Vedant Puri
veantpu@andrew.cmu.edu

Homework 2

Question 1: In image processing, boundary detection is the most critical step. Given a 2D image $\phi(x, y)$, how to detect boundaries of different regions?

Answer: Boundary detection is the most important step in image processing. Given a $2D$ image, $\phi(x, y)$, the boundary would be indicated by sharp changes in the signal. These areas can be identified by large magnitudes of the gradient vector. $\boldsymbol{\nabla}\phi$ is oriented in the direction of steepest change in image intensity, normal to the implied boundary. The gradient magnitude, $|\boldsymbol{\nabla}\phi|$ represents an orientation-independent measure of the boundary strength.

It is essential, however, to first denoise or smooth the image by applying a smoother filter such as a Gaussian. That is because miscellaneous noise can produce large gradients even in the absence of a boundary.

Question 2: Implement the following linear and non-linear filters using the finite different method to smooth the image $\phi(x, y)$, and apply your code to the given 2D image (`foot.pgm`). Please output your results in `.pgm` format and visualize them using `IrfanView` or other software.

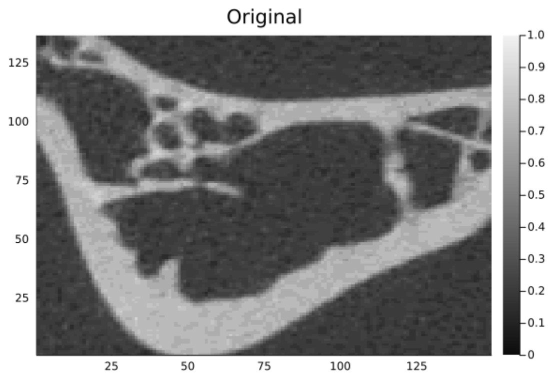

Figure 1: `.pgm` image of a human foot

Answer: It is clear to see that the original image of the human foot has a lot of noise to it: both the black and the white parts appear to be grainy and full of artifacts. In order to denoise the image for downstream tasks, we experiment with linear and nonlinear smoothing filters. We utilize the Julia programming language for these computations with finite difference operators being implemented via convolution operations. The convolution operations are made available by the library `NNlib.jl`, and visualizations using `Plots.jl`. We apply appropriate padding to the convolution operation so as to ensure that the size of the image does not change. Consequently, the finite difference stencil isn't properly applied at the boundaries of the image and those points should be ignored. The convolution kernels are:

```
"""
Convention:
    * * * *
    * * * * | x
    * * * * |
    * * * * v
    * * * * x
     ---> y
"""

const LAPL_F32 = [0f0  1f0 0f0
                  1f0 -4f0 1f0
                  0f0  1f0 0f0] ./ 1f0

const DX_F32 = [-1f0
                 0f0
                 1f0] ./ 2f0

const DY_F32 = [-1f0 0f0 1f0] ./ 2f0
```

We use a simple forward Euler method for time-integration with a fixed step size of $\Delta t = 0.01$ for each filter. Eq. 1 describes the time-stepping scheme for a problem of the type $\frac{du}{dt} = f(u(t), t)$ with Euler-forward time-integrator. For each filter, we pass the right-hand-side function $f(u(t), t)$ to the time-stepper which evolves the system for a set number of iterations.

$$\frac{u^{t_{n+1}} - u^{t_n}}{\Delta t} = f(u^{t_n}, t_n) \tag{1}$$

a) **Linear filtering**:

The filter is described in Eq. 2 and is implemented by convolving the 2D finite difference stencil for the Laplacian, `LAPL_F32` at each time-step. The Laplacian is inherently a smoothing operator that performs local averaging in the interior of the computational domain. Figure 2 shows the result from Laplacian smoothing for 50 iterations and 200 with $\Delta t = 0.01$. These results illustrate that as you continue smoothing

$$\partial_t \phi = \Delta \phi \tag{2}$$

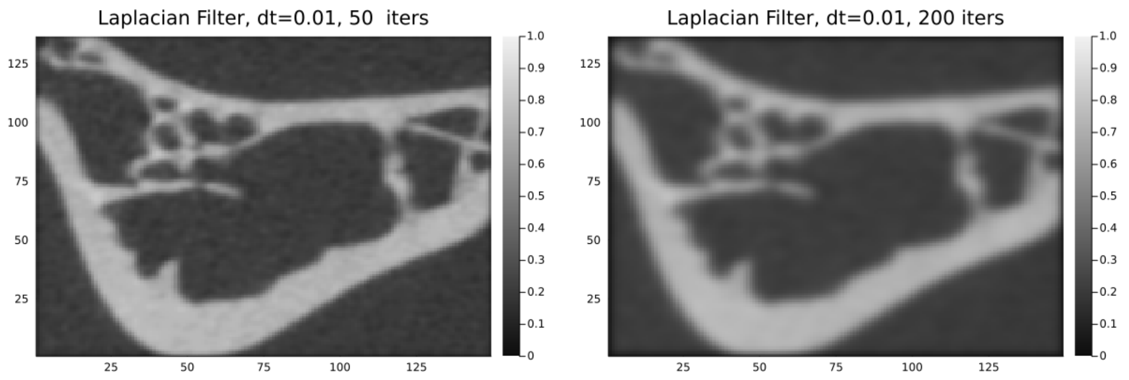
Laplacian Filter, dt=0.01, 50 iters
Laplacian Filter, dt=0.01, 200 iters

Figure 2: Application of linear Laplace smoother for 50 iterations (left), and 200 iterations (right) at $\Delta t = 0.01$

b) **Nonlinear filtering**:

The nonlinear filter described in Eq. 3 performs similar Laplaican-type smoothing (divergence of the gradient) with a nonlinearity introduced by $g(x) = \dfrac{1}{1 + x^2/\lambda^2}$ which is a Gaussian type smoother. $g(|\boldsymbol{\nabla}\phi|)$ is small for larger values of $|\boldsymbol{\nabla}\phi|$, and vice versa, for a fixed λ. This scales down $\boldsymbol{\nabla}\phi$, and penalizes areas with large noise (as they correspond with large $|\boldsymbol{\nabla}\phi|$). λ modulates the amount of penalization, and smoothing increases with λ. The gradient, and divergence computations are done by applying the convolution filters `DX_F32, DY_F32` for x-derivative, and y-derivative computations respectively. As is clear from the results in Figure 3, the increasing the value of λ from 0.1 to 10 dramatically increases the amount of smoothing or blurring in the image for a fixed Δt, and number of iterations.

$$\begin{aligned} \partial_t \phi &= \boldsymbol{\nabla} \cdot (g(|\boldsymbol{\nabla}\phi|)\boldsymbol{\nabla}\phi) \\ g(a) &= \frac{1}{1 + a^2/\lambda^2} \end{aligned} \tag{3}$$

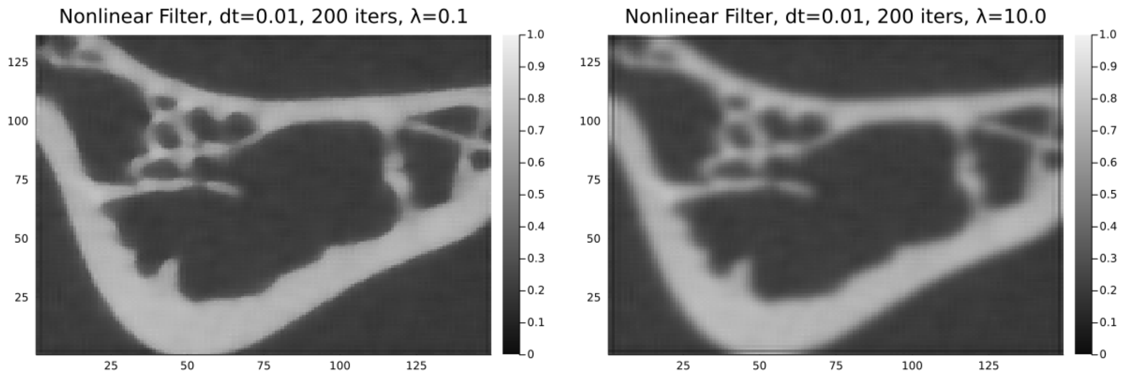
Nonlinear Filter, dt=0.01, 200 iters, λ=0.1
Nonlinear Filter, dt=0.01, 200 iters, λ=10.0

Figure 3: Application of Nonlinear smoother for 200 iterations at $\Delta t = 0.01$ with $\lambda = 0.1$ (left), and $\lambda = 10.0$(right)

Vedant Puri
veantpu@andrew.cmu.edu

Homework 2

ME 24-658
September 21, 2022

Code: The code for this problem has been compiled into a Julia package hosted on a private Github repo at `https://github.com/vpuri3/ImageAnalysis.jl`. A zipped version of the repo is shared with this document. To run the code, you need to install Julia v1.6 or greater from `https://julialang.org/downloads/` (takes less than a minute). Then the results can be generated in directory `ImageAnalysis/homeworks/homework2` by running the script `ImageAnalysis/homeworks/homework2/RUN`. For brevity, relevant sections of the code are reproduced in this document.

`ImageAnalysis.jl`

```julia
module ImageAnalysis

using NNlib: conv

"""
Convention:
    * * * *
    * * * * | x
    * * * * |
    * * * * v
    * * * * x
     ---> y
"""

#=======================================================================#
# CONVOLUTIONS
#=======================================================================#

const LAPL_F32 = [0f0  1f0 0f0
                  1f0 -4f0 1f0
                  0f0  1f0 0f0] ./ 1f0

const DX_F32 = [-1f0
                 0f0
                 1f0] ./ 2f0

const DY_F32 = [-1f0 0f0 1f0] ./ 2f0

function apply_conv(u::AbstractMatrix{T},
                    w::AbstractVecOrMat{T}) where{T}

    U = reshape(u, (size(u)..., 1, 1))

    W = if w isa AbstractVector
        reshape(w, (size(w)..., 1, 1, 1))
    else
```

```
        reshape(w, (size(w)..., 1, 1))
    end

    pad = size(W)[1:2] .÷ 2

    V = conv(U, W; pad=pad)

    dropdims(V; dims=(3,4))
end

#==============================================================#
# VECTOR CALCULUS
#==============================================================#

function grad(u::AbstractArray, ws=(DX_F32, DY_F32))
    Tuple(apply_conv(u, w) for w in ws)
end

function diver(us::NTuple{D, AbstractArray},
               ws::NTuple{D, AbstractArray}=(DX_F32, DY_F32)) where{D}
    tup = Tuple(apply_conv(u, w) for (u,w) in zip(us, ws))

    sum(tup)
end

function norm2(us::NTuple{D, AbstractArray}) where{D}
    u2s = Tuple(u .* u for u in us)
    sum(u2s)
end

#==============================================================#
# GAUSSIAN SMOOTHER
#==============================================================#

function gauss(x, )
    x2 = x * x
    l2 =   *
    gauss2(x2, l2)
end

function gauss2(x2, l2)
    1 / (1+ x2/l2)
end

#==============================================================#
# TIME STEPPERS
#==============================================================#
```

```

function euler_fwd(u, dudt_func; dt=0.01f0, niter=100)
    for i=1:niter
        du = dudt_func(u)
        u += dt*du
    end

    u
end

#===============================================================#

export
       LAPL_F32, DX_F32, DY_F32,

       apply_conv,

       grad, diver, norm2,

       gauss, gauss2,

       euler_fwd

#===============================================================#

end # module
```

driver.jl

```
#
println("Activating environment")
import Pkg
Pkg.activate("../..")
Pkg.instantiate()

println("Importing ImageAnalysis.jl")
using ImageAnalysis

println("Importing external packages")
using Images: load, save
using Plots: plot, plot!, heatmap, savefig

println("Loading image data")
img = load("foot.pgm") .|> Float32

```

```
"""
dudt function for linear Laplace smoother

to be passed down to a time-stepper
"""
function dudt_ln(u::AbstractArray)
    apply_conv(u, LAPL_F32)
end

"""
 dudt function for nonlinear Gaussian smoother

to be passed down to a time-stepper
"""
function dudt_nl(u::AbstractArray; =1f0)
    l2  = ^2
    u  = grad(u)
    u2 = norm2(u)
    g   = gauss2.(u2, l2)

    rhs = Tuple(g .* uxi for uxi in u)

    diver(rhs)
end

#
l1 = 1f-1
l2 = 1f+1

println("Applying linear filter with dt=0.01 for 50 iterations, and 200 iterations")
ln1 = euler_fwd(img, dudt_ln; dt=0.01f0, niter=50)
ln2 = euler_fwd(img, dudt_ln; dt=0.01f0, niter=200)

println("Applying nonlinear filter with dt=0.01 with =0.1, 10.0 for 200 iterations")
nl1 = euler_fwd(img, u -> dudt_nl(u; =l1); dt=0.01f0, niter=200)
nl2 = euler_fwd(img, u -> dudt_nl(u; =l2); dt=0.01f0, niter=200)

println("Producing plots")
p0 = heatmap(img[end:-1:begin, :]; clims=(0,1), c=:grays)
p1 = heatmap(ln1[end:-1:begin, :]; clims=(0,1), c=:grays)
p2 = heatmap(ln2[end:-1:begin, :]; clims=(0,1), c=:grays)
p3 = heatmap(nl1[end:-1:begin, :]; clims=(0,1), c=:grays)
p4 = heatmap(nl2[end:-1:begin, :]; clims=(0,1), c=:grays)

p0 = plot!(p0, title="Original")
```

```julia
p1 = plot!(p1, title="Laplacian Filter, dt=0.01, 50  iters")
p2 = plot!(p2, title="Laplacian Filter, dt=0.01, 200 iters")
p3 = plot!(p3, title="Nonlinear Filter, dt=0.01, 200 iters, =$l1")
p4 = plot!(p4, title="Nonlinear Filter, dt=0.01, 200 iters, =$l2")

filename = "foot_smooth"

println("Saving files $(filename)*.png")

savefig(p0, "foot")
savefig(p1, filename * "_ln1")
savefig(p2, filename * "_ln2")
savefig(p3, filename * "_nl1")
savefig(p4, filename * "_nl2")

println("Saving files $(filename)*.pgm")

save(filename * "_ln1" * ".pgm", ln1)
save(filename * "_ln2" * ".pgm", ln2)
save(filename * "_nl1" * ".pgm", nl1)
save(filename * "_nl2" * ".pgm", nl2)
#
```