Homework series 2 for Data Assimilation course (WI4475 2025)

This is the second homework series for the data assimilation course (WI4475 2025). There are 2 questions with 5 subquestions each. You can earn one point for each of these 10 subquestions, which directly sums up the grade for this homework series. You can select your programming language and editor of choice, but we can only process PDF files for grading (unfortunately). However, the interactive elements in this document will only work with Pluto.jl and it's probably easiest to add the answers after each question and generate a pdf from there using the on the top right of the screen.

The files for this homework series are all available here.

Julia and Pluto

This homework was created in Pluto. Pluto.jl is a notebook system where one can combine text and code into one document that contains a list of cells. Each cell can contain code or text. Unlike the commonly used jupyter the order of execution of the cells is inferred automatically, and when you change a cell, then all dependent cells will be run again. The consequence is that at any time, the results that you see are the output of the current inputs, which is not the case in Jupyter. For installation go to the install at Pluto.jl site

The Julia language is high-level like Python, yet fast like C or Fortran. Here's a tiny example:

```
ratio = 2*π #Greek symbols are allowed and are typed as \pi<TAB>
a=randn(2,100) #some random Gaussian points
scatter(a[1,:],a[2,:]) #make a scatter plot
```

With 'ctrl-m' you turn a cell into a text cell. The text is written in markdown format

In the Pluto notebook, the code can be hidden and shown again by clicking on the eye symbol:

Just try to show the markdown code for this block of text now.

The first question of this homework will focus on the concept of adjoint equations and how to derive them manually in continuous and discrete forms. The second question will then show how this can be automated in various ways in modern programming languages.

```julia
# Packages used in this notebook. Packages will be downloaded automatically which can take a bit of time when you first start the notebook
using Plots, ForwardDiff, DifferentialEquations, ReverseDiff, Interpolations, Statistics
```

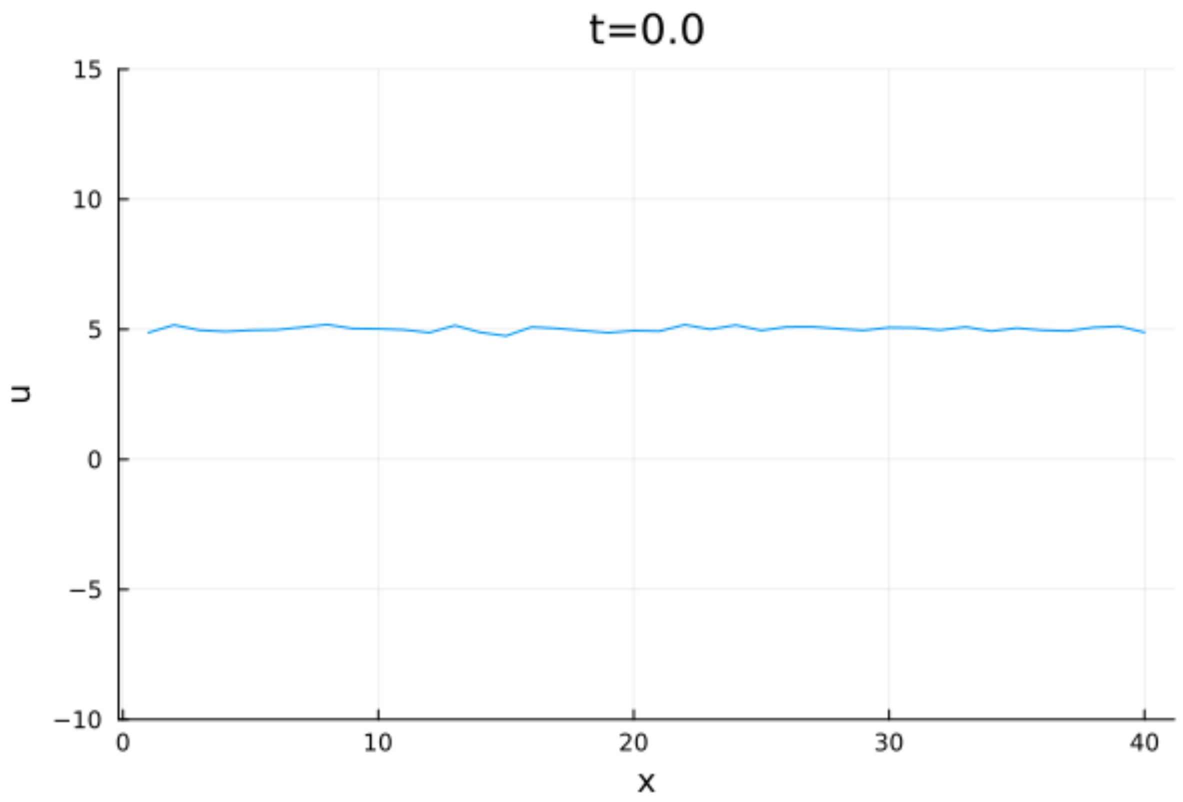
Question 1: An adjoint for the Lorenz 95 equation

The Lorenz equation is an Ordinary Differential Equation (ODE) that describes a weather variable around our planet at a fixed latitude. It is a very simplified model, yet it shows some interesting chaotic behavior. The Lorenz 95 system is given by:

$$\frac{du_m}{dt} = -u_{m-2}u_{m-1} + u_{m-1}u_{m+1} - u_m + F$$

on a periodic domain $m = 1, \ldots, M$, so that $u_0 = u_M$ and $u_{M+1} = u_1$. The initial condition is $u_m(0) = c_m$ and time runs from 0 to T.

Just open up the code below, if you're interested in the numerical integration for the Lorenz equation.



```
Saved animation to /tmp/jl_Ehyze36di5.gif
```

1a: Model linearization

Linearize the model around a reference trajectory $u_m(t)$, using an infinitesimal perturbation δc_m of c_m, which leads to a change $\delta u_m(t)$ of $u_m(t)$. Note that the linearization depends on the reference trajectory.

Answer ...

```
md"""
Answer ...
"""
```

1b: Lagrange multipliers

Assume that we observe all state variables at time T with some noise, i.e. $Z_m(T) = u_m(T) + v_m$, where v_m are independent with variance σ^2.

Write down the cost-function J for this case and add the linearized model equations as constraints using the functions $\lambda_m(t)$ as Lagrange multipliers.

Answer ...

```
md"""
Answer ...
"""
```

1c: Adjoint solution

Continuing on the previous result, find an expression for the adjoint-model and the gradient of J with respect to the initial condition c_m $m \in 1, \ldots, M$. First, develop an expression for δJ

and apply integration by parts.

Answer ...

```
md"""
Answer ...
"""
```

1d: Discrete-time system

Next, we apply a different route. First, write the equations in the standard system form for discrete time:

$$\mathbf{x}_{k+1} = \mathbf{M}(\mathbf{x}_k)$$

For this, you can apply the Euler forward method for discretizing the timesteps. Although this method works poorly for this system, it will simplify the resulting equations.

Assume that all state variables are observed at discrete times $k = 1, \ldots, N$. Shape this into the form:

$$\mathbf{z}(k) = \mathbf{H}\mathbf{x}(k) + \mathbf{v}(k)$$

where $\mathbf{v}(k)\ N(0, \sigma^2\mathbf{I})$. In summary, write the Lorenz system in standard (discrete time) form, and derive the gradient of the cost-function with the adjoint formalism in this context.

Answer ...

```
md"""
Answer ...
"""
```

1e: Parameter estimation$

Finally, we also want to estimate the parameter F, so we extend the model to:

$$\mathbf{x}_{k+1} = \mathbf{M}(\mathbf{x}_k, F)$$

On way of solving this, is by extending the state-vector $\mathbf{x}_k$ with the additional variable F. Show how this can be used to reshape the problem into a form where we estimate again the initial condition, albeit of an augmented system.

Answer ...

```
md"""
Answer ...
"""
```

Question 2: Wind-driven profile

In this question, we'll consider a bit more complicated model, but now the aim is to compute the derivatives with an algorithm instead of manually.

Consider the following momentum equation for a water column profile for $z \in [-H, 0]$

$$\frac{\partial u}{\partial t} = \frac{\partial \tau}{\partial z}$$

with $\tau = \nu \frac{\partial u}{\partial z}$

The boundary condition at the seabed is:

$$\tau(z=0) = \tau_b/\rho = -c_b u(z=b)|u(z=b)|$$

and at the surface a wind stress τ_s is forcing the top of the water column.

$$\tau(z=H) = \tau_s/\rho$$

Note that we're abusing the notation for τ since the $\tau(z)$ profile represents $\tau(z)/\rho$.

As a greatly simplified turbulence closure, we use:

$$\nu = \nu_{max} 4(z/H)(1 - z/H)$$

In this model, the wind stress at the surface drives the current, which is spread through the water columns by the eddy diffusivity. Finally, the bottom stress eventually balances the wind stress, resulting in an equilibrium. For practical purposes, we assume that the equilibrium is reached at $t = 5 hours$.

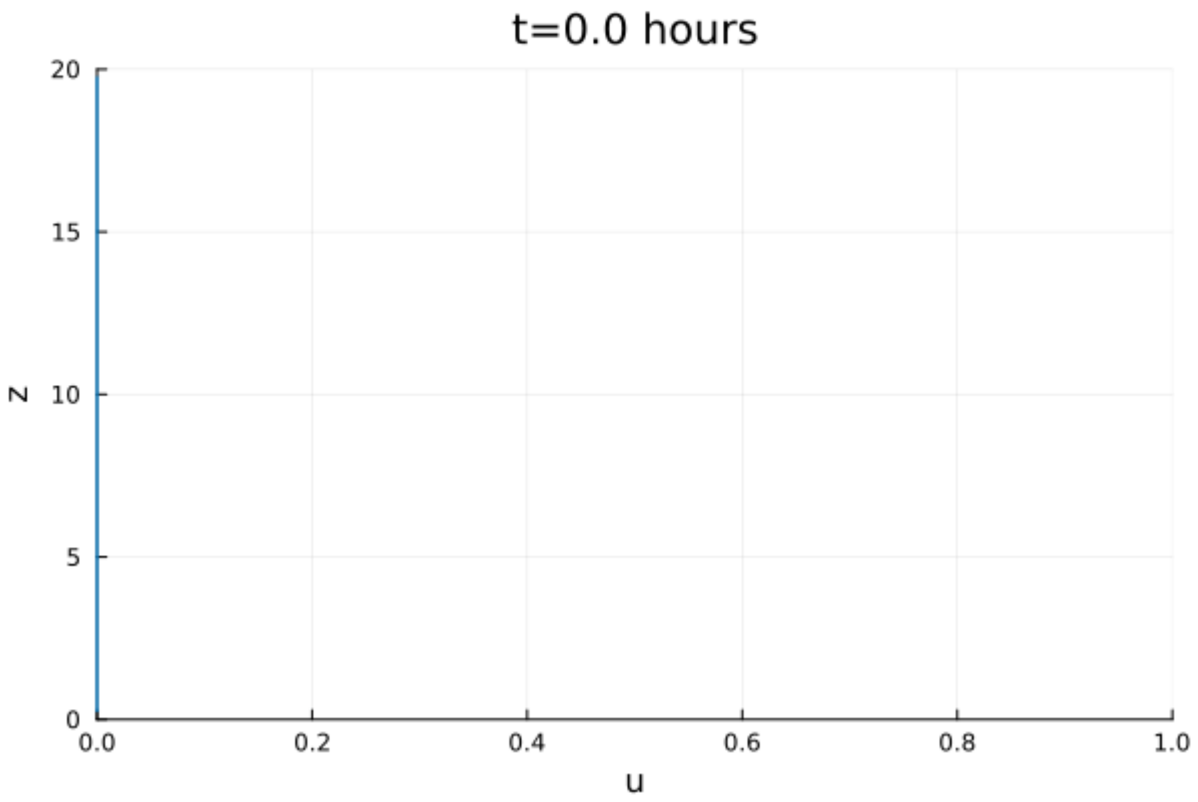
For computing a numerical solution we discretize the equation in z.

Using $u_k \approx u(z = (k - 1/2)\Delta z)$ and $\tau_k \approx \tau(z = (k-1)\Delta z)$, we can approximate the momentum as:

$$\frac{\partial u_k}{\partial t} = \frac{\tau_{k+1} - \tau_k}{\Delta z}$$

with $\tau_k = \nu \frac{u_k - u_{k-1}}{\Delta z}$

Next we collect all u_k into a vector $\mathbf{u}$, so we can use an ODE-solver. The example below gives all the details.


t=0.0 hours
z
u
0
5
10
15
20
0.0
0.2
0.4
0.6
0.8
1.0

```julia
begin
    # compute z grid at u cell centers
    function z_u(H,n)
        Δz=H/n
        z=collect(((1:n).-0.5)*Δz) # z values for u (midpoints of the cells)
        return z
    end

    #plotting and animation for velocity profile equation
    function plot_profile(u, t, p)
        H=20.0
        n=length(u)
        z=z_u(H,n)
        plot(u, z, title="t=$(round(t/3600,digits=2)) hours",label=false, ylims=(0,H),xlims=(0.0,1.0))
        xlabel!("u")
        ylabel!("z")
    end

    function animate_profile(sol, p, timestep=200.0, fps=15)
        # create an animation
        times = sol.t[1]:timestep:sol.t[end] # times at which to plot the solution
        nt = length(times)
        anim = @animate for i in 1:nt
            u=sol(times[i])
            plot_profile(u,times[i],p)
        end
        return gif(anim, fps = fps)
    end


function profile(du_dt, u, p, t)
    κ=0.4       # von Karman number
    H=20.0      # depth of the fluid
    c_b=0.01    # bottom friction coefficient
    ρ=1000.0    # density of the fluid
    ν_max=0.2   #maximum eddy viscosity
    τ_s_values=p #wind stress is a formal parameter of the model
    tspan=(0.0,5*3600.0) #time span of wind stress function
    n=length(u)
    Δz=H/n
    τ_b=c_b*u[1]^2 #bottom stress from velocity near the bottom
    # make function for interpolation of wind stress
    nt=length(τ_s_values)
    Δt=(tspan[2]-tspan[1])/(nt-1)
    t_values=collect(0.0:Δt:(5*3600.0))
    τ_s_function=linear_interpolation(t_values, τ_s_values, extrapolation_bc=Line())

    ν=similar(u,n) #eddy viscosity (array with elements of same type as u)
    for k in 1:n
        z=(k-1)*Δz
        ν[k]=ν_max*4.0*κ*z/H*(1-z/H)
    end
    τ=similar(u,n+1) #stress profile
    τ[1]=τ_b #bottom stress
    τ[n+1]=τ_s_function(t)/ρ #surface stress - ! time dependent
    for k in 2:n
        τ[k]=ν[k]*(u[k]-u[k-1])/Δz #stress from velocity gradient
    end
    for k in 1:n
        du_dt[k]=(τ[k+1]-τ[k])/Δz # momentum balance
    end
end


    τ_s=1.0     # surface stress
    p_profile=[τ_s, τ_s] # parameter vector has value at beginning and end
    n_profile=40 # number of grid cells in profile
    u₀_profile = zeros(n_profile) # initial condition
    tspan_profile=(0.0,5*3600.0)

    prob_profile=ODEProblem(profile,u₀_profile,tspan_profile,p_profile)
    sol_profile=solve(prob_profile)
    animate_profile(sol_profile,p_profile)
end
```

```
Saved animation to /tmp/jl_Ovyx2TgLG7.gif
```

2a: Discrete derivative

A simple way to approximate derivatives is with finite differences. For example, the forward difference gives:

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

This can also be applied to more complex computations, such as the function that maps the constant wind stress to the velocity profile at the final time.

In Julia, it is often convenient to wrap your entire computation in a function with the inputs and outputs that you want/need.

```
0.515271952209384
```

```julia
begin
# Wrap model in a function mapping stress to velocity at time T=5 hours
function stress_to_u5(τ_s)
    p=[τ_s, τ_s] #parameters, here wind-stress at begin and end of tspan
    n=40 # number of grid cells
    u₀ = zeros(n) # initial velocities
    tspan=(0.0,5*3600.0)

    prob=ODEProblem(profile,u₀,tspan,p)
    sol=solve(prob)
    return sol(5*3600.0)
end

@show u5=stress_to_u5(1.0) # evaluate at tau=1.0 N/m^2

@show mean(u5)

end
```

```
u5 = stress_to_u5(1.0) = [0.2871123662964366, 0.3400802850272524, 0.3673
4963667818893, 0.3860892194697356, 0.4005914603872569, 0.4125743737422755,
0.42289822737851396, 0.4320559421632511, 0.4403569778709574, 0.448009068069
92735, 0.4551591053619514, 0.4619154663348682, 0.46836106335614075, 0.47456
14103242268, 0.48056984726505064, 0.48643106955393633, 0.4921836086532145,
0.497861647978122, 0.5034964123461594, 0.509117286875998, 0.514752773683301
9, 0.5204313681168607, 0.5261824234026002, 0.5320370700663484, 0.5380292637
081213, 0.5441970531868923, 0.5505841955502898, 0.5572423031747465, 0.56423
38098778354, 0.5716362197943698, 0.5795484231042993, 0.5881004688922333, 0.
5974693988881562, 0.6079063469375003, 0.6197861790092273, 0.633706731115841
4, 0.6507120507739613, 0.672887658604553, 0.7054439020755495, 0.76921597327
92111]
mean(u5) = 0.515271952209384
```

Use finite differences to compute $\partial\mathbf{u}/\partial\tau_s$. Determine a good step size for $\Delta\tau_s$. Why can it be too small or too large? You can also use central differences. What is an advantage of central differences?

Answer ...

```julia
md"""
Answer ...
"""
```

```julia
begin
    # your code
end
```

2b: Forward mode auto differentiation

A modern alternative for finite differences in this context is forward mode differentiation. Note that this method uses the rules of calculus for differtiation, so it doesn't rely on a finite difference. FowardDiff is often quite easy to apply.

Adapt the example below, so it computes the derivative $\partial\mathbf{u}/\partial\tau_s$ for $\tau_s = 1.0\ N/m^2$. Also compare the result to your answer in 2a.

```
[1.0, 0.866025, -0.5]
```

```
begin
    f_test(x)=[x,sin(x),cos(x)]
    df_dx=ForwardDiff.derivative(f_test,π/6) # second argument is point for
    derivative
end
```

Answer ...

```
md"""
Answer ...
"""
```

2c: Forward mode with dual numbers

A common method for the implementation of forward-diff uses dual numbers. It is useful to think of a complex computation as a sequence of elementary steps, like addition and multiplication. We know the calculus rules, for each step, so we can implement them each. Object-oriented programming can then be used to insert dual numbers for all variables that depend on the input variable. In a modern language like Julia or Python, we can then recompile the computation for our extended type.

Let's start with an example. Consider $x + \epsilon 1$ and apply the function $h(x) = x^2$. This gives:

$$h(x+\epsilon) = x^2 + 2x\epsilon + \epsilon^2$$

We recognize the original function value, the derivative at the ϵ term, and a higher-order of ϵ. We can think of this as mapping the pair (a, b) with interpretation $a + b\epsilon$ to the pair $(h(a), b\frac{dh}{da})$ with interpretation $h(a) + b\frac{dh}{da}\epsilon + O(\epsilon^2)$. These pairs are often called dual numbers. Note that we are computing actual derivatives, not approximating with finite differences.

Below, you see a partial implementation of a simple dual number method. In this exercise you're asked to complete the implementation, so you can complete the derivatives of:

$$f_1(x) = sin(x)$$

$$f_2(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$$

```
0.5000000000000001
```

```julia
begin
    # This code was based on: https://book.sciml.ai/notes/08-Forward-
    Mode_Automatic_Differentiation_(AD)_via_High_Dimensional_Algebras/

    struct Dual{T}
        val::T   # value
        der::T  # derivative
    end

    # some calculus rules
    Base.:+(f::Dual, g::Dual) = Dual(f.val + g.val, f.der + g.der)
    Base.:+(f::Dual, α::Number) = Dual(f.val + α, f.der) # for expressions
    Dual(a,b)+2.0
    Base.:+(α::Number, f::Dual) = f + α

    Base.:sin(f::Dual) = Dual(sin(f.val), cos(f.val) * f.der)

    # Compute
    function df_dx_dual(f::Function, x::Number)
        y = f(Dual(x, 1.0))
        return y.der
    end

    f_1(x)=sin(x)

    @show df_dx_dual(f_1, π/3)
end
```

```
df_dx_dual(f_1, π / 3) = 0.5000000000000001
```

```
0.0
```

```julia
begin
    # Complete where you see TODO
    const TODO=0.0 # just a trick to keep incomplete code from crashing

    # First terms of Taylor series for sin(x)
    f_2(x)=x - x^3/factorial(3) + x^5/factorial(5) - x^7/factorial(7) + x^9/
    factorial(9)

    # to power integer, like x^3
    Base.:^(f::Dual, n::Integer) = Dual(f.val^n, TODO)

    # division of two Dual numbers
    Base.:/(f::Dual, g::Dual) = Dual(f.val / g.val, TODO)

    # division of Dual number by a normal number
    Base.:/(f::Dual, α::Number) = Dual(f.val / α, f.der / α)

    # subtraction f - g
    Base.:-(f::Dual, g::Dual) = Dual(f.val - g.val, TODO)

    @show df_dx_dual(f_2, π/3)
end
```

```
df_dx_dual(f_2, π / 3) = 0.0
```

2d: Forward mode for multiple inputs

For larger models, we will have more than one input variable. The state x is often a vector. Show that application of a vector function F to a vector of dual numbers, corresponding to $x + \epsilon e_i$, where e_i is a vector with a one as i'th element and zeros elsewhere will give $\partial F / \partial x_i$.

Answer ...

```julia
md"""
Answer ...
"""
```

2e: Reverse mode

If the number of inputs is large and the number of outputs is small, the reverse mode (adjoint method) is more efficient than the forward mode. There are several packages in Julia that can compute gradients in reverse mode. Here we use ReverseDiff.

As an example consider $\tau_s(t)$ as a function of time and map to the depth-averaged flow at $t = 5\ hours$, i.e. $\bar{u}(T)$. For the representation of $\tau_s(t)$ we use a vector $[\tau_s(t_1), \ldots, \tau_s(t_m)]$ and linear interpolation.

The code below shows a simple example of how to compute the gradient of a function with a reverse mode method.

```
[1.0, 2.0, 3.0]
```

```
begin
    # example function
    g(x)=x[1]+x[2]^2+x[3]^3

    x_ref=[1.0,1.0,1.0]
    gradient_g = ReverseDiff.gradient(g,x_ref) #args: function, input to func

    @show gradient_g
end
```

```
gradient_g = [1.0, 2.0, 3.0]
```

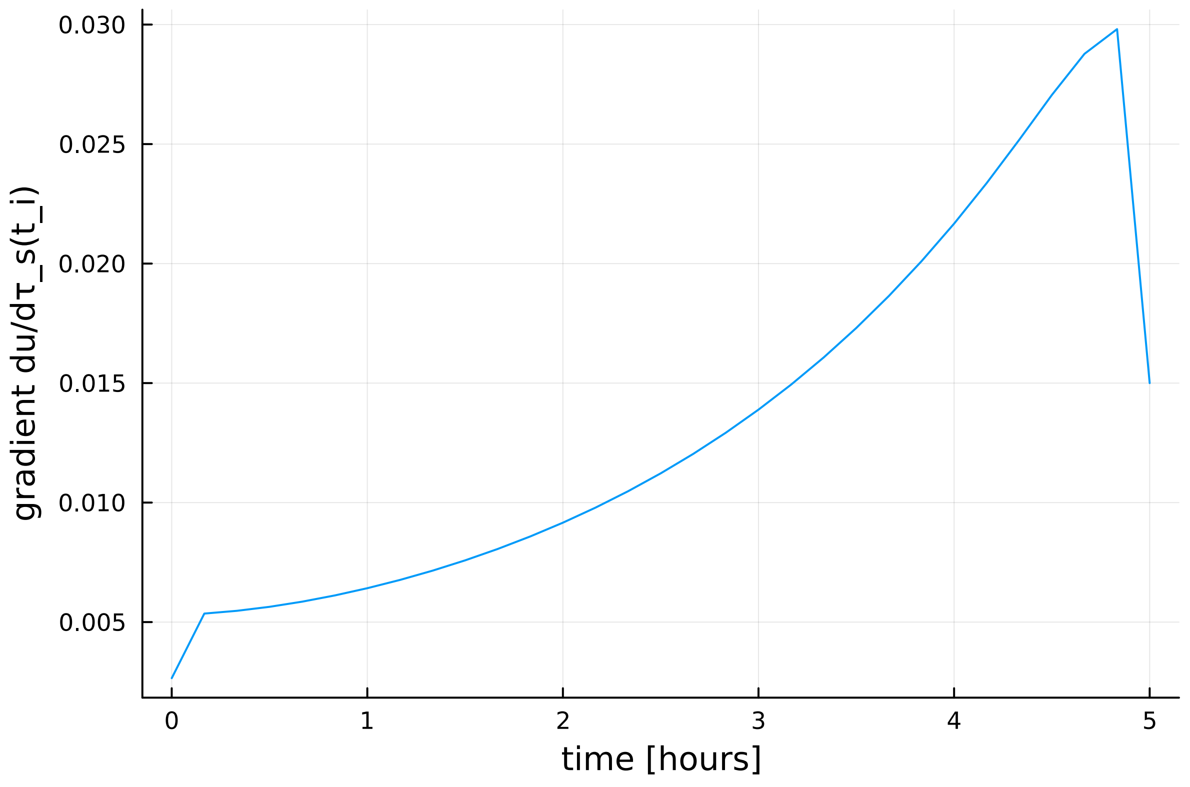
Next, we apply the reverse mode method to the profile model, using a time series for τ_s as input. With the use of the adjoint, it can efficiently compute the gradient even though there are now many parameters.

The output is simplified to the depth averaged velocity $\bar{u}(T) = 1/N_k \sum u_k(T)$, i.e. the gradient $\nabla_{\tau_s(t_i)} \bar{u}(T)$

```
[0.00265223, 0.00535587, 0.00547141, 0.00563844, 0.00585378, 0.00611408, 0.00641732, 0
```

```julia
begin
function stress_series_to_umean(τ_s_values)
    p=τ_s_values #parameters
    n=40
    u₀ = zeros(n)
    tspan=(0.0,5*3600.0)

    prob=ODEProblem(profile,convert.(eltype(τ_s_values),u₀),tspan,p)
    sol=solve(prob,Tsit5(),sensealg=SensitivityADPassThrough())
    umean=mean(sol(tspan[2]))
    return umean
end

t_values=collect(3600.0.*(0:(1/6):5))
τ_s_values=1.0.*ones(length(t_values)) # reference is 1.0 Nm^-2 for all times
τ_s_function=linear_interpolation(t_values, τ_s_values, extrapolation_bc=Line())

umean=stress_series_to_umean(τ_s_values) # evaluate at tau=1.0
@show umean

gradient_umean = ReverseDiff.gradient(stress_series_to_umean,τ_s_values)
@show gradient_umean

end
```

```
umean = 0.5150593659748988
gradient_umean = [0.0026522261806142654, 0.005355868192617231, 0.0054714140
07809018, 0.005638438361393182, 0.005853776763675166, 0.006114082036907772,
0.006417324538425794, 0.00676275334142729, 0.007150687156779636, 0.00758223
1947517239, 0.008059192824022652, 0.008583880460270964, 0.00915915335303420
4, 0.009788274741513388, 0.010475013524424094, 0.011223522196537427, 0.0120
38473207672339, 0.01292494209189312, 0.013888570711774555, 0.01493544727451
7379, 0.016072287999306073, 0.017306295297823116, 0.018645292278628003, 0.0
20097354230686402, 0.02167017599581899, 0.023368029028936612, 0.02518195049
9690135, 0.027055451155433247, 0.028775995162819486, 0.029810291332873187,
0.0149982119375649]
```



If we make a small perturbation $\Delta\tau_s = [\Delta\tau_s(t_1), \ldots, \Delta\tau_s(t_N)]$ to the stress time-series $\tau_s = [\tau_s(t_1), \ldots, \tau_s(t_1)]$, then the impact can be estimated using the gradient. Create a vector $\Delta\tau_s$ based on a Gaussian bump and compare the estimate to the actual change in the mean velocity at final time.

Answer ...

```
md"""
Answer ...
"""
```