Homework 2: Generalized-Alpha method

Name: Abdelrahman Fathy Abdelhaleem Aly Abdelrahman

Matr.-Nr.: 108023251500


```
begin
    using LinearAlgebra
    using Plots
    using LaTeXStrings
end
```

Task 1:

1.1. From small amplitude vibrations assumption: $sin(\theta) \approx \theta$.

1.2. Then from the given two balance of linear momentum equations, one can formulate the $\boldsymbol{M}$ & $\boldsymbol{K}$ matrices, as follows:

$$\boldsymbol{M} = \begin{bmatrix} \frac{\rho A L^3}{3} & 0 \\ 0 & \frac{\rho A L}{3} \end{bmatrix}, \; \boldsymbol{K} = \begin{bmatrix} \frac{\rho A g L^2}{2} & 0 \\ 0 & \frac{EA}{L} \end{bmatrix}$$

with $\boldsymbol{u}$ & $\ddot{\boldsymbol{u}}$ are vectors, and there components are as follows:

$$\ddot{\boldsymbol{u}} = \begin{bmatrix} \ddot{\theta} \\ \ddot{u} \end{bmatrix}, \; \boldsymbol{u} = \begin{bmatrix} \theta \\ u \end{bmatrix}$$

1.3. For given α_1 & α_2, one can calculate the Rayleigh damping matrix $\boldsymbol{M}$ from the following equation: $\boldsymbol{C} = \alpha_1 \boldsymbol{M} + \alpha_2 \boldsymbol{K}$

1.4. Finally, the semi-discrete equation can be written as follows:

$$\boldsymbol{M} \cdot \ddot{\boldsymbol{u}} + \boldsymbol{C} \cdot \dot{\boldsymbol{u}} + \boldsymbol{K} \cdot \boldsymbol{u} = \boldsymbol{R}$$

with,

$$\boldsymbol{R} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \; \dot{\boldsymbol{u}} = \begin{bmatrix} \dot{\theta} \\ \dot{u} \end{bmatrix}$$

Note

Semi-discrete means that the equation is discrete in space only, but continous in time.

Data inputs:

$$\Delta t = 0.01\ s$$
$$L = 1.0\ m$$
$$\rho A = 1.0\ kg/m$$
$$EA = 500.0\ N$$
$$g = 9.8\ m/s^2$$

```
9.8
```

```
begin
    # Data inputs
    L = 1.0
    ρA = 1.0
    EA = 500
    g = 9.8
end
```

```
(
    M = 2×2 Matrix{Float64}:
         0.333333  0.0
         0.0       0.333333
    K = 2×2 Matrix{Float64}:
         4.9    0.0
         0.0  500.0
    R =  [0, 0]
)
```

```
begin
    # Linear Momentum.
    M = [(ρA*L^3)/3 0; 0  (ρA*L)/3]
    K = [(ρA*g*L^2)/2 0; 0 EA/L]
    R = [0 , 0]
    (;M,K,R)
end
```

```
(
    u₀ =  [0.0, -0.2]
    v₀ =  [1.27802, 0.0]
)
```

```
begin
    # Initial Conditions
    u₀ = [0 ; -L/5] # initial displacement
    v₀ = [sqrt(g/6L) ; 0] # initial velocity
    (;u₀,v₀)
end
```

```
2×2 Matrix{Float64}:
 0.333333  0.0
 0.0       0.333333
```

```
begin
    # Calculate Rayleigh damping
    α_1 = 1.0
    α_2 = 0.0
    C = α_1*M + α_2 * K
end
```

Task 2:

2.1. Calculate the initial conditions $\boldsymbol{u_o}, \boldsymbol{\dot{u}_o}, \boldsymbol{\ddot{u}_o}$ as follows:

$$\boldsymbol{\dot{u}_o} = \begin{bmatrix} 0 \\ -\frac{L}{5} \end{bmatrix}, \; \boldsymbol{\dot{u}_o} = \begin{bmatrix} \sqrt{\frac{g}{6L}} \\ 0 \end{bmatrix} \rightarrow \textbf{both are given in the problem prompt}$$

whereas, $\boldsymbol{\ddot{u}_o}$ can be calculated from the semi-discrete equation of motion as follows:

$$\begin{aligned} \boldsymbol{M} \cdot \boldsymbol{\ddot{u}_o} + \boldsymbol{C} \cdot \boldsymbol{\dot{u}_o} + \boldsymbol{K} \cdot \boldsymbol{u_o} &= \boldsymbol{R} \\ \Rightarrow \boldsymbol{\ddot{u}_o} &= \boldsymbol{M}^{-1} \cdot (\boldsymbol{R} - \boldsymbol{C} \cdot \boldsymbol{\dot{u}_o} + \boldsymbol{K} \cdot \boldsymbol{u_o}) \end{aligned}$$

2.2. For given $\boldsymbol{\rho_\infty}$, calculate the Newmark parameters according to *Chung and Hulbert (1993)* method:

$$\alpha_m = \frac{2\rho_\infty - 1}{\rho_\infty + 1}, \; \alpha_f = \frac{\rho_\infty}{\rho_\infty + 1}, \; \beta = \frac{1}{4}[1 - \alpha_m + \alpha_f]^2, \; \gamma = \frac{1}{2} - \alpha_m + \alpha_f$$

2.2. Calculate $\boldsymbol{K_{eff}}$ from the following equation:

$$\boldsymbol{K_{eff}} = \boldsymbol{M}\frac{1 - \alpha_m}{\beta \Delta t^2} + \boldsymbol{C}\frac{\gamma(1 - \alpha_f)}{\beta \Delta t} + \boldsymbol{K}(1 - \alpha_f) \rightarrow \textbf{(slides pg. 63)}$$

2.3. Loop over time from $\boldsymbol{t_o = 0}$ upto $\boldsymbol{t_{final}}$ with time step = $\boldsymbol{\Delta t}$ and for each step $\boldsymbol{n}$ do:
2.3.1 calculate the effective right hand side from the following equation:

$$\begin{aligned} \boldsymbol{r_{eff}} &= -\boldsymbol{K} \cdot \alpha_f \boldsymbol{u_n} \\ &+ \boldsymbol{C} \cdot \left[\frac{\gamma(1 - \alpha_f)}{\beta \Delta t} \boldsymbol{u_n} + \frac{\gamma - \gamma\alpha_f - \beta}{\beta} \boldsymbol{\dot{u}_n} + \frac{(\gamma - 2\beta)(1 - \alpha_f)}{2\beta} \Delta t \boldsymbol{\ddot{u}_n} \right] \\ &+ \boldsymbol{M} \cdot \left[\frac{1 - \alpha_m}{\beta \Delta^2} \boldsymbol{u_n} \right] \rightarrow \textbf{(slides pg. 63)} \end{aligned}$$

2.3.2. solve for $\boldsymbol{u_{n+1}}$ as follows:

$$\boldsymbol{u_{n+1}} = \boldsymbol{K_{eff}^{-1}} \cdot \boldsymbol{r_{eff}}$$

2.3.3 update velocity and acceleration (i.e. $\boldsymbol{\dot{u}_{n+1}}, \boldsymbol{\ddot{u}_{n+1}}$) from the following equations:

$$\dot{\boldsymbol{u}}_{n+1} = \frac{\gamma}{\beta\Delta t}(\boldsymbol{u}_{n+1} - \boldsymbol{u}_n) - \frac{\gamma-\beta}{\beta}\dot{\boldsymbol{u}}_n - \frac{\gamma-2\beta}{2\beta}\Delta t\ddot{\boldsymbol{u}}_n \rightarrow \text{(slides pg. 60)}$$

$$\ddot{\boldsymbol{u}}_{n+1} = \frac{1}{\beta\Delta t^2}(\boldsymbol{u}_{n+1} - \boldsymbol{u}_n) - \frac{1}{\beta\Delta t}\dot{\boldsymbol{u}}_n - \frac{1-2\beta}{2\beta}\ddot{\boldsymbol{u}}_n \rightarrow \text{(slides pg. 60)}$$

> **Note**
>
> The following structs are used to define the **input parameters** for the *Generalized-Alpha* method.

```
struct LinearMomentum
    M::Matrix{Float64} # mass matrix
    K::Matrix{Float64} # stiffness matrix
    R::Vector{Float64} # load vector (i.e. right hand side)
end
```

```
struct InitialConditions
    u₀::Vector{Float64} # initial displacement
    v₀::Vector{Float64} # initial velocity
end
```

```
# define abstract supertype for damping that will be inherited by
# physcial (i.e. α₁, α₂) and numerical damping (i.e. ρ_∞)
abstract type DampingParameters end
```

```
struct PhysicalDamping <: DampingParameters
    α₁::Float64
    α₂::Float64
end
```

```
struct NumericalDamping <: DampingParameters
    ρ_∞::Float64
end
```

```
struct RunTime
    tf::Float64 # amount of time we solve our system for
    Δt::Float64 # fixed time step or initial time step for the adaptive method.
end
```

> **Note**
>
> The following structs are used to define the **output parameters** for the *Generalized-Alpha* method.

```
struct DynamicResponse
    u::Matrix{Float64} # displacement
    ud::Matrix{Float64} # velocity
    udd::Matrix{Float64} # acceleration
end
```

```
struct Error
    e_abs::Vector{Float64} # absolute error (Zienkiewicz and Xie)
    η::Vector{Float64} # relative error
    e_cum::Vector{Float64} # cummulative error
end
```

```
struct TimeEvolution
    times::Vector{Float64} # current time vector.
    steps::Vector{Float64} # Δt vector
end
```

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

```
function generalized_alpha(moment::LinearMomentum,ic::InitialConditions,
time::RunTime, pd::PhysicalDamping,nd::NumericalDamping)

    M = moment.M # mass matrix
    K = moment.K # stiffness matrix

    α₁ = pd.α₁
    α₂ = pd.α₂

    # calculate damping
    C = α₁*M + α₂ * K

    # Initial conditions
    u_o = ic.u₀
    ud_o = ic.v₀
    udd_o = inv(M) * (R - C * ud_o - K * u_o)

    # Chung and Hulbert (1993)
    ρ_∞ = nd.ρ_∞
    α_m = (2*ρ_∞ - 1)/(ρ_∞ + 1)
    α_f = (ρ_∞)/(ρ_∞+1)
    β = 0.25 * (1 - α_m + α_f)^2
    γ = 0.5 - α_m + α_f


    # create time interval range to loop over later
    tf = time.tf
    Δt = time.Δt
    t = 0:Δt:tf
    n = length(t)
    steps = fill(Δt,n)

    # define our respone containers
    n = length(t)
    u = zeros(2,n)
    ud = zeros(2,n)
    udd = zeros(2,n)

    # set initial conditions in our response containers
    u[:,1] = u_o
    ud[:,1] = ud_o
    udd[:,1] = udd_o

    # define the containers for errors
    e_abs = zeros(n) # absolute error
    η = zeros(n) # relative error
    e_cum = zeros(n) # cummulative error

    K_eff = M * ((1-α_m)/(β*Δt^2)) + C * (γ * (1 - α_f))/(β*Δt) + K * (1 - α_f)
    for i = 1:n-1

        r_eff = -K * α_f * u[:,i] +

```

```
                C * (((γ * (1-α_f))/(β*Δt)) * u[:,i] + ((γ - γ*α_f - β)/(β)) *
                ud[:,i] + (((γ-2*β)*(1-α_f))/(2*β)) * Δt*udd[:,i]) +
                M * (((1-α_m)/(β*Δt^2))*u[:,i] + ((1-α_m)/(β*Δt))* ud[:,i] + ((1-α_m
                - 2 *β)/(2*β)) * udd[:,i])

        # solve for u.
        u[:,i+1] = K_eff \ r_eff

        # update velocity and acceleration
        ud[:,i+1] = ((γ)/(β*Δt)) * (u[:,i+1] - u[:,i]) - ((γ - β)/(β))*ud[:,i] - ((γ
        - 2*β)/(2*β)) * Δt * udd[:,i]

        udd[:,i+1] = (1/(β*Δt^2))*(u[:,i+1] - u[:,i]) - (1/(β*Δt)) * ud[:,i] - ((1-
        2*β)/(2*β)) * udd[:,i]

        # calculate errors
            e_abs[i+1] = norm(((6*β - 1)/(6))*(udd[:,i+1] - udd[:,i]) * Δt^2)
        η[i+1] = e_abs[i+1] / norm(u[i+1] - u[i])
        e_cum[i+1] = sum(e_abs)
    end

    (response = DynamicResponse(u,ud,udd), errors = Error(e_abs,η,e_cum),time =
    TimeEvolution(t,steps))

end
```

Task: Solve the system:

```
(
    momentum =   LinearMomentum(
                    M = 2×2 Matrix{Float64}:
                         0.333333  0.0
                         0.0       0.333333
                    K = 2×2 Matrix{Float64}:
                         4.9    0.0
                         0.0  500.0
                    R =   [0.0, 0.0]
                )
    initial_coditions =   InitialConditions(
                             u₀ =   [0.0, -0.2]
                             v₀ =   [1.27802, 0.0]
                          )
    runtime =   RunTime(
                    tf = 5.0
                    Δt = 0.01
                )
)
```

```
begin
    # common input parameters for both task 3a & 3b
    momentum = LinearMomentum(M,K,R) # linear momentum parameters
    initial_coditions = InitialConditions(u₀,v₀) # initial conditions
    runtime = RunTime(5.0,0.01)
    (;momentum,initial_coditions,runtime)
end
```

Task 3a:

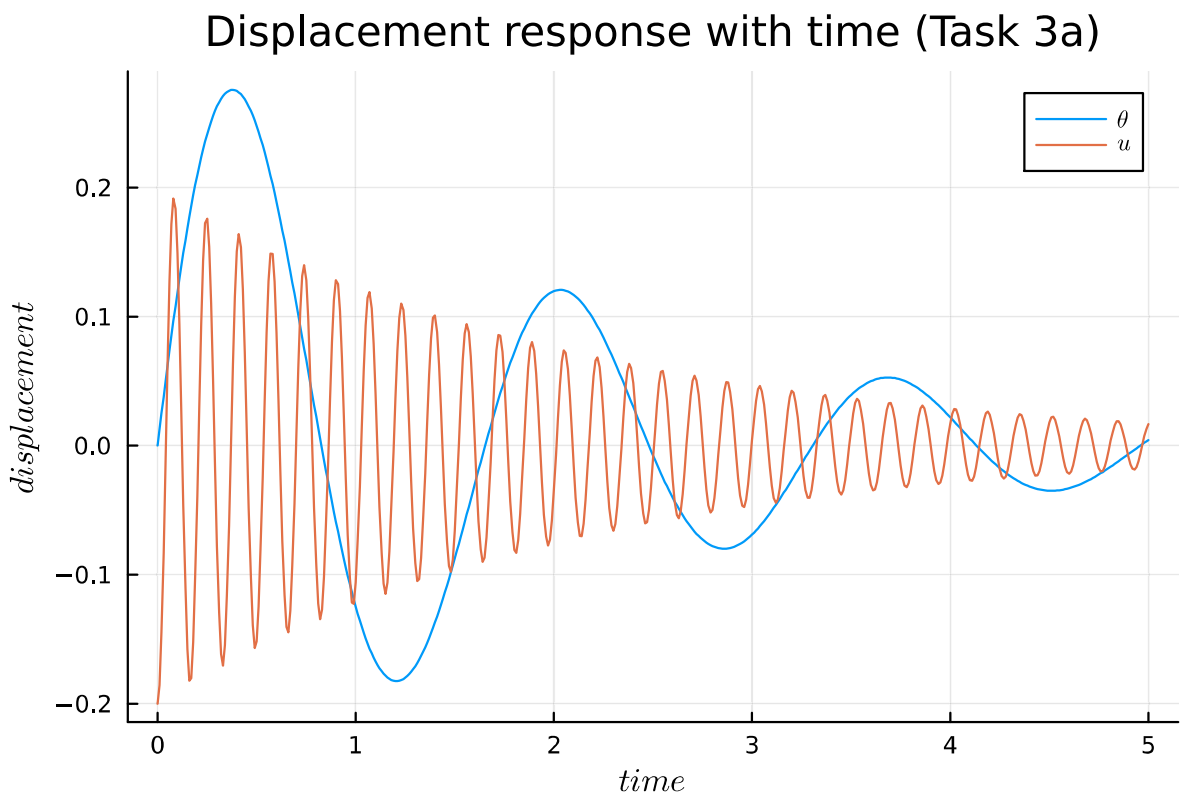
Given: $t_{final} = 5\ s, \alpha_1 = 1, \alpha_2 = 0, \rho_\infty = 1.0$

```
(
    physical_damping_a =   PhysicalDamping(
                               α₁ = 1.0
                               α₂ = 0.0
                           )
    numerical_damping_a =   NumericalDamping(
                                ρ_∞ = 1.0
                            )
)
```

```
begin # parameters specific for task 3a.
    physical_damping_a = PhysicalDamping(1.0,0.0) # (α₁ = 1.0, α₂=0.0)
    numerical_damping_a = NumericalDamping(1.0) # (ρ_∞ = 1.0)
    (;physical_damping_a,numerical_damping_a)
end
```

```
(
    response =   DynamicResponse(
                     u = 2×501 Matrix{Float64}:
                           0.0   0.012712   0.0252789   0.0376839  …  0.00219718  0.003236
                          -0.2  -0.185612  -0.144654   -0.0832765     0.00744564  0.013021
                     ud = 2×501 Matrix{Float64}:
                           1.27802  1.26437  1.24901  1.23198  …  0.105926  0.104627  0.10
                           0.0      2.8777   5.31373  6.96184     0.705139  0.637077  0.47
                     udd = 2×501 Matrix{Float64}:
                             -1.27802   -1.45124   -1.62061   -1.78593  …   -0.136926   -0
                            300.0      275.54     211.668    117.953       -11.8055    -20
                )
    errors =   Error(
                    e_abs =   [0.0, 0.000203842, 0.000532267, 0.000780959, 0.000914898,    mo

                    η =   [0.0, 0.00101921, 0.00250229, 0.0039378, 0.00433826,    more ,0.001

                    e_cum =   [0.0, 0.000203842, 0.000736109, 0.00151707, 0.00243197,    more
                )
    time =   TimeEvolution(
                   times =   [0.0, 0.01, 0.02, 0.03, 0.04,    more ,5.0]

                   steps =   [0.01, 0.01, 0.01, 0.01, 0.01,    more ,0.01]
               )
)
```

```
resp_a , err_a ,time_a=
generalized_alpha(momentum,initial_coditions,runtime,physical_damping_a,numerical_damp
ing_a)
```




```
begin
    plot(time_a.times,resp_a.u[1,:],title="Displacement response with time (Task
    3a)",xlabel=L"time",ylabel=L"displacement",label=L"$\theta$")
    plot!(time_a.times,resp_a.u[2,:],label=L"u")

end
```

Task 3b:

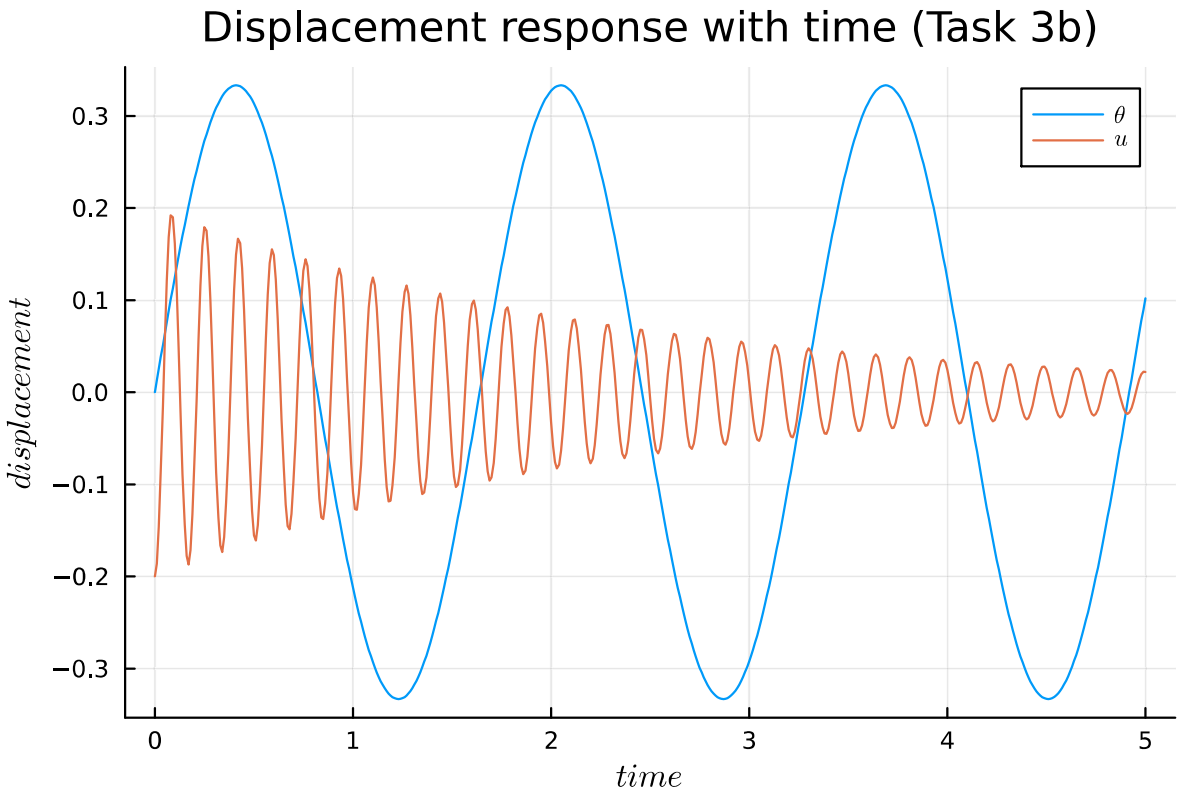
Given: $t_{final} = 5\ s, \alpha_1 = 0, \alpha_2 = 0, \rho_\infty = 0.1$

```
(
    physical_damping_b =   PhysicalDamping(
                               α₁ = 0.0
                               α₂ = 0.0
                           )
    numerical_damping_b =   NumericalDamping(
                                ρ_∞ = 0.1
                            )
)
```

```
begin # parameters specific for task 3b.
    physical_damping_b = PhysicalDamping(0.0,0.0) # (α₁ = 0.0, α₂ = 0.0)
    numerical_damping_b = NumericalDamping(0.1) # (ρ_∞ = 0.1)
    (;physical_damping_b,numerical_damping_b)
end
```

```
(response = DynamicResponse(2×501 Matrix{Float64}:
                            0.0   0.012772   0.0255218   0.0382327  …  0.0774721  0.089
```

```
1 resp_b , err_b, time_b=
  generalized_alpha(momentum,initial_coditions,runtime,physical_damping_b,numerical_damp
  ing_b)
```



Task 3 (Explanation):

Observtion:

In *Task 3a* **both** degree of freedoms (i.e. $\boldsymbol{u}$ & $\boldsymbol{\theta}$) are being damped with time, whereas in *Task 3b* **only** $\boldsymbol{u}$ is being damped with time and the amplitude of $\boldsymbol{\theta}$ remains constant through time.

Explanation:

The reason behind such observation is; $\boldsymbol{\alpha_1}$ & $\boldsymbol{\alpha_2}$ are called **physical damping parameters** because they contribute to the formulation of the *Rayleigh damping* (i.e. $\boldsymbol{C = \alpha_1 M + \alpha_2 K}$) and as given in *Task 3a* $\boldsymbol{\alpha_1 \neq 0 \rightarrow C \neq 0}$, consequently, all degree of freedoms are being damped by $\boldsymbol{C}$ regardless their frequencies.

However, $\boldsymbol{\rho_\infty}$ is called **numerical damping parameter** due to the numerical error arised from this parameter that leads to damping. Additionally, it only damps degree of freedoms that has higher frequencies and this obvious in *task 3b* which has **no physical** damping but has **numerical** damping.

Finaly, in *task 3a* there is no numerical damping because, $\boldsymbol{\gamma = \frac{1}{2} \rightarrow \rho_\infty = 1}$ (slides pg. 59) and numerical damping only occurs if $\boldsymbol{\gamma > \frac{1}{2} \rightarrow \rho_\infty < 1}$

Task 4 (Error Calculation):

> **Note**
>
> Error calculation is already implemented in `generalized_alpha`. Accordingly, in this section, I will only show the equations that were used, some notes regarding the implementation and the error graphs as well.

4.1. Zienkiewicz and Xie (i.e. absolute error):

$$\boldsymbol{e}^{ZX} = \frac{6\beta - 1}{6}(\ddot{\boldsymbol{u}}_{n+1} - \ddot{\boldsymbol{u}}_n)\Delta t^2 \rightarrow \text{(slides pg. 90)}$$

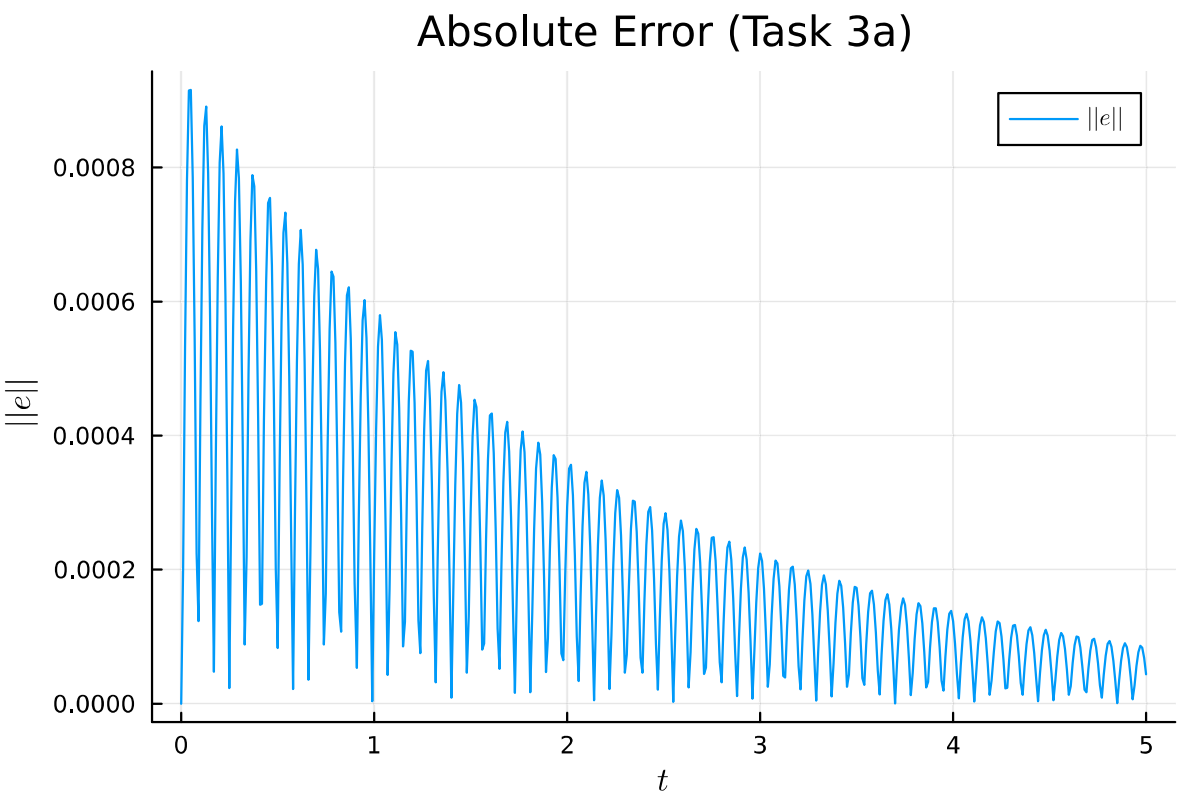
4.2. Relative error:

$$\eta = \frac{||\boldsymbol{e}||}{||\boldsymbol{u}_{n+1} - \boldsymbol{u}_n||} \rightarrow \text{(slides pg. 91)}$$

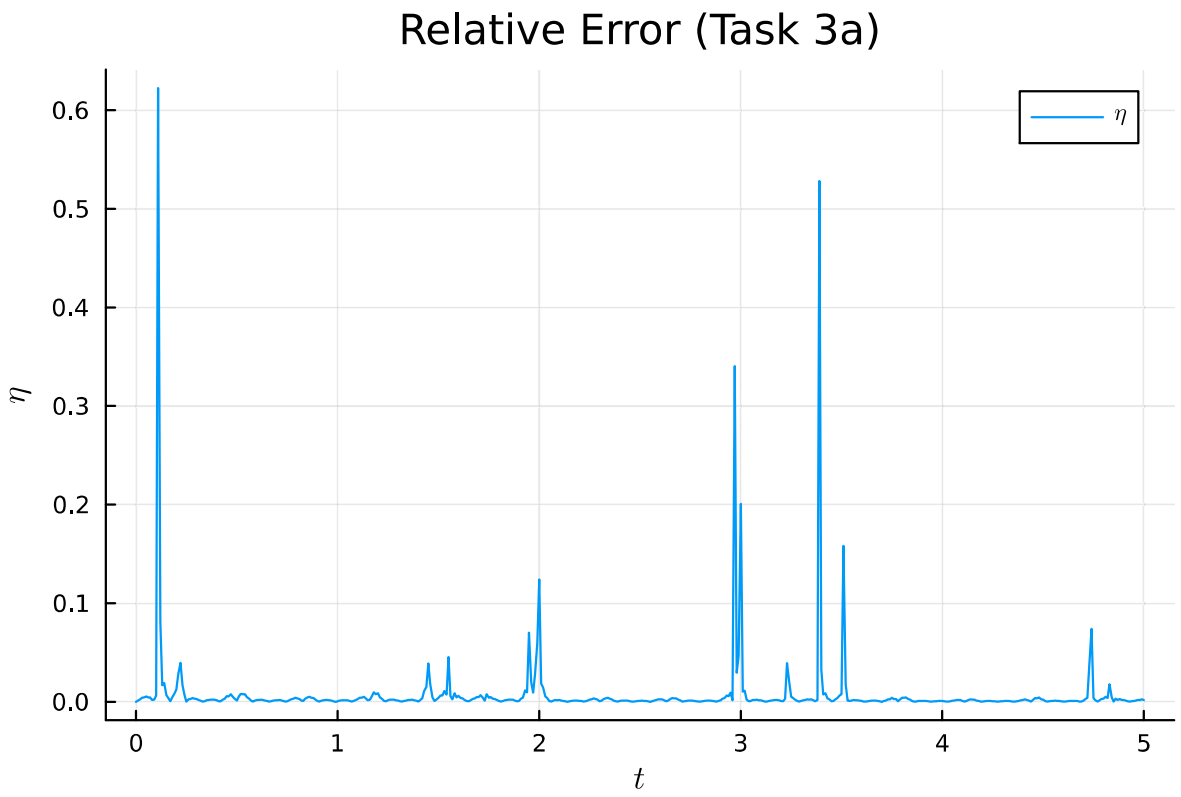
4.3. Cummulative error:

$$e_{cm} = \sum_{i=1}^{n} ||\boldsymbol{e}||$$

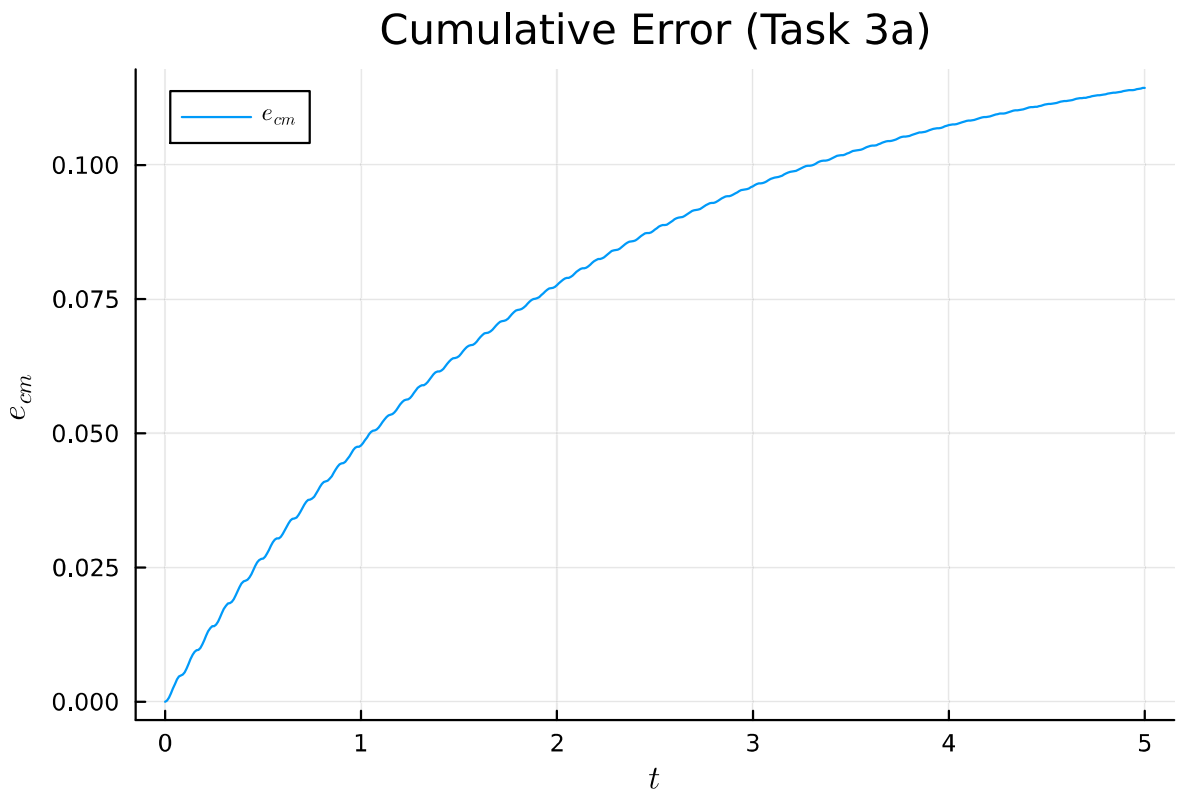
where $\boldsymbol{n}$ is the index of the current time step.



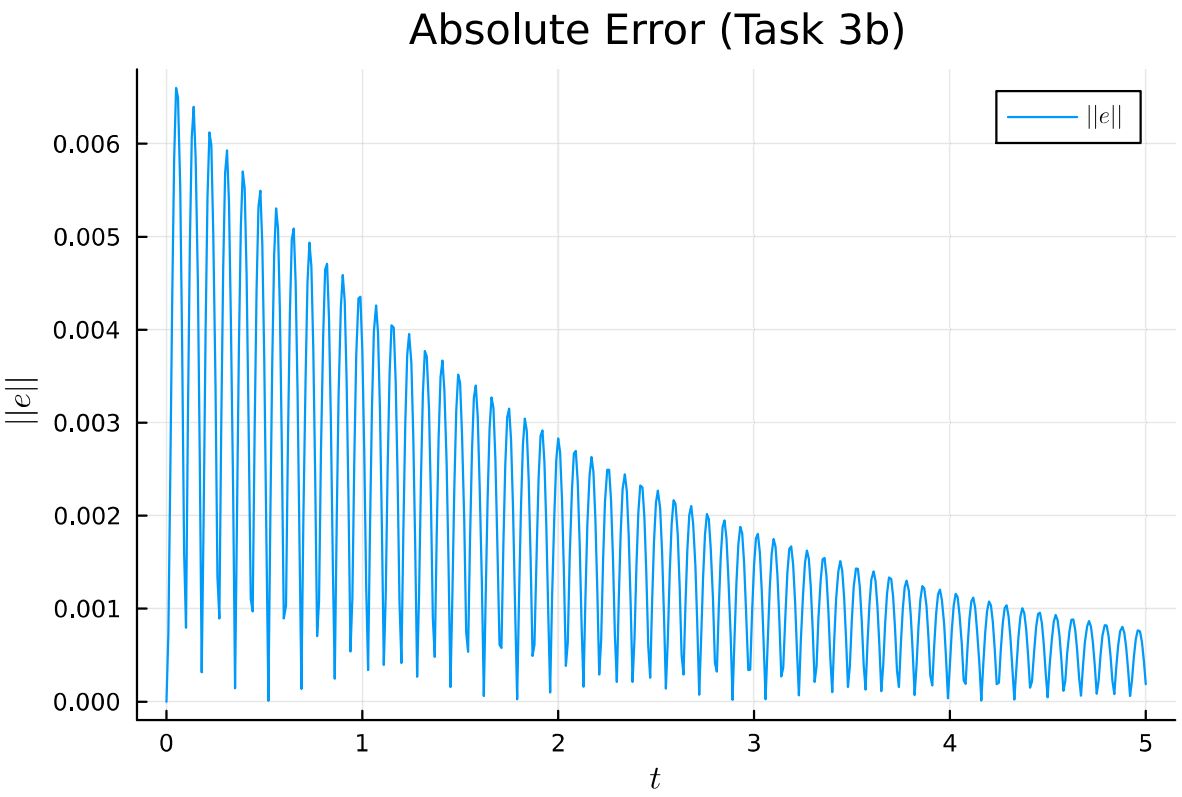

```
begin
    plot(time_a.times,err_a.e_abs,title="Absolute Error (Task
    3a)",xlabel=L"t",ylabel=L"||e||",label=L"$||e||$")
end
```



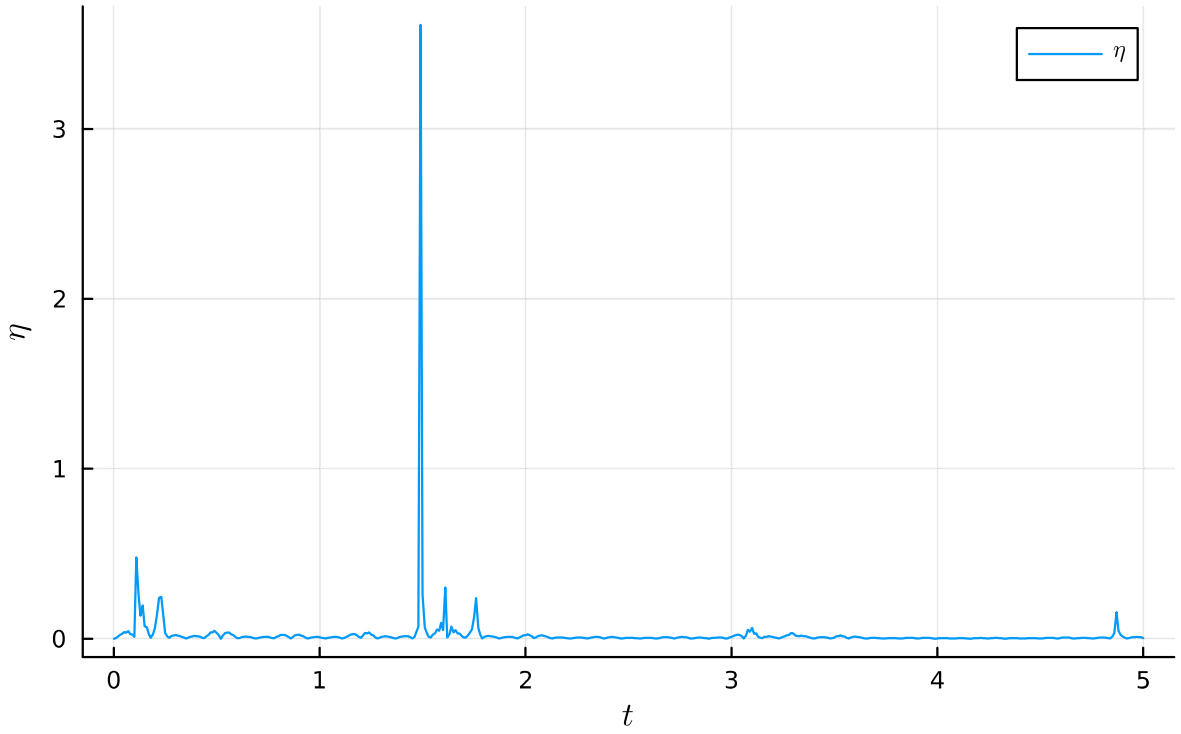

```
begin
    plot(time_a.times,err_a.η,title="Relative Error (Task
    3a)",xlabel=L"t",ylabel=L"η",label=L"$\eta$")
end
```



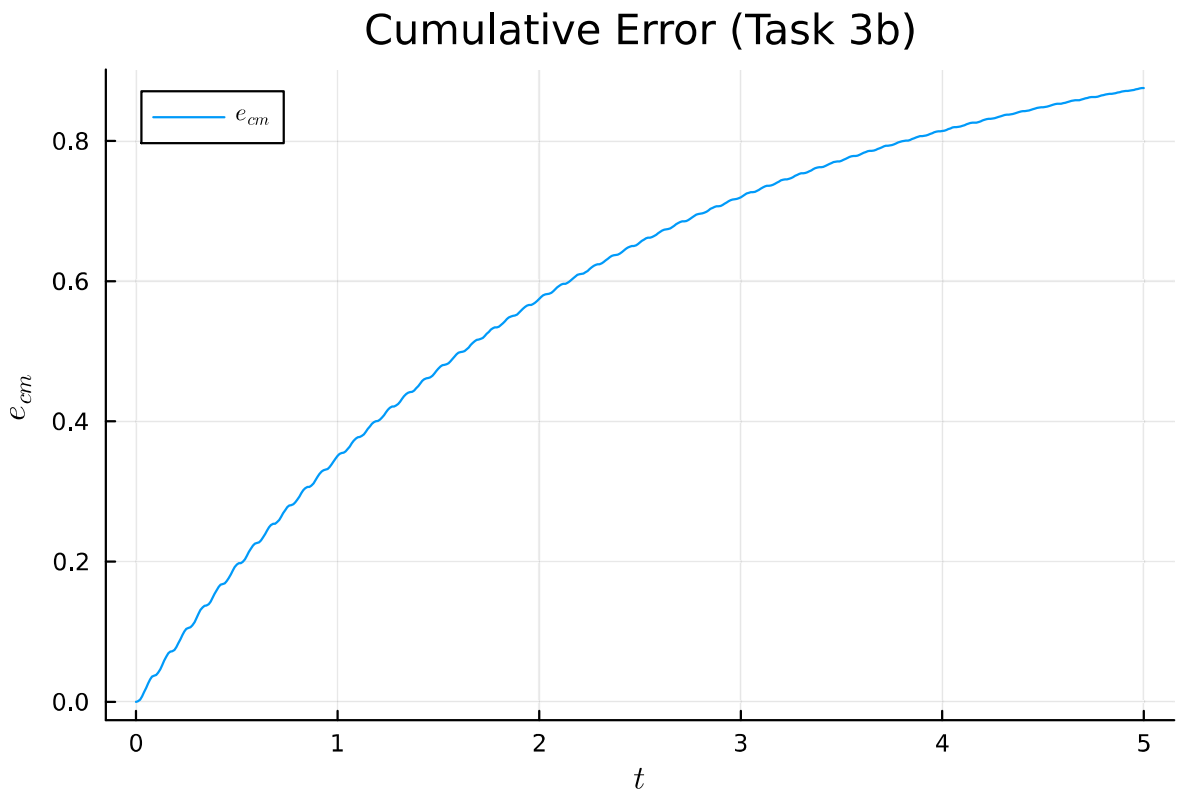

```
begin
    plot(time_a.times,err_a.e_cum,title="Cumulative Error (Task
    3a)",xlabel=L"t",ylabel=L"e_{cm}",label=L"$e_{cm}$")
end
```




```
begin
    plot(time_b.times,err_b.e_abs,title="Absolute Error (Task
    3b)",xlabel=L"t",ylabel=L"||e||",label=L"$||e||$")
end
```




```
begin
    plot(time_b.times,err_b.η,title="Relative Error (Task
    3b)",xlabel=L"t",ylabel=L"η",label=L"$\eta$")
end
```




```
begin
    plot(time_b.times,err_b.e_cum,title="Cumulative Error (Task
    3b)",xlabel=L"t",ylabel=L"e_{cm}",label=L"$e_{cm}$")
end
```

Task 5 (Adaptive time stepping algorithim):

5.1. We start by setting the current time to the initial time $t \leftarrow t_o$, and the current time step with the initil time step $\Delta t \leftarrow \Delta t_o$. 5.2. Loop as long as current time is less or equal final time $(t \leq t_f)$:
5.2.1. calculate the same objects as in the `generalized_alpha`.

> **Note**
>
> Unlike `generalized_alpha`, K_{eff} now has to be calculated inside the loop, because Δt is now changing.

5.2.2. At the end of the loop we check the relative error (η), whether it is inside the boundary $[\nu_1\eta_e, \nu_2\eta_e]$ or outside, and if it is outside, update Δt:

$$\Delta t \leftarrow \begin{cases} \Delta t\sqrt{\frac{\eta_e}{\eta}}, & \text{for } \eta \leq \nu_1\eta_e \text{ or } \eta \geq \nu_2\eta_e \\ \Delta t, & \text{otherwise} \end{cases}$$

5.2.3. Update current time (t):

$$t \leftarrow t + \Delta t$$

```
# this struct represents out time boundary that we need to stay in.
struct AdaptiveTimeBoundary
    ν₁::Float64
    ν₂::Float64
    ηₑ::Float64
end
```

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

```
function generalized_alpha_adaptive(moment::LinearMomentum,ic::InitialConditions,
time::RunTime,pd::PhysicalDamping,nd::NumericalDamping,boundary::AdaptiveTimeBoundary
)

    M = moment.M # mass matrix
    K = moment.K # stiffness matrix

    α₁ = pd.α₁
    α₂ = pd.α₂

    # calculate damping
    C = α₁*M + α₂ * K

    # Initial conditions
    u_o = ic.uₒ
    ud_o = ic.vₒ
    udd_o = inv(M) * (R - C * ud_o - K * u_o)

    # Chung and Hulbert (1993)
    ρ_∞ = nd.ρ_∞
    α_m = (2*ρ_∞ - 1)/(ρ_∞ + 1)
    α_f = (ρ_∞)/(ρ_∞+1)
    β = 0.25 * (1 - α_m + α_f)^2
    γ = 0.5 - α_m + α_f



    # create time interval range to loop over later
    ν₁ = boundary.ν₁
    ν₂ = boundary.ν₂
    ηₑ = boundary.ηₑ
    lb = ν₁ * ηₑ # lower boundary
    ub = ν₂ * ηₑ # upper boundary


    # define our respone containers
    u = zeros(2,0)
    ud = zeros(2,0)
    udd = zeros(2,0)

    # time data
    tₒ = 0.0 # initial time
    t_current = tₒ # current time
    tf = time.tf # final time we want to solve for
    Δtₒ = time.Δt # initial time step
    t = [tₒ] # container for current time values
    tstep = [Δtₒ] # container for the evolution of time steps
    Δt = Δtₒ # current time step

    # set initial conditions in our response containers
    u = hcat(u,u_o)
    ud =hcat( ud,ud_o)
    udd = hcat(udd,udd_o)
```

```
52
53      # define the containers for errors
54      e_abs = [0.0] # absolute error
55      η = [0.0] # relative error
56      e_cum = [0.0] # cummulative error
57      while t_current <= tf
58
59          K_eff = M * ((1-α_m)/(β*Δt^2)) + C * (γ * (1 - α_f))/(β*Δt) + K * (1 - α_f)
60
61          uₙ = u[:,end] # current displacement
62          udₙ = ud[:,end] # current velocity
63          uddₙ = udd[:,end] # current acceleration
64
65          r_eff = -K * α_f * uₙ +
66                  C * (((γ * (1-α_f))/(β*Δt)) * uₙ + ((γ - γ*α_f - β)/(β)) * udₙ +
                    (((γ-2*β)*(1-α_f))/(2*β)) * Δt*uddₙ) +
67                  M * (((1-α_m)/(β*Δt^2))*uₙ + ((1-α_m)/(β*Δt))* udₙ + ((1-α_m - 2
                    *β)/(2*β)) * uddₙ)
68
69          # solve for u.
70          uₙ₁ = K_eff \ r_eff # u_{n+1} next displcement
71          u = hcat(u, uₙ₁)
72
73          # update velocity and acceleration
74          udₙ₁ = ((γ)/(β*Δt)) * (uₙ₁ - uₙ) - ((γ - β)/(β))*udₙ - ((γ - 2*β)/(2*β)) *
            Δt * uddₙ
75          ud = hcat(ud,udₙ₁)
76
77          uddₙ₁ = (1/(β*Δt^2))*(uₙ₁ - uₙ) - (1/(β*Δt)) * udₙ - ((1-2*β)/(2*β)) * uddₙ
78          udd = hcat(udd,uddₙ₁)
79
80          # calculate errors
81              e_absₙ₁ = norm(((6*β - 1)/(6))*(uddₙ₁ - uddₙ) * Δt^2)
82              push!(e_abs,e_absₙ₁)
83          ηₙ₁ = e_absₙ₁ / norm(uₙ₁ - uₙ)
84          push!(η,ηₙ₁)
85          push!(e_cum, sum(e_abs))
86
87          # adapting the time step
88          if ηₙ₁  > ub || ηₙ₁ < lb
89              # here means Δt is either to big ()
90              Δt = Δt * sqrt(ηₑ/ηₙ₁)
91          end
92          t_current += Δt
93          push!(t,t_current)
94          push!(tstep,Δt)
95
96      end
97
98      (response =  DynamicResponse(u,ud,udd), error =  Error(e_abs,η,e_cum),time =
        TimeEvolution(t,tstep))
99  end
```

Task 6:

Given $\alpha_1 = 1$, $\alpha_2 = 0$, and $\rho_\infty = 1$, It's required to:

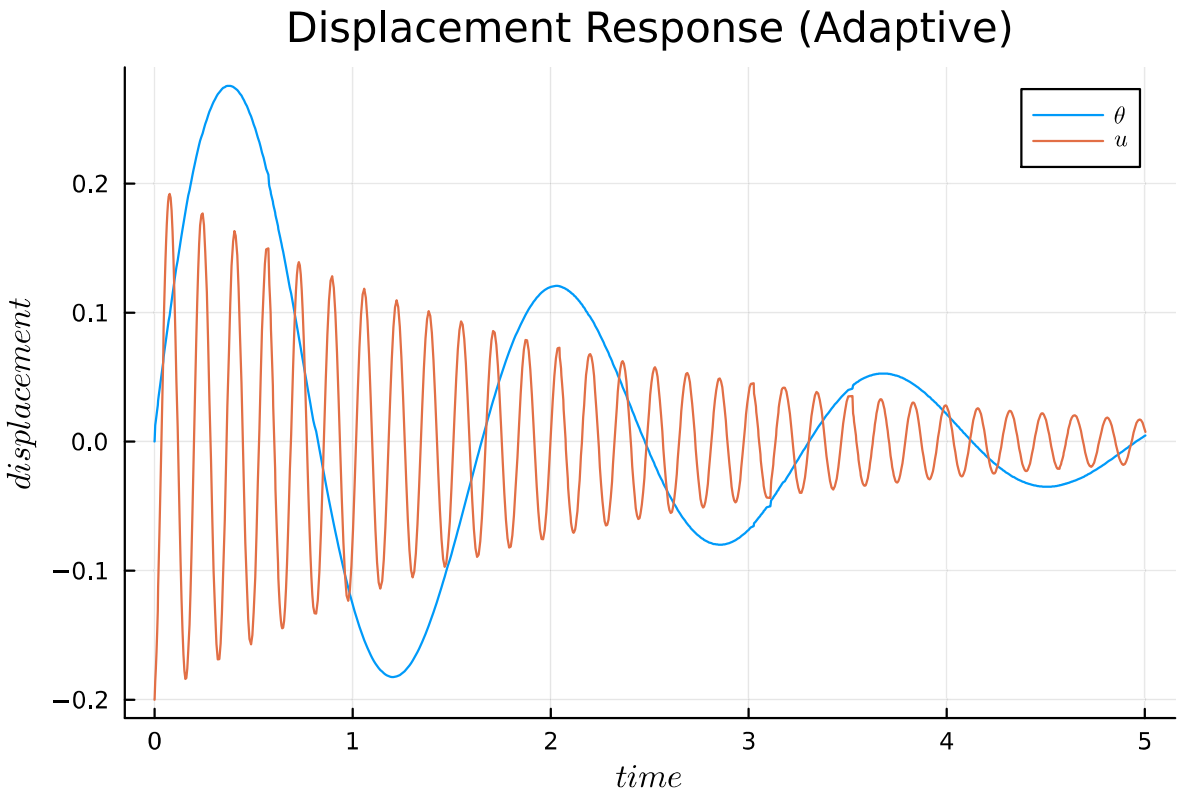
- Solve the system for 5.0 s.
- Plot the dynamic response, error, and the evolution of time step.
- Compare the results with those from task 3a.
- Cumulative error for both cases after 5.0 s and the number of time steps.

```
time_bound =   AdaptiveTimeBoundary(
                   ν₁ = 1.0
                   ν₂ = 10.0
                   ηₑ = 0.001
               )
```

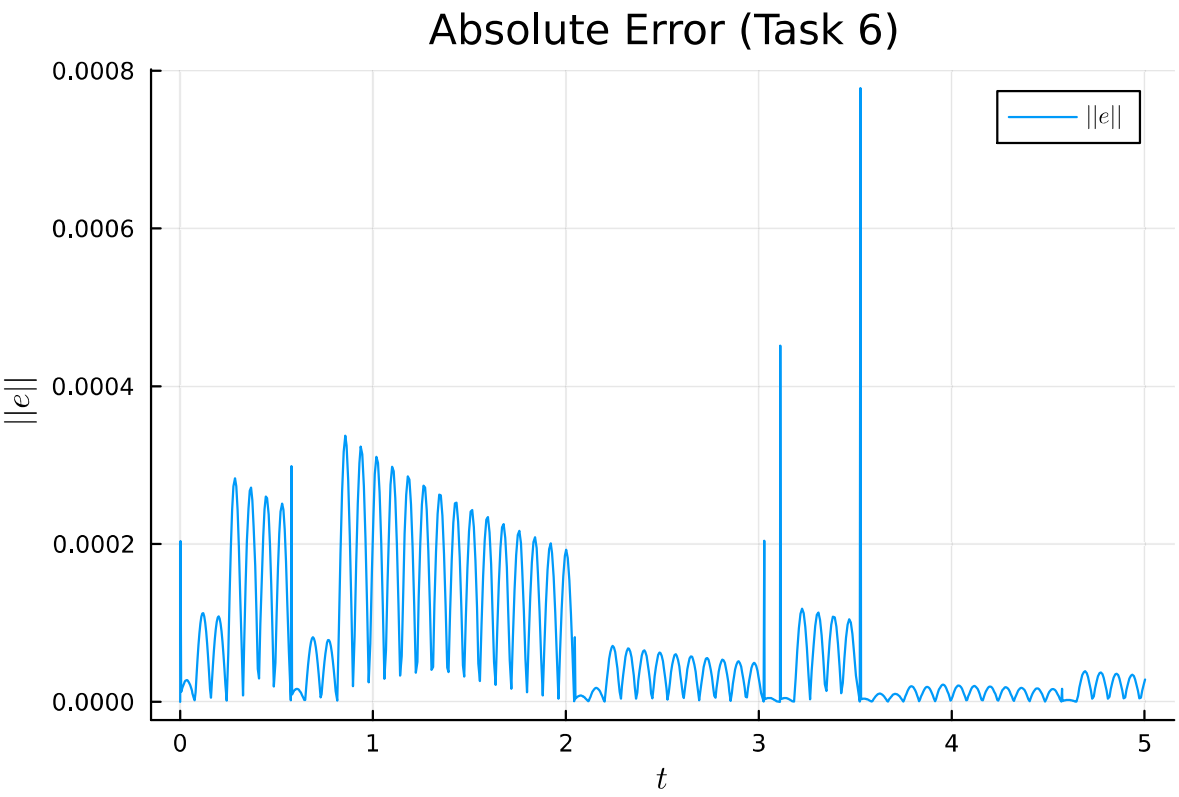
```
time_bound = AdaptiveTimeBoundary(1.0,10.0,1e-3)
```

```
(
    response =   DynamicResponse(
                     u = 2×871 Matrix{Float64}:
                           0.0   0.012712   0.0165854   0.0204446  …  0.00336907  0.004126
                          -0.2  -0.185612  -0.17552    -0.162992      0.0147341   0.011607
                     ud = 2×871 Matrix{Float64}:
                           1.27802  1.26437  1.25984  1.25514  1.25029  …   0.102958   0.1
                           0.0      2.8777   3.69884  4.46549  5.16702     -0.350848  -0.4
                     udd = 2×871 Matrix{Float64}:
                             -1.27802   -1.45124   -1.50364   -1.55568  …   -0.152483   -0
                            300.0      275.54     259.581    240.022       -21.7502    -16
                 )
    error =   Error(
                 e_abs =   [0.0, 0.000203842, 1.25261e-5, 1.53519e-5, 1.79533e-5,    more ,

                 η =   [0.0, 0.010617, 0.0011588, 0.00117109, 0.0011755,    more ,0.0068476

                 e_cum =   [0.0, 0.000203842, 0.000216368, 0.00023172, 0.000249673,    more
             )
    time =   TimeEvolution(
                 times =   [0.0, 0.00306901, 0.00613802, 0.00920704, 0.012276,    more ,5.00

                 steps =   [0.01, 0.00306901, 0.00306901, 0.00306901, 0.00306901,    more ,0
             )
)
```

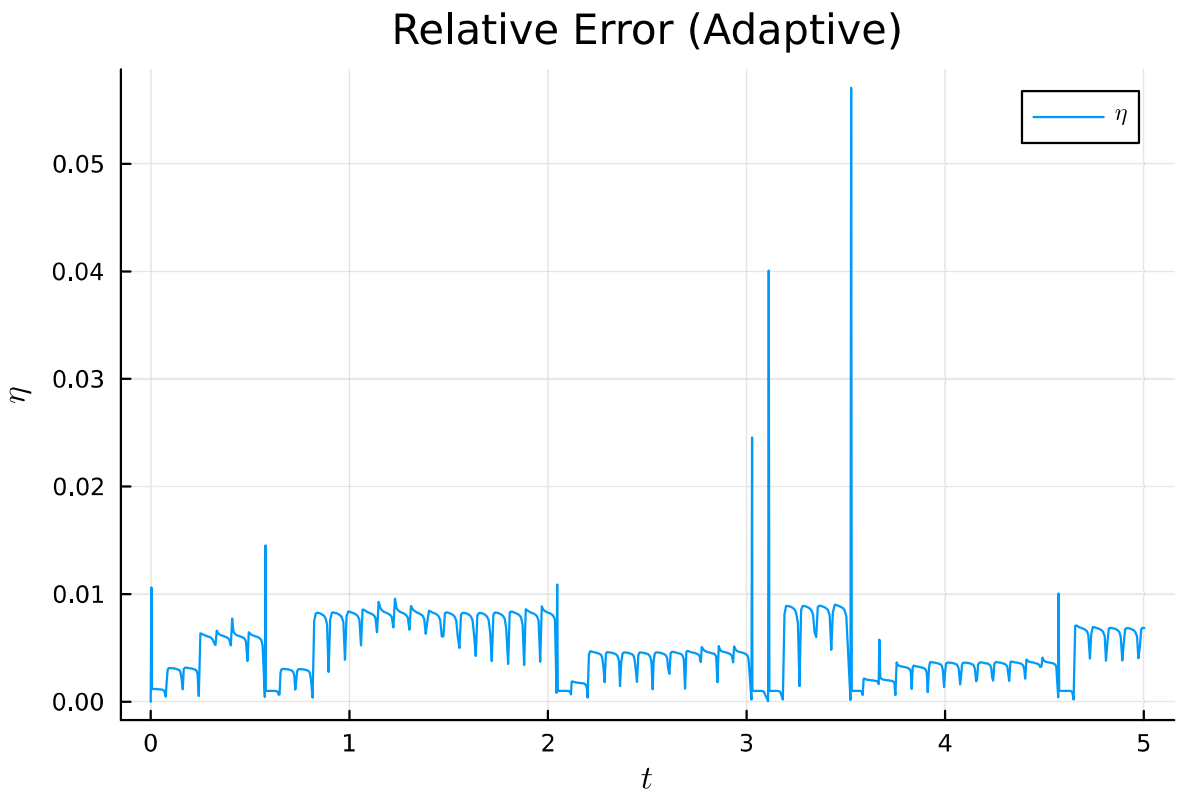
```
ad_resp , ad_err,ad_time = generalized_alpha_adaptive(momentum,initial_coditions,runtime,physical_damping_a,numerical_damping_a,time_bound)
```



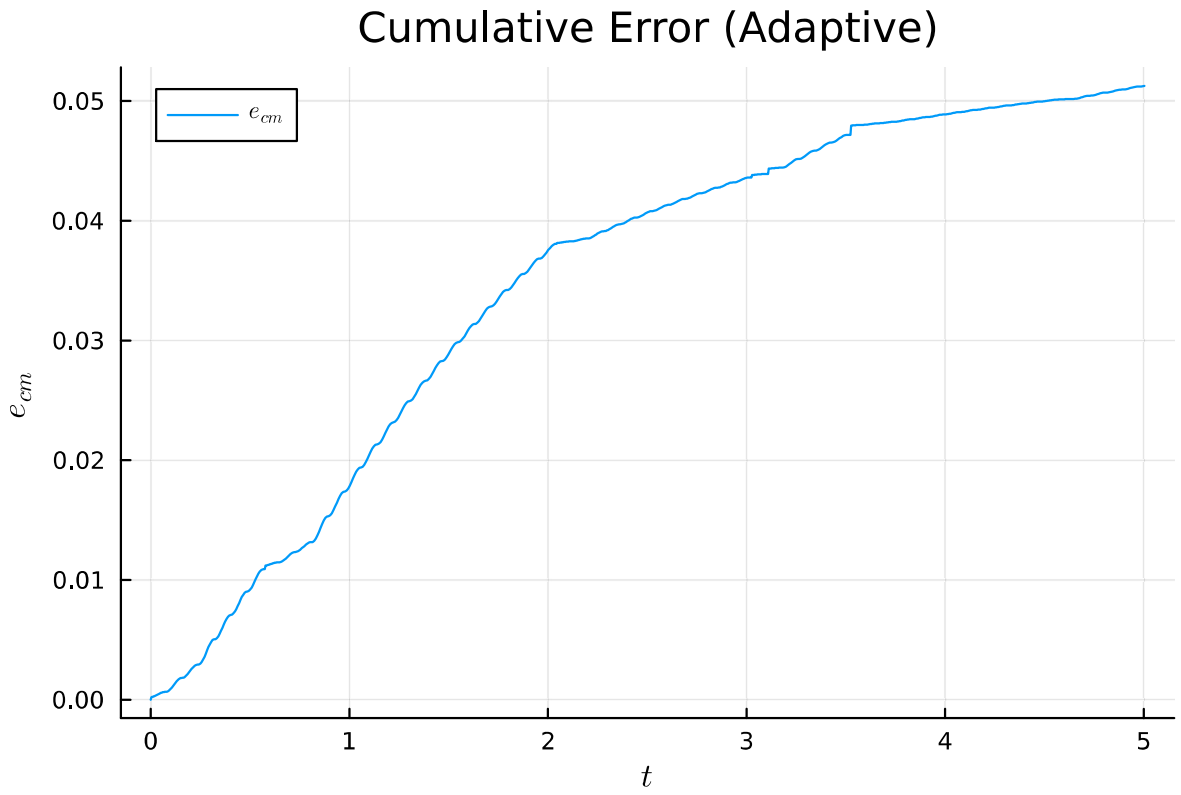

```
begin
    plot(ad_time.times,ad_resp.u[1,:],title="Displacement Response
    (Adaptive)",xlabel=L"time",ylabel=L"displacement",label=L"$\theta$")
    plot!(ad_time.times,ad_resp.u[2,:],label=L"u")

end
```



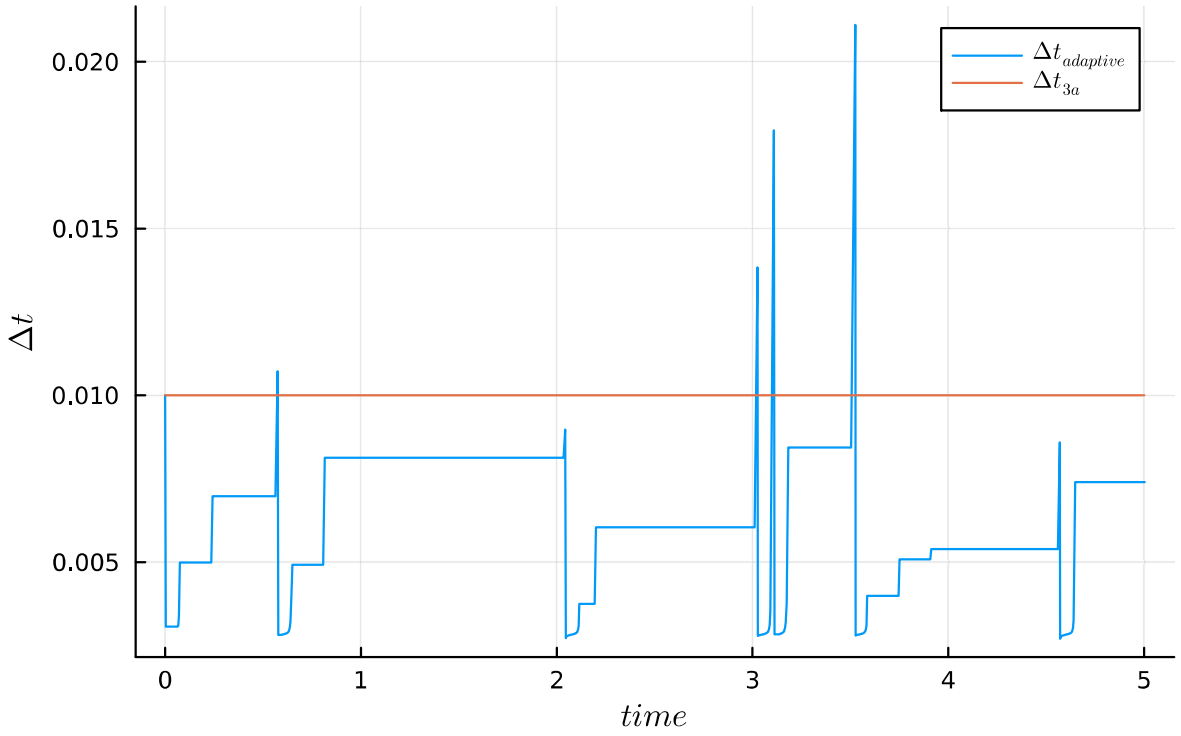

```
begin
    plot(ad_time.times,ad_err.e_abs,title="Absolute Error (Task
    6)",xlabel=L"t",ylabel=L"||e||",label=L"$||e||$")
end
```



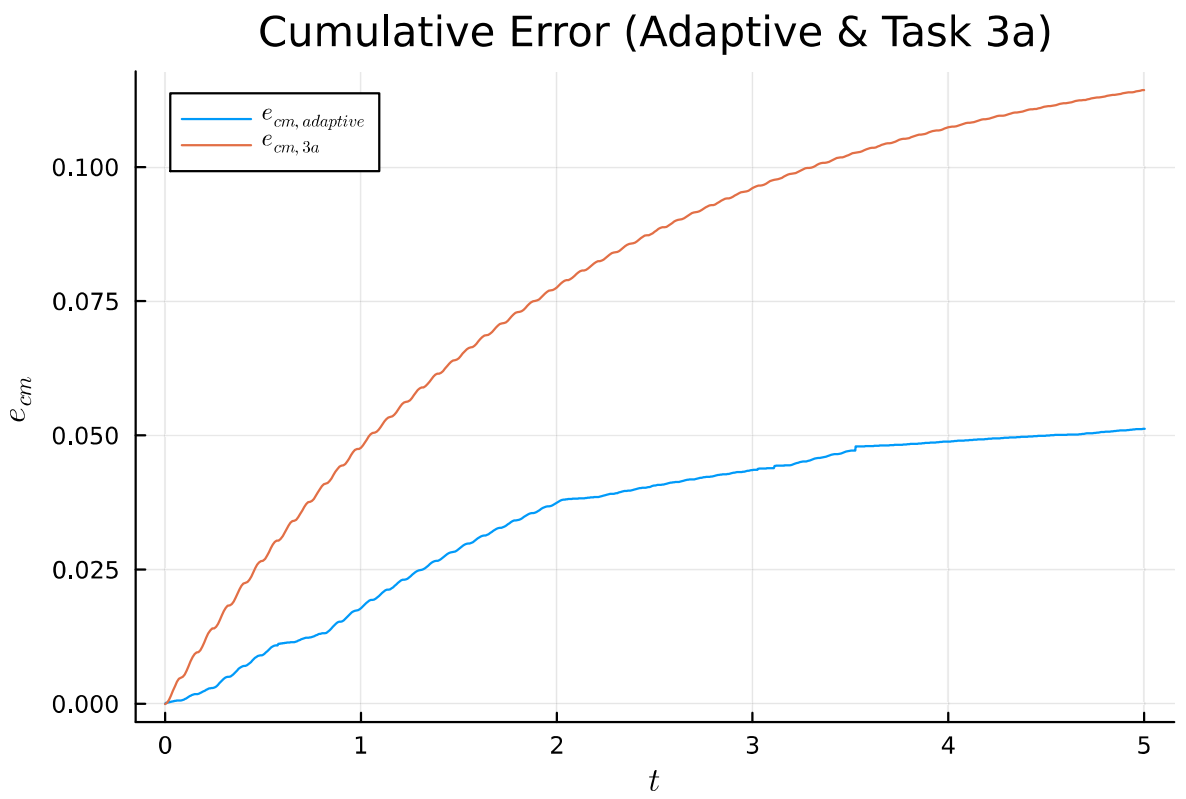

```
begin
    plot(ad_time.times,ad_err.η,title="Relative Error
    (Adaptive)",xlabel=L"t",ylabel=L"η",label=L"$\eta$")
end
```




```
begin
    plot(ad_time.times,ad_err.e_cum,title="Cumulative Error
    (Adaptive)",xlabel=L"t",ylabel=L"e_{cm}",label=L"$e_{cm}$")
end
```




```
begin
    plot(ad_time.times,ad_time.steps,title="Time step evolution
    (adaptive)",xlabel=L"time",ylabel=L"Δt",label=L"$Δt_{adaptive}$")
    plot!(time_a.times,time_a.steps,label=L"Δt_{3a}")
end
```




```
begin
    plot(ad_time.times,ad_err.e_cum,title="Cumulative Error (Adaptive & Task
    3a)",xlabel=L"t",ylabel=L"e_{cm}",label=L"$e_{cm, adaptive}$")
    plot!(time_a.times,err_a.e_cum,label=L"$e_{cm, 3a}$")
end
```

Cumulative error after 5 s:

```
(adapt = 0.0512708, t3a = 0.114384)
```

```
(adapt = ad_err.e_cum[end],t3a =err_a.e_cum[end] )
```

> **Note**
>
> From the previous result, we can observe that the cumulative error in the adaptive method is less than the fixed one and this was anticipated.

Number of steps:

```
(adapt = 871, t3a = 501)
```

```
(adapt = length(ad_time.steps),t3a = length(time_a.steps) )
```