`reciandenoise`: 2D and 3D Total Variation Based Rician Denoising

Pascal Getreuer, Sep 2009

Package Contents

`riciandenoise_demo.m`	A demo of 2D denoising
`riciandenoise3_demo.m`	A demo of 3D denoising
`riciandenoise.m`	2D TV Rician denoising (M-code)
`riciandenoisemx.c`	2D TV Rician denoising (C/MEX)
`riciandenoise3mx.c`	3D TV Rician denoising (C/MEX)
`ricianrnd.m`	A function for simulating Rician noise
`volslice.m`	A function for plotting volume slices
`riciandenoise.pdf`	This document

How to use `reciandenoise`

The functions `reciandenoise` and `reciandenoisemx` implement TV-based denoising on 2D images, and they are essentially the same but are implemented respectively in MATLAB M-code and C (callable from MATLAB via MEX). The function `reciandenoise3mx` implements the denoising in 3D.

The `reciandenoisemx` and `reciandenoise3mx` codes need to be compiled before use. Provided MATLAB has already been configured, you can compile the C/MEX functions by the commands

```
mex reciandenoisemx.c
mex reciandenoise3mx.c
```

The calling syntax for `reciandenoise` is

Syntax: `u = reciandenoise(f,sigma,lambda,Tol)`

and similarly for `reciandenoisemx` and `reciandenoise3mx`. Here, `f` is the noisy input data: it is a 2D array for `reciandenoise` and `reciandenoisemx`, and it is a 3D array for

`reciandenoise3mx`. The pixels `f(i,j)` are assumed to be i.i.d. Rician distributed, i.e., with probability density

$$P(f_{i,j}|u_{i,j},\sigma) = \frac{f_{i,j}}{\sigma^2}\exp\left(\frac{-(f_{i,j}^2+u_{i,j}^2)}{2\sigma^2}\right) I_0\left(\frac{u_{i,j}f_{i,j}}{\sigma^2}\right), \tag{1}$$

where u is the clean image that we are attempting to recover, σ is a positive parameter (specified by the second argument `sigma`), and I_0 is the modified Bessel function of the first kind of order zero. Parameter `lambda` ≥ 0 controls the denoising strength, where smaller `lambda` implies stronger denoising. `Tol` is the stopping tolerance, which should be a value on the order of $(\max f) \times 10^{-3}$ or smaller. The output `u` is the denoised result.

As part of the denoising method, `riciandenoise` must approximate the function $I_1(x)/I_0(x)$. The calling syntax

```
riciandenoise(...,FastApprox)
```

specifies how this function should be approximated. `FastApprox` is true, then an efficient and moderately accurate cubic rational approximation of $I_1(x)/I_0(x)$ is used (error less than 8×10^{-4} for all $x \geq 0$). Otherwise, if `FastApprox` is false, MATLAB's very accurate but more time-consuming `besseli` function is used. If unspecified, `riciandenoise` uses the cubic rational approximation.

The C/MEX codes `riciandenoisemx` and `riciandenoise3mx` do not accept parameter `FastApprox` in the calling syntax; they always use the cubic rational approximation. Also, other differences from the M-code version is that, for simplicity, `riciandenoisemx` and `riciandenoise3mx` only denoise the interior of the image leaving a one-pixel border around the image equal to f. One final difference: the M-code vs. C/MEX versions use a slightly different update formula that will produce visually similar but numerically different results (this is explained in more detail later).

Usage Demo

The following demonstrates how to use `riciandenoise`. This example is the same as in `riciandenoise_demo`.

First, to prepare the input data, we read a clean image from a file and rescale it so that intensity values are in the range $[0, 1]$. We then call `ricianrnd` to simulate Rician noise of parameter $\sigma = 0.05$.

```
% Get a clean input image
uexact = imread('knee-mri.png');
uexact = mean(double(uexact)/255,3);
% Simulate Rician noise
sigma = 0.05;
f = ricianrnd(uexact,sigma);
```

Then we call the denoising function as

```
lambda = 0.065;  % Denoising strength parameter
Tol = 2e-3;      % Convergence tolerance
u = riciandenoise(f,sigma,lambda,Tol);
```

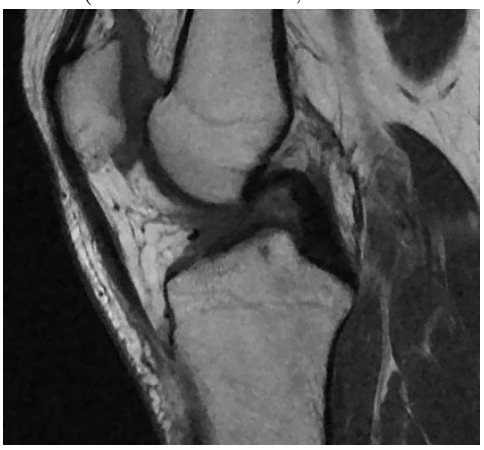
When `riciandenoise` returns, it prints a message on the console telling whether it converged and the number of method iterations*:

```
Converged in 14 iterations with tolerance 0.002.
```

Finally, we plot the results and the PSNRs.

Input (PSNR 25.81 dB)

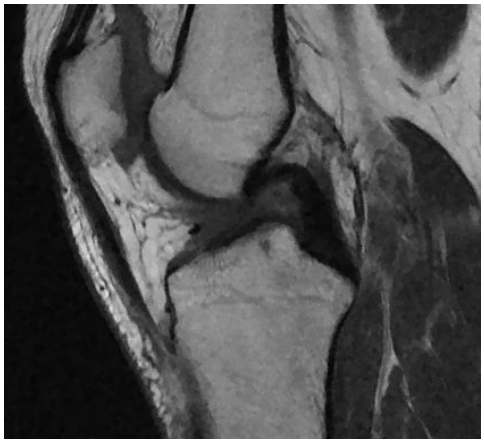
Denoised (PSNR 32.32 dB, CPU time 0.72 s)



We can also perform the denoising with the C/MEX version `riciandenoisemx`:

```
u = riciandenoisemx(f,sigma,lambda,Tol);
```

Denoised (PSNR 32.53 dB, CPU time 0.16 s)



*Results may vary slightly due to the pseudo-randomness of `f`.

Derivation

The Rudin-Osher-Fatemi (ROF) model for Rician noise is

$$\min_{u\in BV(\Omega)} \int_\Omega |\nabla u|\ dx + \lambda \int_\Omega \left[\frac{u^2+f^2}{2\sigma^2} - \log I_0\left(\frac{uf}{\sigma^2}\right)\right] dx \tag{2}$$

where the denoised image u is found as the minimizer over all images. We assume that the noisy image f is nonnegative with values in $[0,1]$. Unfortunately, for f/σ^2 sufficiently small, the objective is nonconvex. This means that we cannot gaurantee finding global minimizers, though we can still find local minimizers.

We can find these local minimizers by performing gradient descent. Gradient descent on (2) is the partial differential equation

$$\partial_t u = \operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right) - \gamma u + \gamma \frac{I_1(\frac{uf}{\sigma^2})}{I_0(\frac{uf}{\sigma^2})} f. \tag{3}$$

where $\gamma = \frac{\lambda}{\sigma^2}$. Numerically, we can approximate the curvature term $\operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right)$ as

$$\begin{aligned}
\operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right) &\approx g_{i+1,j}(u_{i+1,j}-u_{i,j}) + g_{i-1,j}(u_{i-1,j}-u_{i,j}) \\
&\quad + g_{i,j+1}(u_{i,j+1}-u_{i,j}) + g_{i,j-1}(u_{i,j-1}-u_{i,j}), \\
\frac{1}{|\nabla u|} \approx g_{i,j} &:= \big[\epsilon + (u_{i+1,j}-u_{i,j})^2 + (u_{i-1,j}-u_{i,j})^2 \\
&\quad + (u_{i,j+1}-u_{i,j})^2 + (u_{i,j-1}-u_{i,j})^2\big]^{-1/2}.
\end{aligned}$$

For 3D denoising, these formulas should also include the axial neighbors in the z-direction, $(\cdot)_{i,j,k+1}$ and $(\cdot)_{i,j,k-1}$. Let $\mathcal{N}_{i,j}$ denote the neighbors of (i,j) such that the approximation is written concisely as

$$\operatorname{div}\left(\frac{\nabla u}{|\nabla u|}\right) \approx \sum_{n\in\mathcal{N}_{i,j}} g_n(u_n - u_{i,j}).$$

This approximation enables a semi-implicit update scheme:

$$\begin{aligned}
\frac{u_{i,j}^{k+1}-u_{i,j}^k}{dt} &= \sum_{n\in\mathcal{N}_{i,j}} g_n^k(u_n^k - u_{i,j}^{k+1}) - \gamma u_{i,j}^{k+1} + \gamma \tfrac{I_1(u_{i,j}^k f_{i,j}/\sigma^2)}{I_0(u_{i,j}^k f_{i,j}/\sigma^2)} f_{i,j} \\
u_{i,j}^{k+1} &= \frac{u_{i,j}^k + dt\big(\sum_n g_n^k u_n^k + \gamma \frac{I_1(u_{i,j}^k f_{i,j}/\sigma^2)}{I_0(u_{i,j}^k f_{i,j}/\sigma^2)} f_{i,j}\big)}{1 + dt\big(\sum_n g_n^k + \gamma\big)}.
\end{aligned} \tag{4}$$

Since g_n and γ are nonnegative and $0 \le \frac{I_1(\cdots)}{I_0(\cdots)} f \le 1$, the scheme is unconditionally stable for any nonnegative timestep dt.

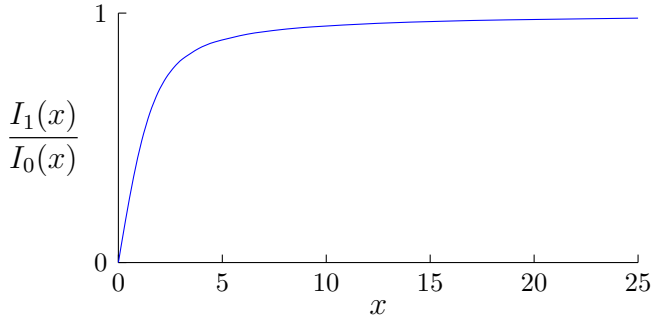
The update formula (4) is a nonlinear Jabacoi iteration, where all $u_{i,j}^{k+1}$ are computed from u^k. The update can also be applied as a nonlinear Gauss-Sidel iteration, e.g. as

$$u_{i,j}^{k+1} = \frac{u_{i,j}^k + dt\big(g_{i+1,j}^k u_{i+1,j}^k + g_{i,j+1}^k u_{i,j+1}^k + g_{i-1,j}^k u_{i-1,j}^{k+1} + g_{i,j-1}^k u_{i,j-1}^{k+1} + \gamma(\cdots)\big)}{1 + dt(\cdots)}. \tag{5}$$

The advantage computationally is that u can be stored in a single array and updated directly, whereas the Jacobi iteration requires a second array to hold the intermediate result for u^{k+1} while computing it from u^k. In the implementations, the C/MEX version `riciandenoisemx` uses Gauss-Sidel iteration (5) and the M-code version `riciandenoise` uses Jacobi iteration (4).

Approximation of I_1/I_0

Solving (3) requires approximating the function $I_1(x)/I_0(x)$, where I_0 and I_1 are the modified Bessel functions of the first kind of orders zero and one.



The function $I_1(x)/I_0(x)$ is zero at $x = 0$ and increases monotonically to one. MATLAB's `besseli` function can approximate them very accurately (nearly to machine precision), based on a FORTRAN package by Amos.

However, `besseli` is very time-consuming. The MATLAB profiler reveals that `riciandenoise` spends over 90% of its time in `besseli`. For this reason, we develop an alternative for approximating $I_1(x)/I_0(x)$ more efficiently.

The approximation is a cubic rational approximation of the form

$$\frac{I_1(x)}{I_0(x)} \approx \frac{p(x)}{q(x)} = \frac{x^3 + p_2 x^2 + p_1 x}{x^3 + q_2 x^2 + q_1 x + q_0}.$$

Some of the coefficients of p and q are already determined so that the approximation satisfies $p(0)/q(0) = 0$ and $p(x)/q(x) \to 1$ as $x \to \infty$. The five remaining coefficients can

be selected such that the approximation interpolates $I_1(x)/I_0(x)$ at five points:

$$\frac{I_1(x_i)}{I_0(x_i)} = \frac{p(x_i)}{q(x_i)}, \quad i = 1, \ldots, 5.$$

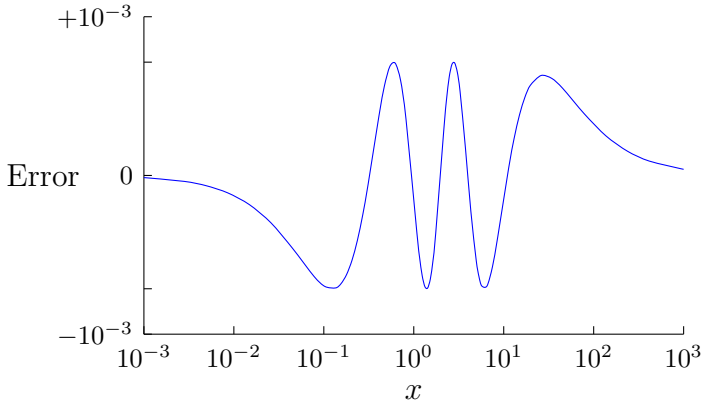
We select the (x_i) as to minimize the approximation error:

$$\min_{(x_i)} \|I_1/I_0 - p/q\|_\infty .$$

The solution is approximately $x_1 = 0.322495$, $x_2 = 0.944725$, $x_3 = 1.97199$, $x_4 = 3.98091$, $x_5 = 10.9936$, which is

$$\frac{I_1(x)}{I_0(x)} \approx \frac{x^3 + 0.950037x^2 + 2.38944x}{x^3 + 1.48937x^2 + 2.57541x + 4.65314},$$

with maximum approximation error of 7.14×10^{-4}.



Similarly, approximations of other degrees can be constructed. It seems that cubic degree is sufficiently accurate for image denoising. For example, for the `reciandenoise` demo shown earlier, the maximum difference between the results with the cubic rational approximation vs. `besseli` is 2.76×10^{-4} (where intensities are scaled to $[0, 1]$), which is well below significance if the result is quantized to 256 levels.