




Homework series 2 for Data Assimilation course (WI4475)

This is the second homework series for the data assimilation course (WI4475 2024). 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* $\pi$  #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.

```
1 # Packages used in this notebook. Packages will be downloaded automatically
   which can take a bit of time when you first start the notebook
2 using Plots, PlutoUI, Symbolics, ForwardDiff, DifferentialEquations
```

Question 1: An adjoint for the Burgers' equation

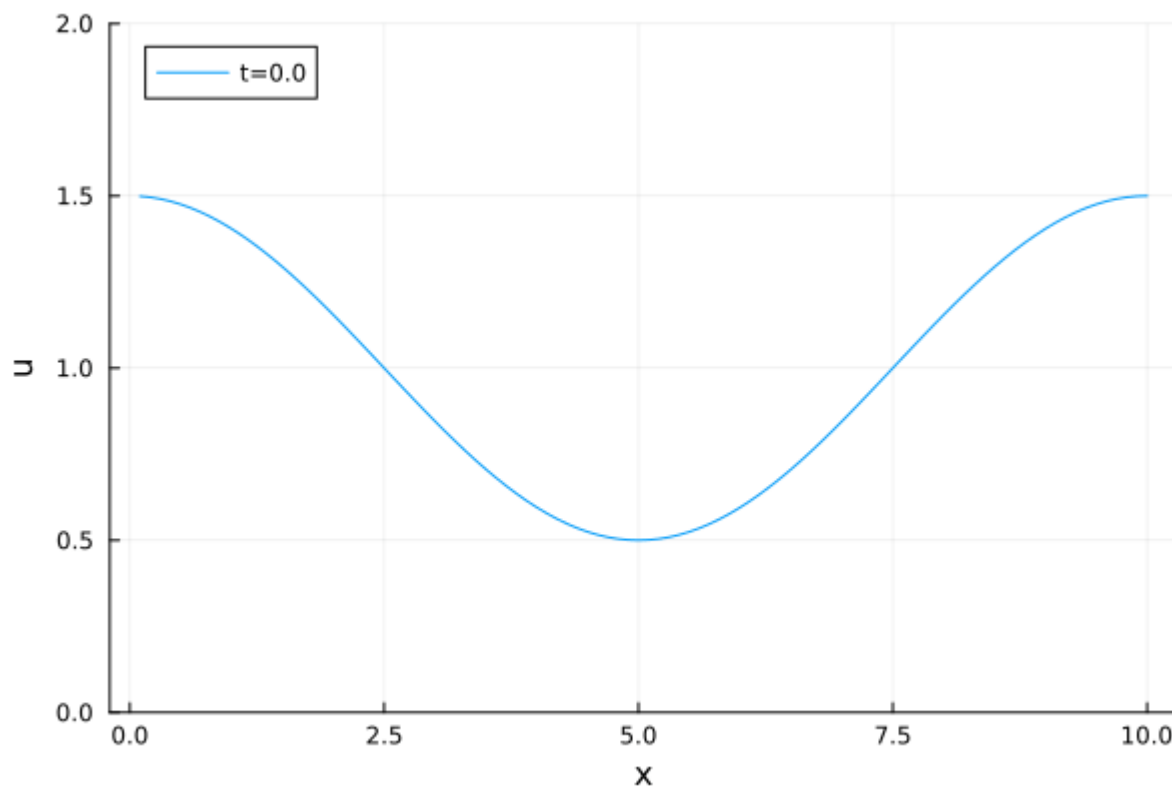
The Burgers' equation is a Partial Differential Equation (PDE) that describes convection and diffusion. It can also be viewed as a simplified version of the Navier-Stokes equation. In this exercise, we start with the equation in the following form:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}$$

on the domain $x \in [0, L]$ and $0 \leq t \leq T$. The domain is considered periodic in x . And the initial condition is given by:

$$u(x, t = 0) = 1.0 + 0.5 \cos(2\pi/Lx)$$

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



Saved animation to /tmp/jl_Poz2BEbEbC.gif

1a: Linearization

The Burgers' equation is non-linear. Before we can find its adjoint it must be linearized. For this, we use a constant equilibrium solution $u(x, t) = \bar{u}$, with $\bar{u} > 0$. To find the linearized equation, we'll substitute a perturbation $u(x, t) = \bar{u} + \epsilon u'(x, t)$ with infinitesimal scalar ϵ . Find the equation for $u'(x, t)$, which will be the linearized equation.

Answer ...

```
1 md"""
2 Answer ...
3 """
```

1b: Adjoint equation

Find the adjoint equation by moving all terms for the linearized equations to one side, ie in the form $F(u') = 0$. Next form the dot product $\int \int F(u'(x, t)) \lambda(x, t) dx dt$ and reorder to $\int \int u'(x, t) F'(\lambda(x, t)) dx dt$, where F' becomes the adjoint equation. Hint: use integration by parts.

Answer ...

```
1 md"""
2 Answer ...
3 """
```

1c: Adjoint for semi-discretized equation

Assuming for simplicity that $u(x, t) > 0$, then we can semi-discretize the Burgers equation as:

$$\frac{du_k}{dt} = -u_k(u_k - u_{k-1})/\Delta x + \nu(u_{k-1} - 2u_k + u_{k+1})/\Delta x^2$$

where $u_k(t)$ is the semi-discretized value of $u(x = k\Delta x, t)$. Note that the notation is sloppy around the periodic boundary. We use $K\Delta x = L$, thus $u_K(t) = u_0(t)$.

Write the equation in the form $d\mathbf{x}/dt = \mathbf{M}(\mathbf{x})$, with \mathbf{x} as the state-vector

$\mathbf{x} = [u_1, u_2, \dots, u_K]'$. Then compute the Jacobian and the adjoint equation as

$$d\lambda/dt = [\partial M/\partial \mathbf{x}]' \lambda$$

Answer ...

```
1 md"""
2 Answer ...
3 """
```

1d: Interpretation of the adjoint state λ

Let's define a cost function that only directly depends on the state of the final time T , ie

$J(\mathbf{x}(T)) = 1/2(\mathbf{z}(T) - \mathbf{H}\mathbf{x}(T))'(\mathbf{z}(T) - \mathbf{H}\mathbf{x}(T))$. Show that the adjoint state $\lambda(t)$ can be interpreted as: $\lambda(t) = \frac{\partial J}{\partial \mathbf{x}(t)}$.

Answer...

```
1 md"""
2 Answer ...
3 """
```

1e: State augmentation with parameter ν

State augmentation is a trick to artificially extend the variables in the state vector $\mathbf{x}(t)$ to cast a more complex problem into a known form. This is often easier than deriving the equations for the extended problem from scratch (which you should also be able to do). For example, one can extend the case above (1d) to include a parameter, by adding the parameter to the state vector, with the equation $dp/dt = 0$. Use this approach to find the equations to find the gradient for the parameter ν in the semi-discretized Burgers' equation building on exercises 1c and 1d. As we see in 1d the adjoint at $t = 0$ will give us $\lambda(t) = \frac{\partial J}{\partial \mathbf{x}(t)}$. If we have included ν into this vector, this will include the derivative wrt ν .

Answer...

```
1 md"""
2 Answer ...
3 """
```

Question 2: The logistic map

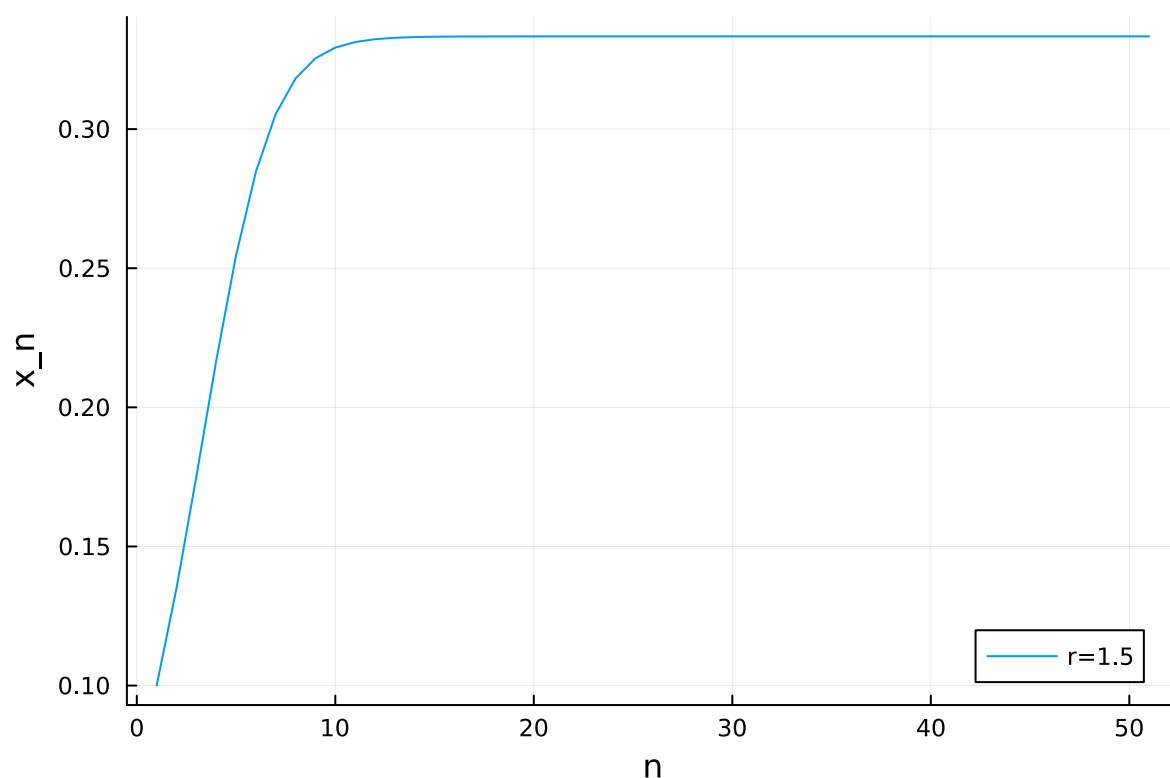
In question one you explored the manual derivation of the adjoint equations, that we need for variational data assimilation methods. For larger models, the manual derivation of the adjoint equations becomes a tedious and error-prone task. Until recently this was resulting in a shift from variational methods to ensemble methods for data assimilation. However, recently automated computer techniques are becoming available, that were mostly developed for machine learning. You are much more likely to use these methods in practice than to derive an adjoint manually after your graduation. This question explores this topic a bit.

The logistic map is a simple yet interesting operator of the form:

$$x_{n+1} = r * x_n * (1 - x_n)$$

We start with $x_0 = 0.1$ and let n run from 1 to 50. With the slider below you can change the value of r and see how this changes the dynamics from damped, to periodic to chaotic.

```
1 f(x,r)=r*x*(1.0-x); # define the logistic map
```



2a: exploding number of terms

With a sequence like this the number of terms in the derivative of $\frac{\partial x_n}{\partial x_0}$ usually grows exponentially with n , as you can see below. For this reason it is often a bad idea to expand the computation into one formula and compute the derivatives like this.

Why does this happen? And can you think of a sequence $y_{n+1} = g(y_n)$ where this doesn't

happen? What are the requirements for such a function?



$n=1$

$x_1=$

$$r_s(1 - x_0)x_0$$

$$\frac{dx_n}{dx_0} =$$

$$-r_s x_0 + r_s(1 - x_0)$$

Answer ...

```
1 md"""
2 Answer ...
3 """
```

2b: Symbolic derivatives

So, instead of expressing the derivative directly, we aim for a sequence of steps that expresses the derivative. The first way to achieve this is to turn the function into a symbolic expression, compute the derivative symbolically, and finally generate code for this expression. So, here the pattern is from code to formula to code again. Below, you can see this in action. Assuming that an implementation of df/dx exists, how can you compute dx_n/dx_0 in forward mode? With forward mode we mean first dx_1/dx_0 then dx_2/dx_0 , etc.

```

1 begin
2   # starting with f as a regular function.
3   @variables x_s
4   f_symbolic = f(x_s, r_s) #evaluate the function with a symbolic variable as input
5   @show f_symbolic #this gives a symbolic formula
6   df_dx_symbolic = Symbolics.derivative(f_symbolic, x_s) #compute the derivative
7   @show df_dx_symbolic
8   df_dx = build_function(df_dx_symbolic, x_s, r_s, expression = Val{false})
9   @show df_dx #this is a regular function
10  @time x_1 = df_dx(0.1, 1.5) #the first time may include the compilation time in
11  Julia
12  @time x_1 = df_dx(0.1, 1.5) #runs in a micro-second on my computer
13  @show x_1
14  nothing
end

```

```

f_symbolic = r_s*(1.0 - x_s)*x_s
df_dx_symbolic = -r_s*x_s + r_s*(1.0 - x_s)
df_dx = RuntimeGeneratedFunctions.RuntimeGeneratedFunction{(:x_s, :r_s), Symbolics.var"#_RGF_ModTag", Symbolics.var"#_RGF_ModTag", (0xf8ad5c7b, 0x76a871c2, 0xaa4589ce, 0x35b61446, 0x45822736), Expr}(quote
    #= /home/verlaan/.julia/packages/SymbolicUtils/r1pzW/src/code.jl:373 =#
    #= /home/verlaan/.julia/packages/SymbolicUtils/r1pzW/src/code.jl:374 =#
    #= /home/verlaan/.julia/packages/SymbolicUtils/r1pzW/src/code.jl:375 =#
    (+)((*)(*)(-1, r_s), x_s), (*(r_s, (+)(1.0, (*( -1, x_s))))))
end)
0.000002 seconds (1 allocation: 16 bytes)
0.000000 seconds (1 allocation: 16 bytes)
x_1 = 1.2000000000000002

```

Answer ...

```

1 md"""
2 Answer ...
3 """

```

2c: Forward mode with dual numbers

Another approach to automate derivatives is to think of a computation as a sequence of elementary steps, like addition and multiplication. For each of the steps we know the calculus rules, so we can implement them one at a time. In a modern language like Julia or Python, we can then recompile the computation for our extended type. A common method is to start, for example with infinitesimals $x + \epsilon 1$ and apply the function $h(x) = x^2$ to that. 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.

Write the logistic map f as a sequence of elementary operations, then work out the operations for the infinitesimals and compute the derivative df/dx . Then extend the code below to

compute a derivative for the logistic map f .

Answer ...

```
1 md"""
2 Answer ...
3 """
```

0.50000000000000001

```
1 begin
2 # This code was based on: https://book.sciml.ai/notes/08-Forward-
  Mode_Automatic_Differentiation_(AD)_via_High_Dimensional_Algebras/
3
4 struct Dual{T}
5     val::T    # value
6     der::T    # derivative
7 end
8
9 Base.:+(f::Dual, g::Dual) = Dual(f.val + g.val, f.der + g.der)
10 Base.:+(f::Dual, α::Number) = Dual(f.val + α, f.der) # for expressions Dual(a,b)
11 +2.0
12 Base.:+(α::Number, f::Dual) = f + α
13
14 Base.:sin(f::Dual) = Dual(sin(f.val), cos(f.val) * f.der)
15 function df_dx_dual(f::Function, x::Dual)
16     y = f(Dual(x.val, 1.0))
17     return y.der
18 end
19 @show df_dx_dual(sin, Dual(π/3, 1.0))
end
```

```
df_dx_dual(sin, Dual(π / 3, 1.0)) = 0.50000000000000001
```



2d: Forward mode for multiple inputs

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

Answer ...

```
1 md"""
2 Answer ...
3 """
```

2e: Forward mode in action

In Julia there is no need to implement dual numbers yourself, since the forward mode differentiation is available through the ForwardDiff package. In this exercise, we'll use this to find an estimate for the parameter \mathbf{r} .

First change the program below to compute the sum of the squared differences. Then compute $dJ/d\mathbf{r}$ and use gradient descent to find a good estimate of \mathbf{r} . This results in a very small working

parameter estimation method, that is easily extended to larger problems.

```
1 begin
2   # observations : these were generated with an unknown r value, but with
3   correct x0
4   # The values correspond to x_1 ... x_10
5   z = [0.27000000000000001, 0.59130000000000002, 0.72499292999999999,
6         0.5981345443500454, 0.7211088336156269, 0.603332651091411,
7         0.7179670896552621, 0.6074710434816448, 0.7153499244388992,
8         0.6108732301324811]
9
10  r_init=2.75
11
12  function J(r)
13    result=0.0
14    x=x0
15    for i=1:10
16      x=f(x,r)
17      result=result+x^2 # ADAPT THE CODE
18    end
19    return result
20  end
21
22  @show J(r_init)
23
24  dJ_dr = ForwardDiff.derivative(J,r_init)
25  @show dJ_dr
26
27  nothing
end
```

```
J(r_init) = 3.589884842490911
dJ_dr = 1.677225433553669
```

