dense and sparse second derivative matrices. For small N, the dense matrix computation is faster, but for larger N (starting around 50-100), the sparse computation begins to outperform the dense computation. The time required to solve $Ax = b$ with b a vector of ones is shown below, and appears to scale similarly to the respective LU factorization times for dense and sparse matrices.

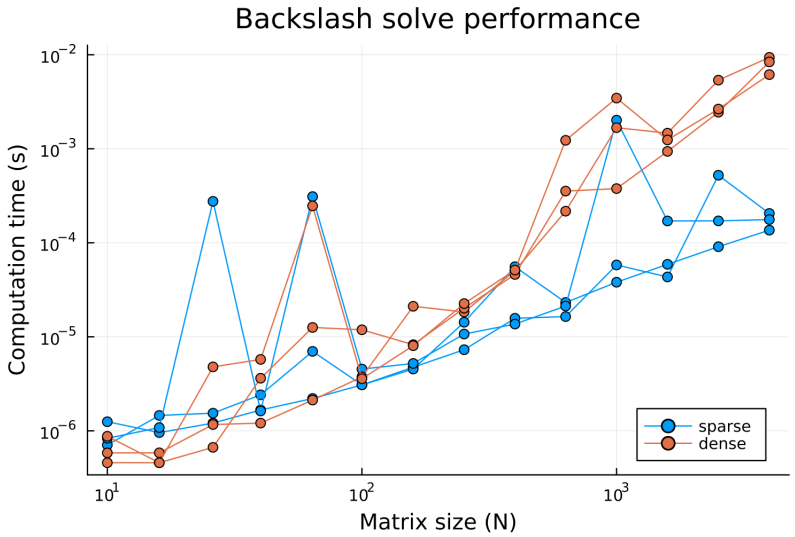

Figure 6: Time required to solve $Ax = b$ with LU factorization. Blue curves are trials when A is sparse, red when it is dense.

Source code

Problem 1

New additions to `my_solvers.jl`

```
function norm(v)
"""
    norm(v)
Compute the norm of the vector (or array) v.
"""
    return sqrt(sum(v.^2))

end

function normsq(v)
"""
    compute the squared norm of v
"""
    return sum(v.^2)
end

function matconj(v,A)
"""
    compute the conjugated inner product
transpose(v)*A*v
"""
    return transpose(v)*A*v
end

function spdrmat(N)
    """
        pdtestmat(N)
    Return a symmetric positive-definite random NxN
    matrix
    """
    B = rand(N,N)
    return B'*B + identity(N)
end

function conj_grad(A,x0,b,eps,maxiter)
"""
    conj_grad(A,x0,b,eps,maxiter)
Solve Ax = b using the conjugate gradient method with
initial guess x0.
eps: tolerance
maxiter: maximum number of iterations

Algorithm from Schewchuk, "An Introduction to the
Conjugate Gradient Method Without the Agonizing
Pain, Edition 1 1/4," 1994
"""

    N = size(A)[1]
    iter = 0
    x = x0
    r = b - A*x0 # initial residual
    err =  norm(r)/norm(x0)
    d = r # initial step direction
```

```
    while err >= eps && iter <= maxiter
        alpha = normsq(r)/matconj(d,A)
        x = x + alpha .* d
        rnext = r - alpha.*A*d
        beta = normsq(rnext)/normsq(r)
        d = rnext + beta.*d
        r = rnext
        err = norm(r)/norm(x)
        iter += 1
        #@printf("\niter = %d\n",iter)
        #@printf("err = %f\n",err)
    end
    return x
end
```

Client script for problem 1

```
using Plots
using Printf

include("../my_solvers.jl")


function plotCGSolTime(Ns,eps=1e-6,maxiter=1000;nruns=1)
    """
        plotCGSolTime(Ns)
    plot the time required to solve a linear system
    with the conjugate gradient method.
    Ns = list of matrix sizes
    eps = tolerance
    nruns: number of trial runs
    """
    p = plot(legend=false)
    title!("CG performance at various matrix sizes")
    xlabel!("Matrix size N")
    ylabel!("Solution time (s)")
    for i = 1:nruns
        @printf("run number %d\n",i)
        t = zeros(size(Ns))
        @printf("Solving: N = %d\n",N)
        for (i,N) in enumerate(Ns)
            A = spdrmat(N)
            b = rand(N)
            x0 = zeros(N)
            t[i] = @elapsed conj_grad(A,x0,b,eps,maxiter)
            @printf("Solved with N = %d in %.4e seconds\n",
                N,t[i])
        end
        Plots.scatter!(p,Ns,t)
    end
    display(p)
end

function testCGAccuracy(Ns,eps=1e-6,maxiter=1000;nruns=3)
    """
        testCGAccuracy(Ns)
    Test the accuracy of the CG solver for various
    matrix sizes N. Record the relative error and plot
    it as a function of N.
    Ns: list of matrix sizes
    eps: tolerance
    nruns: number of repetitions
    """
    p = plot(legend=false)
    title!(p,"Relative error of CG solver")
    xlabel!("N (matrix size)")
    ylabel!("Relative error")
    @printf("\n")
    for i = 1:nruns
        @printf("Run number %d\n",i)
        errs = zeros(size(Ns))
        for (i,N) in enumerate(Ns)
            x0 = zeros(N)
            A = spdrmat(N)
```

```
            xtrue = rand(N)
            b = A*xtrue
            x = conj_grad(A,x0,b,eps,maxiter)
            errs[i] = norm(x-xtrue)/norm(xtrue)
            @printf("Solved with N = %d\n",N)
        end
        Plots.scatter!(p,Ns[2:end],errs[2:end])
    end
    display(p)
end

# Solve with LU factorization and CG and display results
N = 5
B = rand(N,N)
A = transpose(B)*B + identity(N)
b = rand(N)

xLU = solvePLU(A,b)

x0 = zeros(N)
eps = 1e-6
maxiter = 1000
xCG = conj_grad(A,x0,b,eps,maxiter)

@printf("LU factorization solution: \n")
display(xLU)
@printf("CG solution: \n")
display(xCG)


# Time CG factorization and plot performance
Ns = 1:100:2001
#plotCGSolTime(Ns,nruns=3)
testCGAccuracy(Ns)
```

Problem 2

euler_methods.jl

```
function my_forward_euler(f,y0,T,dt)
"""
    my_forward_euler(f,y0,tf,dt)
Solve the ODE y'(t) = f(t,y) with initial condition y(0) = y0
on the time interval [0,T]
"""

    N = Integer(ceil(T/dt))
    t = 0:dt:N*dt
    y = zeros(N+1)
    y[1] = y0

    for i = 1:N
        y[i+1] = y[i] .+ f(t[i],y[i]) .* dt
    end
    return t,y
end

function my_backward_euler(A,y0,T,dt)
"
    my_backward_euler(A,y0,T,dt)
Solve the linear ODE y' = Ay via the backward Euler
method with initial condition y(0) = y0 over the
time interval [0,T]
"
    N = Integer(ceil(T/dt))
    t = 0:dt:N*dt
    y = zeros(N+1)
    y[1] = y0

    for i = 1:N
        y[i+1] = (1/(1-A*dt))*y[i]
    end
    return t,y
end
```

Client script for problem 2

```
using Plots

include("../euler_methods.jl")
pwd()
######### Problem 1 ##########
A = -1
y0 = 1
f(t,y) = A.*y
T = 3
dt = 0.1 # max stable timestep is -2/A = 2

# Forward Euler solution
tf,yf = my_forward_euler(f,y0,T,dt)

# Backward Euler solution
tb,yb = my_backward_euler(A,y0,T,dt)

# Exact solution
tex = 0:0.01:T
yex = y0.*exp.(A.*tex)

# Plots
p = plot(tex,yex,label="Exact")
scatter!(tf,yf,label = "Forward Euler",marker=:circle)
title!("Forward Euler and Exact Solution")
display(p)

p = plot(tex,yex,label="Exact")
scatter!(tb,yb,label = "Backward Euler",marker=:circle)
title!("Backward Euler and Exact Solution")
display(p)
```

Problem 3

dense_sparse_lu.jl

```
using LinearAlgebra
using SparseArrays
using Plots

include("../my_solvers.jl")

# Create A in dense or sparse format
function ddMatrix(N;make_sparse=true)
    """
        ddMatrix(N)
    compute the discrete second derivative matrix as either a dense
    array or a sparse array.
    """
    if !make_sparse
        # Create A in dense format
        A = zeros(N,N)
        A[1,1:2] = [-2 1]
        for i = 2:N-1
            A[i,i-1:i+1] = [1 -2 1]
        end
        A[N,N-1:N] = [1 -2]
    else
        # Create A in sparse format
        rowInds = [1:N-1;1:N;2:N]
        colInds = [2:N;1:N;1:N-1]
        vals = [ones(N-1); -2 .* ones(N); ones(N-1)]
        A = sparse(rowInds,colInds,vals)
    end
    return A
end

# Compute and time LU factorization with LinearAlgebra
function timeLU(Ns; make_sparse=true,nruns=1)
    """
        timeLU(Ns)
    Time how long it takes Julia to compute the LU factorization of a
    second derivative matrix as defined in ddMatrix.
    Ns = vector of matrix sizes
    nruns = number of trials for each N value
    make_sparse: whether or not to use a sparse array
    solve: whether or not to time
    returns: t, (length(Ns) x nruns) array of factorization times
    """

    t = zeros(length(Ns),nruns)
    for i in 1:nruns
        for (j,N) in enumerate(Ns)
            A = ddMatrix(N,make_sparse=make_sparse)
            b = ones(N)
            t[j,i] = @elapsed lu(A)
        end
    end
    return Ns,t
end
```

```
# Solve Ax = b and time
function timesolve(Ns; make_sparse=true,nruns=1)
    """
        timesolve(Ns)
    Time how long it takes Julia to solve Ax = b with LU factorization
    when A is a second derivative matrix and b is a vector of ones.
    Ns = vector of matrix sizes
    nruns = number of trials for each N value
    make_sparse: whether or not to use a sparse array
    solve: whether or not to time
    returns: t, (length(Ns) x nruns) array of factorization times
    """

    t = zeros(length(Ns),nruns)
    for i in 1:nruns
        for (j,N) in enumerate(Ns)
            A = ddMatrix(N,make_sparse=make_sparse)
            b = ones(N)
            F = lu(A)
            t[j,i] = @elapsed F\b
        end
    end
    return Ns,t
end
```

Client script for problem 3

```
using Plots
include("dense_sparse_lu.jl")

# Generate plots
functs = [timeLU,timesolve]
titles = ["LU factorization performance","Backslash solve performance"]
for (i,funct) in enumerate(functs)
    Ns = [2;Integer.(ceil.(10 .^collect(1:0.2:3.6)))]
    display(Ns)
    Ns_sparse,tsparse = funct(Ns,nruns=3,make_sparse=true)
    Ns_dense,tdense = funct(Ns,nruns=3,make_sparse=false)
    p=plot(legend=:bottomright,title=titles[i],
    xlabel="Matrix size (N)", ylabel="Computation time (s)")
    plot!(p, Ns_sparse[2:end],tsparse[2:end,:],xaxis=:log,yaxis=:log,marker=:circle,
        color = :1,labels=["sparse" "" ""])
    plot!(p, Ns_dense[2:end],tdense[2:end,:],xaxis=:log,yaxis=:log,marker=:circle,
        color = :2,labels=["dense" "" ""])
    display(p)
end
```