tvreg v2: Variational Imaging Methods for Denoising, Deconvolution, Inpainting, and Segmentation

Pascal Getreuer, December 2010

Copyright © 2009–2010, Pascal Getreuer
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This material is based upon work supported by the National Science Foundation under Award No. DMS-1004694. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

1 Overview

The tvreg package applies total variation (TV) regularization [25] to perform image denoising, deconvolution, and inpainting. Three different noise models are supported: additive white Gaussian noise (traditional L^2 data fidelity), Laplace noise (L^1 data fidelity), and Poisson. The implementation solves the general TV restoration problem

$$\min_{u \in BV(\Omega)} \int_\Omega |\nabla u(x)| \, dx + \int_\Omega \lambda(x) F\big(Ku(x), f(x)\big) \, dx$$

to perform denoising, deconvolution, and inpainting as special cases. It is efficiently solved using the split Bregman method [17]. Also included is an efficient implementation of Chan-Vese two-phase segmentation [8]. All functions support grayscale, color, and arbitrary multichannel images.

⚠ **These are computationally demanding programs!** Using them on large images will consume a great deal of memory and may take a long time to complete. Be cautious and experiment first with medium-sized images.

Get Started Quickly

1. Install the FFTW library (`http://www.fftw.org`). Windows users can download precompiled .dll files `http://www.fftw.org/install/windows.html`.
2. Compile the programs with GCC using `make -f makefile.gcc` or Microsoft Visual C++ using `nmake -f makefile.vc`. See section 7 for help.
3. Try the demos

`tvdenoise_demo`	Total variation denoising demo
`tvdeconv_demo`	Total variation deconvolution demo
`tvinpaint_demo`	Total variation inpainting demo
`chanvese_demo`	Chan-Vese segmentation demo

Get Started Quickly in MATLAB

Compiling is not required to use tvreg in MATLAB. Try the demos

`tvdenoise_demo`	Total variation denoising demo
`tvdeconv_demo`	Total variation deconvolution demo
`tvinpaint_demo`	Total variation inpainting demo
`chanvese_demo`	Chan-Vese segmentation demo

For improved performance, run the included script `complex_mex.m` to compile the main computation routines as MEX functions. This requires that FFTW is installed, please see section 7.3.

1.1 Changes in tvreg v2

The significant change in this version of tvreg is that it is now written entirely in ANSI C code. It may be called from C/C++ programs and used as a standalone command line program. tvreg may still be used from MATLAB as a MEX-function. Additionally, some algorithmic improvements have been made.

- Fix: Boundary handling is now consistently half-sample symmetric in all operations
- New: Spatially-varying fidelity weight $\lambda(x)$
- New: DCT-based solver for faster deconvolution with symmetric kernels

1.2 Notation

All integrals "$\int_\Omega f(x)\,dx$" are over two-dimensional sets Ω with points x in $\mathbb{R}^2$, and dx is the usual Lebesgue measure on $\mathbb{R}^2$. We also sometimes refer to points with the notation (x, y) when it is needed to refer to their coordinates.

The symbol $*$ is used to denote two-dimensional convolution, defined as

$$(f * g)(x_0) := \int_{\mathbb{R}^2} f(x_0 - x)g(x)\,dx = \int_{\mathbb{R}^2} f(x)g(x_0 - x)\,dx.$$

If f is defined only on a rectangular subset of $\mathbb{R}^2$, then f is first extrapolated to $\mathbb{R}^2$ by its symmetric even extension (similarly for g).

2 Command line program usage

The following describes how to perform image restoration and segmentation using the command line programs `tvrestore` and `chanvese`. To obtain these programs, follow the C/C++ compiling instructions in section 7.

The basic syntax for these programs is

```
tvrestore  [param:value ...] ⟨input⟩ ⟨output⟩
chanvese   [param:value ...] ⟨input⟩ ⟨output⟩
```

where ⟨*input*⟩ and ⟨*output*⟩ specify the file names of the input image f and the output image u. These files should be Windows Bitmap BMP files, for example `noisy.bmp`. It is possible to compile the programs to include support for JPEG, PNG, and TIFF images (see section 7). If ⟨*output*⟩ is not specified, then u is written to `out.bmp`.

Text file format Some parameters require specifying a two-dimensional array. For these parameters, the array can be read from a text file or from the graylevel values of an image. To specify an array using a text file, the file should be ASCII text of numeric values delimited by whitespace and line breaks to indicate new rows. Blank lines or lines starting with # are ignored. For example, this file represents a 7×7 array.

```
# mykernel.txt: a 7x7 filter

0.0000    0.0002    0.0011    0.0018    0.0011    0.0002    0.0000
0.0002    0.0029    0.0131    0.0215    0.0131    0.0029    0.0002
0.0011    0.0131    0.0586    0.0965    0.0586    0.0131    0.0011
0.0018    0.0215    0.0965    0.1596    0.0965    0.0215    0.0018
0.0011    0.0131    0.0586    0.0965    0.0586    0.0131    0.0011
0.0002    0.0029    0.0131    0.0215    0.0131    0.0029    0.0002
0.0000    0.0002    0.0011    0.0018    0.0011    0.0002    0.0000
```

2.1 Image restoration

The command line program `tvrestore` performs TV-regularized image restoration by solving the minimization problem

$$\min_{u \in BV(\Omega)} \int_\Omega |\nabla u(x)| \, dx + \int_\Omega \lambda(x) F\big(Ku(x), f(x)\big) \, dx$$

including denoising, deconvolution, and inpainting as special cases. The important options are

`lambda:`$\langle$*number*$\rangle$	specify λ value
`K:`$\langle$*kernel*$\rangle$	specify the blur K for deconvolution
`D:`$\langle$*domain*$\rangle$	specify inpainting domain $\mathcal{D}$

These options are described in detail below.

Color If the input f is a color image, then TV is replaced with vectorial TV

$$\|u\|_{VTV} = \int_\Omega \Big(\sum_{i \in channels} |\nabla u_i(x)|^2 \Big)^{1/2} \, dx.$$

Compared to processing each channel independently, the advantage of vectorial TV is that it forces the channels to stay aligned, which prevents false color artifacts near edges.

Denoising The problem of image noise removal or *denoising* is, given a noisy image $f : \Omega \to \mathbb{R}$, to estimate the clean underlying image u. For (additive white) Gaussian noise, the degradation model describing the relationship between f and u is

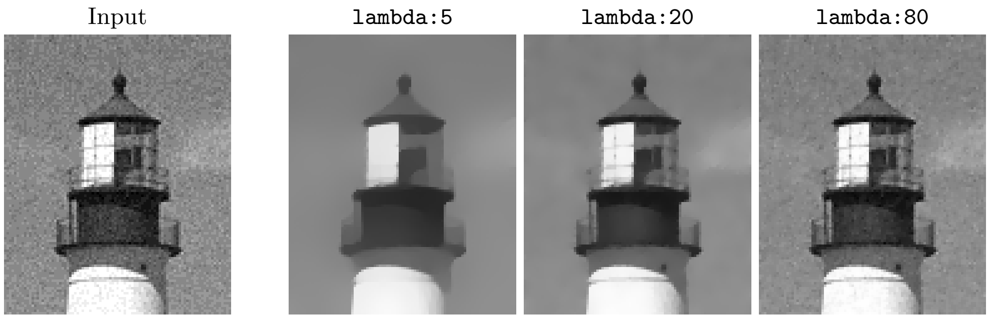
$$f = u + \eta,$$

where η is i.i.d. zero-mean Gaussian distributed.

The `tvrestore` program implements TV-regularized denoising:

```
tvrestore lambda:⟨number⟩ f.bmp u.bmp
```

The option `lambda:`⟨*number*⟩ is used to specify parameter λ, which is a positive value controlling the denoising strength. Smaller λ implies stronger denoising.



```
tvrestore lambda:5 lighthouse.bmp result1.bmp
tvrestore lambda:20 lighthouse.bmp result2.bmp
tvrestore lambda:80 lighthouse.bmp result3.bmp
```

The images above show a noisy input image f and the results of TV-regularized denoising with three different λ values. For this image, the right λ seems to be about 20.

Deblurring (deconvolution) The image *deblurring* problem is to recover u from a given blurry and noisy image f. For Gaussian noise, the degradation model is

$$f = Ku + \eta,$$

where K is the blur operator. For simplicity, the tvreg package is limited to the easier case of *deconvolution*, where $Ku = \varphi * u$, where φ is the blur point spread function. To deal with boundaries in the convolution, u is first extended to $\mathbb{R}^2$ by its symmetric even extension. It is assumed that φ is known or at least that a reasonable approximation is available. There are also "blind deconvolution" methods for the case where φ is unknown [12, 37], however, we do not discuss them further here.

Use the `K` option with `tvrestore` to specify the point spread function:

```
tvrestore lambda:⟨λ⟩ K:⟨kernel⟩ f.bmp u.bmp
```

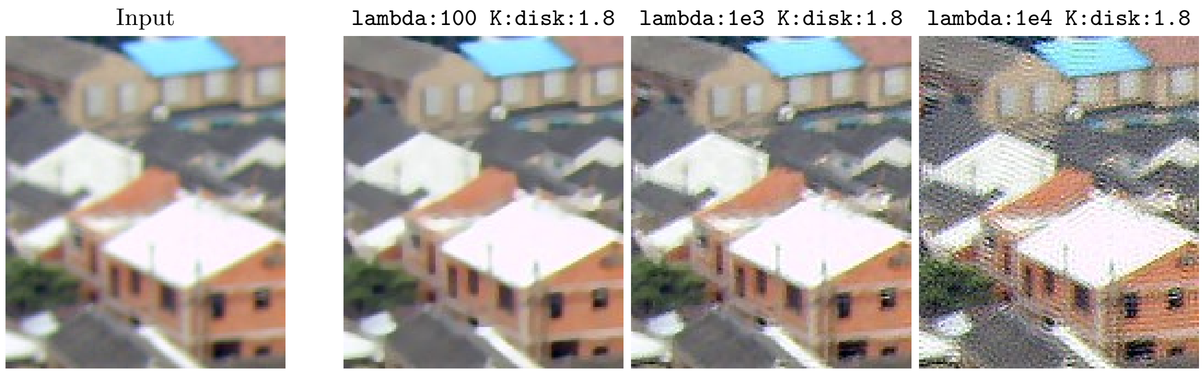
where ⟨*kernel*⟩ is one of the following

		Example
`disk:`⟨*radius*⟩	filled disk of radius ⟨*radius*⟩	`K:disk:3.1`
`gaussian:`⟨*sigma*⟩	Gaussian of parameter ⟨*sigma*⟩	`K:gaussian:0.85`
image file	read from an image	`K:mykernel.bmp`
text file	read from a text file	`K:mykernel.txt`

If the kernel is specified as an image, the kernel coefficients are derived from the grayscale pixel values and normalized to sum to one.

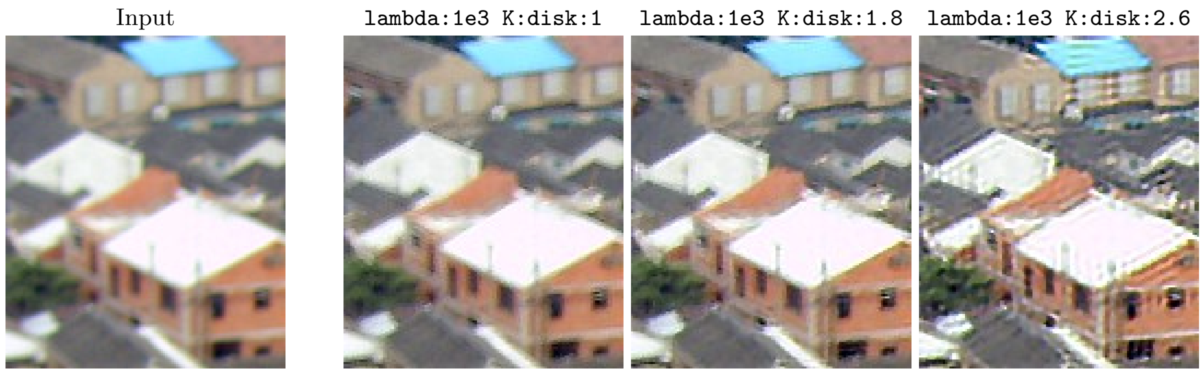
Alternatively, the kernel may be specified with a text file. This method allows for specifying coefficients with more precision and for specifying negative coefficient values. The text file format is described above. When using a text file, the coefficients *are not* normalized to sum to one, nor is it required that they do sum to one.

The program solves for $\mathtt{u}$ approximately equal to $\mathtt{f} * \varphi$. Parameter λ balances between deblurring accuracy and denoising, where smaller λ implies stronger denoising (but at the cost of deblurring accuracy).



```
tvrestore lambda:100 K:disk:1.8 blurry.bmp result1.bmp
tvrestore lambda:1e3 K:disk:1.8 blurry.bmp result2.bmp
tvrestore lambda:1e4 K:disk:1.8 blurry.bmp result3.bmp
```

The input image above is from a photograph that was taken out of focus. Restoration results are shown using a disk-shaped kernel of radius 1.8 for three different λ values. Notice that overshooting λ is much worse than undershooting.



```
tvrestore lambda:1e3 K:disk:1 blurry.bmp result1.bmp
tvrestore lambda:1e3 K:disk:1.8 blurry.bmp result2.bmp
tvrestore lambda:1e3 K:disk:2.6 blurry.bmp result3.bmp
```

In these images, the λ value is fixed at 10^3 and the kernel radius is varied. We learn another lesson here: overshooting the kernel radius is much worse than undershooting.

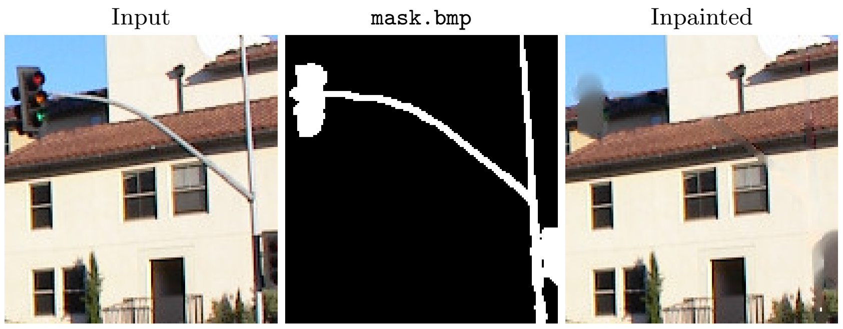
Inpainting In the image inpainting problem, the given image f is known only on $\Omega\backslash\mathcal{D}$, and the problem is to interpolate the unknown region $\mathcal{D}$.

Use the `D` option to specify the inpainting domain:

```
tvrestore lambda:⟨λ⟩ D:⟨domain⟩ f.bmp u.bmp
```

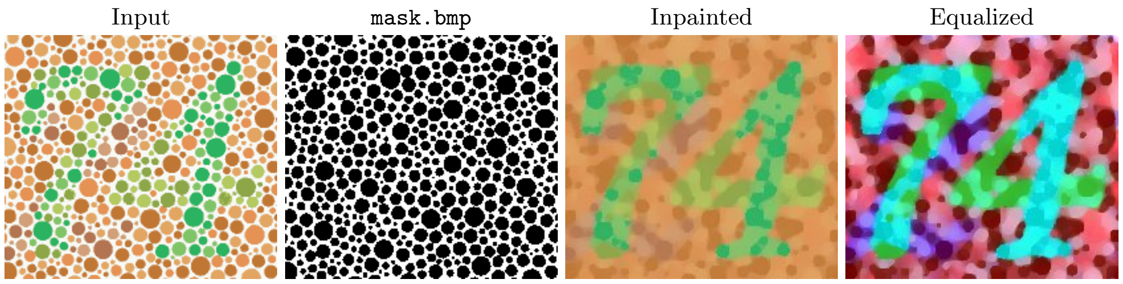
where ⟨*domain*⟩ is a text or image file describing an array of the same size as the f. The domain $\mathcal{D}$ is derived from the values that are greater than 0.5 (or if reading the domain from an image, pixels whiter than 50% graylevel). These pixels are considered as unknown in the inpainting problem.

Parameter `lambda` controls the denoising strength outside of the inpainting region, where smaller `lambda` implies stronger denoising. To keep pixels outside of the inpainting region approximately unchanged, set `lambda` to a large value.



```
tvrestore D:mask.bmp lambda:1e3 input.bmp inpainted.bmp
```

Inpainting is more effective on domains that are thin, like on wires or text. Results are usually poor on domains with large diameter. In the example above, I attempted to use inpainting to remove a traffic signal. The result is reasonable (although yet unnatural) over the pole mount, but much worse on the signal itself.



```
convert -threshold 75% ishihara.bmp mask.bmp
tvrestore D:mask.bmp lambda:10 ishihara.bmp inpainted.bmp
convert -equalize inpainted.bmp enhanced.bmp
```

Another application for inpainting is passing Ishihara color blindness tests. This particular image was designed so that the number 74 is visible to those with normal color vision while a dichromat may see the number 21. We apply a 75% graylevel threshold to find the white gaps between the dots to form an inpainting domain, then use `tvrestore` to perform inpainting. Histogram equalization is applied in the final image to make colors more distinct.

Noise models Good results depend on using a noise model that accurately describes the noise in the given image. Three different noise models are supported:

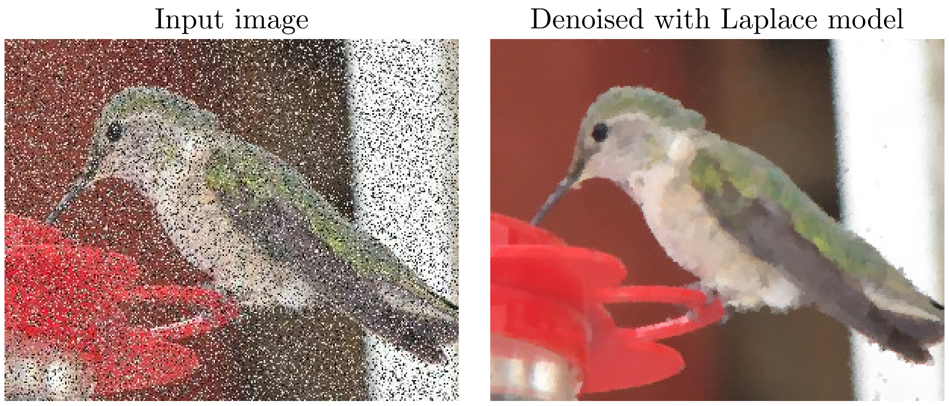
Gaussian	$P\big(f(x)\vert z(x)\big) = \frac{1}{Z}\exp\Big(-\frac{\big(z(x)-f(x)\big)^2}{2\sigma^2}\Big)$
Laplace	$P\big(f(x)\vert z(x)\big) = \frac{1}{Z}\exp\Big(-\frac{\vert z(x)-f(x)\vert}{2\sigma^2}\Big)$
Poisson	$P\big(f(x)\vert z(x)\big) = \frac{1}{Z}\exp\big(-z(x)\big)z(x)^{f(x)}$

where $z = u$ for denoising and $z = Ku$ for deconvolution, and $\frac{1}{Z}$ is the normalization such that densities sum to one. The original Gaussian (L^2) model was introduced by Rudin, Osher and Fatemi [25], the Laplace (L^1) model was developed by Chan and Esedoglu [10], and the Poisson model was developed by Le et al. [21]. These three noise models all lead to convex minimization problems.

The noise model is specified with option `noise`:

`noise:gaussian` or `noise:l2`	Gaussian noise model (default)
`noise:laplace` or `noise:l1`	Laplace noise model
`noise:poisson`	Poisson noise model

The Gaussian model is a good default choice since it is often a reasonable approximation of the true noise distribution, the minimization model has reliable theoretical properties, and is computationally most efficient in implementation. The Laplace and Poisson models are more effective in certain specialized applications. The Laplace model is better for fat tail noise distributions like salt-and-pepper and dark shot noise. The Poisson model describes low-light image acquisition and also is a rough approximation of multiplicative noise.



Spatially-varying fidelity weight The fidelity weight λ controls the amount of denoising. This parameter must be tuned for good results, since choosing a large λ removes a limited amount of noise while a small λ removes more noise but smooths out the signal. If the problem involves a blur kernel, λ also influences the deconvolution: large λ enforces stronger deconvolution ($Ku \approx f$) with weaker denoising while small λ has weaker deconvolution with stronger denoising.

In many cases, tuning λ as a constant parameter is enough to find an reasonable result. However, for some applications (perhaps in special experiments) you may want to apply a different fidelity weight λ in different parts of the image. A spatially-varying $\lambda(x)$ may be specified as

		Example
`lambda:`⟨*file*⟩	read from a file	`lambda:mylambda.txt`
`lambda:`⟨*scalefactor*⟩`:`⟨*file*⟩	read from a file and scale	`lambda:2.5:mylambda.txt`
		`lambda:100:mylambda.bmp`

The array described by the file must have the same number of rows and columns as the input image. The array values may be scaled by a constant factor with `lambda:`⟨*scalefactor*⟩`:`⟨*file*⟩.

When specifying $\lambda(x)$ from an image, the array values are read from the image graylevel intensities with black as 0 and white as 1. To create $\lambda(x)$ values greater than 1, use the syntax `lambda:`⟨*scalefactor*⟩`:`⟨*file*⟩ to scale $\lambda(x)$ by a constant factor.

Additional parameters Optionally, these additional parameters may be set to fine tune the minimization algorithm:

`tol:`⟨*number*⟩	specify convergence tolerance
`maxiter:`⟨*number*⟩	specify maximum number of iterations
`gamma1:`⟨*number*⟩	specify γ_1 the constraint weight for $\vec{d} = \nabla u$
`gamma2:`⟨*number*⟩	specify γ_1 the constraint weight for $z = Ku$

These options do not change the minimization problem, they only affect the speed and accuracy of the algorithm. By default, the tolerance is 10^{-3}, the maximum iterations is 50, and $\gamma_1 = 5$, $\gamma_2 = 8$.

2.2 Image segmentation

Given a grayscale image f, the `chanvese` command line program uses the Chan-Vese "active contours without edges" segmentation method [8] to segment the image into two regions. This is done by finding a local minimizer of

$$\begin{aligned}\min_{c_1,c_2,C}\ &\mu\,\text{Length}(C) + \nu\,\text{Area}(C)\\ &+ \lambda_1 \int_{inside(C)} \bigl(f(x) - c_1\bigr)^2\,dx + \lambda_2 \int_{outside(C)} \bigl(f(x) - c_2\bigr)^2\,dx.\end{aligned}$$

The goal is to find scalars c_1, c_2 and a closed curve C that minimize this functional. The method can be understood as finding a good approximation to f of the form

$$u(x) = \begin{cases} c_1 & \text{if } x \text{ is inside } C,\\ c_2 & \text{if } x \text{ is outside } C.\end{cases}$$

The parameter $\mu \geq 0$ enforces smoothness on the curve by penalizing its length. Parameter ν may be either sign and penalizes (if $\nu > 0$) or rewards (if $\nu < 0$) the area inside the curve. Parameters $\lambda_1 > 0$ and $\lambda_2 > 0$ penalize the approximation error inside and outside of the curve.

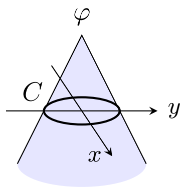
Chan-Vese is a *two-phase* method, meaning it segments the image into two regions (inside of C and outside of C). There are also *multiphase* segmentation methods that segment the image into more than two parts, but we do not investigate them here.

Level set functions Instead of representing the curve C explicitly, it is represented as the zero-crossing of a *level set function* φ, by the relationship $C = \{(x, y) : \varphi(x, y) = 0\}$. With this representation, C is a closed curve and it splits space into two regions according to the sign of φ:

$$\begin{cases} \varphi(x,y) > 0 & (x,y) \text{ is inside of } C, \\ \varphi(x,y) < 0 & (x,y) \text{ is outside of } C. \end{cases}$$

We will refer to the region where φ is positive as the "inside" of C, but this is arbitrary since $-\varphi$ represents the same C. As an example, the level set function

$$\varphi(x,y) = r - \sqrt{x^2 + y^2}$$



represents a circle of radius r.

Color If f is a color image, then the segmentation is performed with the Chan-Sandberg-Vese vectorial extension [6],

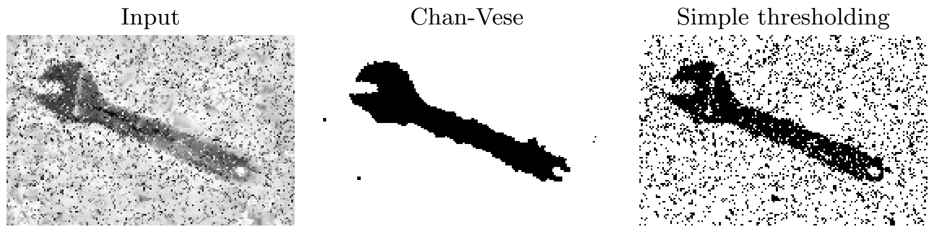
$$\min_{c_1,c_2,C} \mu \operatorname{Length}(C) + \nu \operatorname{Area}(C) + \lambda_1 \int_{inside(C)} \|f(x) - c_1\|_2^2 \, dx + \lambda_2 \int_{outside(C)} \|f(x) - c_2\|_2^2 \, dx.$$

Program options The command line options for the `chanvese` program are

`mu:`⟨*number*⟩	length penalty μ (default 0.25)
`nu:`⟨*number*⟩	area penalty ν (default 0.0)
`lambda1:`⟨*number*⟩	fit weight inside the curve λ_1 (default 1.0)
`lambda2:`⟨*number*⟩	fit weight outside the curve λ_2 (default 1.0)
`phi0:`⟨*file*⟩	read initial level set from an image or text file
`tol:`⟨*number*⟩	convergence tolerance (default 10^{-4})
`maxiter:`⟨*number*⟩	maximum number of iterations (default 500)
`dt:`⟨*number*⟩	time step (default 0.5)
`display:`⟨*style*⟩	display style of the output segmentation (explained below)

The most important parameter is the length penalty μ. Larger μ assigns more penalty to curve length and produces a smoother segmentation curve. The parameter must be balanced between ignoring noise and capturing meaningful detail.

Unfortunately, the implemented stopping condition is not always effective, it is fooled into stopping prematurely if the evolution is slow while in other cases not stopping until the maximum iteration limit. To override the stopping condition, set `tol:0` and `maxiter:`⟨*number*⟩ to specify a fixed number of iterations to run.



```
chanvese mu:0.2 display:binary wrench.bmp chanvese.bmp
convert -threshold 65% wrench.bmp thresh.bmp
```

Chan-Vese finds a wrench in this noisy example image. For comparison, segmentation by simple graylevel thresholding is also shown.

Initialization By default, the segmentation is initialized with the level set function

$$\varphi(x,y) = \sin \tfrac{\pi}{5}x \, \sin \tfrac{\pi}{5}y.$$

which defines the initial segmentation as a checkerboard shape. Such an initialization has been observed to have faster convergence than with more regular level set functions. When using the default initialization, the meaning of the result's "inside" vs. "outside" is arbitrary, since φ and $-\varphi$ represent the same curve.

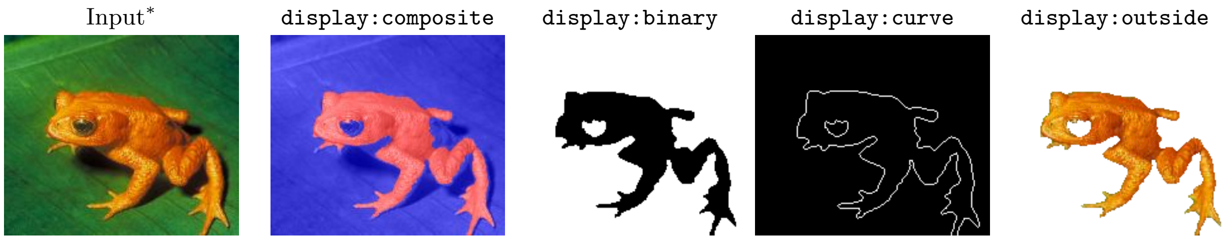
It is important to note that Chan-Vese finds a *local* minimizer, not necessarily a global one, which can be practically useful. Different local minimizers may identify different objects in the image. If an approximate segmentation of the object of interest is known (or just its rough location), then this information can be used to make a better initialization.

An initial level set function can be specified as `phi0:`⟨*file*⟩, where ⟨*file*⟩ is either a text file or image file. When using an image file, the values are read from the graylevel intensities with black as -1 and white as $+1$. Using a specific initialization can help to capture a particular object in the image and also gives meaning to the curve's "inside" vs. "outside."

Display style The segmentation output can be displayed in several different ways.

`display:composite`	as a colorful overlay composited with the input (default)
`display:binary`	as a binary image where white = inside and black = outside
`display:curve`	as a binary image of only the curve
`display:inside`	as a cutout of the input image inside the curve
`display:outside`	as a cutout of the input image outside the curve

The `composite` style is the default, which shows a grayscale version of the input with the segmentation indicated with blue (inside) and red (outside). One of the other styles may be more useful depending on the application, for example, a `binary` or `curve` segmentation output can be easily read as an input to another program.



```
chanvese display:composite toad.bmp result1.bmp
chanvese display:binary toad.bmp result2.bmp
chanvese display:curve toad.bmp result3.bmp
chanvese display:outside toad.bmp result4.bmp
```

3 Usage in C/C++

3.1 Image restoration

To use tvreg image restoration from C/C++ programs, copy the files `tvreg.h`, `tvreg.c`, and `num.h` into your project and #include the `tvreg.h` header file. You will also need to configure your project to link with the FFTW library.

The file `num.h` defines a typedef `num`, which you may configure as either double or float. By default it is double. To use float, add the statement `#define NUM_SINGLE` before including `tvreg.h` (or add `-DNUM_SINGLE` to the project compile flags). The corresponding version of the FFTW library (libfftw3 or libfftw3f) is needed to link the program. See section 7 for help on compiling.

The main computation routine is `TvRestore`, and options are set by creating a `tvregopt` object. For example, the following configures and runs TV-regularized grayscale deconvolution with the Laplace noise model:

```
num Kernel[9] = {0.000, 0.125, 0.000,
                 0.125, 0.500, 0.125,
                 0.000, 0.125, 0.000};
tvregopt *Opt = TvRegNewOpt();          /* Create a tvregopt object   */

TvRegSetLambda(20);                     /* Set lambda = 20            */
TvRegSetKernel(Opt, Kernel, 3, 3);      /* Set kernel of size 3x3     */
TvRegSetNoiseModel(Opt, "Laplace");     /* Set Laplace noise model    */

memcpy(u, f, sizeof(num)*Width*Height); /* Use u = f as initial guess */
TvRestore(u, f, Width, Height, 1, Opt); /* Run restoration            */

TvRegFreeOpt(Opt);                      /* Free tvregopt object       */
```

[*]Photograph by Charles H. Smith, U.S. Fish and Wildlife Service.

TvRestore

```
int TvRestore(num *u, const num *f, int Width, int Height, int NumChannels,
    const tvregopt *Opt)
```

Image `u` is both an input and output of the routine. Image `u` should be set by the caller to an initial guess, for example a reasonable generic initialization is to set `u` as a copy of `f`. Image `u` is overwritten with the restored image.

The input image `f` should be a contiguous array of size `Width`$\times$`Height`$\times$`NumChannels` in planar row-major order,

$$\texttt{f[}x\texttt{ + Width*(}y\texttt{ + Height*}k\texttt{)]} = k\text{th component of pixel } (x, y).$$

The image intensity values of `f` should be scaled so that the maximum intensity range of the true clean image is from 0 to 1. It is allowed that `f` have values outside of $[0, 1]$ (as spurious noisy pixels and the effects of blur may cause f to exceed this range), but it should be scaled so that the restored image is in $[0, 1]$. This scaling is especially important for the Poisson noise model.

Typically, `NumChannels` is either 1 (grayscale image) or 3 (color), but generally it may be any positive integer. If `NumChannels` > 1, then vectorial TV regularization is used in place of TV.

Other options are specified through `Opt`. If `Opt = NULL`, then `TvRestore` does denoising with the Gaussian noise model. But for practical use, you will probably want to call at least `TvRegSetLambda` or `TvRegSetVaryingLambda` to set the fidelity weight λ.

First use `tvregopt Opt = TvRegNewOpt()` to create a new options object with default options (denoising with the Gaussian noise model). Then use any of the following functions.

`TvRegSetLambda`	specify a constant fidelity weight λ
`TvRegSetVaryingLambda`	specify a spatially-varying fidelity weight $\lambda(x)$
`TvRegSetKernel`	kernel for deconvolution problems
`TvRegSetNoiseModel`	noise model
`TvRegSetTol`	convergence tolerance
`TvRegSetMaxIter`	maximum number of iterations
`TvRegSetGamma1`	constraint weight on $\vec{d} = \nabla u$
`TvRegSetGamma2`	constraint weight on $z = Ku$
`TvRegSetPlotFun`	custom plotting function

`TvRestore` does not change `Opt` or any memory attached to it. `Opt` can be reused for multiple restoration problems.

TvRegSetLambda

```
void TvRegSetLambda(tvregopt *Opt, num Lambda)
```

`TvRegSetLambda` specifies a constant fidelity weight λ. The syntax is similar for the other scalar parameters `TvRegSetTol`, `TvRegSetMaxIter`, `TvRegSetGamma1`, `TvRegSetGamma2`. Smaller `Lambda` implies stronger denoising. This setting is ignored if `TvRegSetVaryingLambda` is also set.

TvRegSetVaryingLambda

```
void TvRegSetVaryingLambda(tvregopt *Opt,
    const num *VaryingLambda, int LambdaWidth, int LambdaHeight)
```

`TvRegSetVaryingLambda` specifies a spatially-varying fidelity weight $\lambda(x)$. This can be used to apply different denoising strength over different parts of the image (for example, less denoising on textured areas). It can also be used for inpainting by setting

$$\texttt{VaryingLambda[}x\texttt{ + Width*}y\texttt{]} = \begin{cases} 0 & \text{if pixel } (x,y) \text{ is unknown,} \\ C & \text{if pixel } (x,y) \text{ is known,} \end{cases}$$

where C is any positive value. Pixels where `VaryingLambda` is zero are considered unknown and are inpainted. The other pixels are denoised (and deconvolved, if a kernel is also set). To inpaint but keep the known pixels approximately unchanged, set C to a large value.

If `TvRegSetVaryingLambda` is used to set a non-null `VaryingLambda`, then this overrides the constant λ setting made with `TvRegSetLambda`.

TvRegSetKernel

```
void TvRegSetKernel(tvregopt *Opt,
    const num *Kernel, int KernelWidth, int KernelHeight)
```

`TvRegSetKernel` specifies the kernel for a deconvolution problem. `Kernel` should be a contiguous array of size `KernelWidth`×`KernelHeight` in row-major order,

$$\texttt{Kernel[}x\texttt{ + KernelWidth*}y\texttt{]} = K(x,y).$$

If `Kernel = NULL`, then no deconvolution is performed.

TvRegSetNoiseModel

```
int TvRegSetNoiseModel(tvregopt *Opt, const char *NoiseModel)
```

`TvRegSetNoiseModel` specifies the noise model. `NoiseModel` should be a string specifying one of the following:

`"Gaussian"` or `"L2"`	(default) Additive white Gaussian noise (AWGN), this is the noise model used in the traditional Rudin-Osher-Fatemi model;
`"Laplace"` or `"L1"`	Laplace noise, effective for salt-and-pepper noise;
`"Poisson"`	Each pixel is an independent Poisson random variable with mean equal to the exact value.

TvRegSetPlotFun

```
void TvRegSetPlotFun(tvregopt *Opt,
    int (*PlotFun)(int, int, num, const num*, int, int, int, void*),
    void *PlotParam)
```

`TvRegSetPlotFun` specifies a plotting function. Setting a plotting function gives control over how `TvRestore` displays information. Setting `PlotFun = NULL` disables all normal display (error messages are still displayed). An example `PlotFun` is

```
int ExamplePlotFun(int State, int Iter, num Delta,
    const num *u, int Width, int Height, int NumChannels, void *PlotParam)
{
    switch(State)
    {
    case 0: /* Running */
        fprintf(stderr, " RUNNING   Iter=%4d, Delta=%7.4f\r", Iter, Delta);
        break;
    case 1: /* Converged successfully */
        fprintf(stderr, " CONVERGED Iter=%4d, Delta=%7.4f\n", Iter, Delta);
        break;
    case 2: /* Maximum iterations exceeded */
        fprintf(stderr, " Maximum number of iterations exceeded!\n");
        break;
    }
    return 1;
}
```

The `State` argument is either 0, 1, or 2, and indicates `TvRestore`'s status. `Iter` is the number of iterations completed, and `Delta` is the change in the solution

$$\texttt{Delta} = \|u^{\text{cur}} - u^{\text{prev}}\|_2 / \|f\|_2.$$

Argument `u` gives a pointer to the current solution, which can be used to plot an animated display of the solution progress. `PlotParam` is a void pointer that can be used to pass additional information to `PlotFun` if needed.

The plot function return value tells `TvRestore` whether to continue the computation. A nonzero value tells `TvRestore` to continue and zero value means stop.

TvRegPrintOpt

```
void TvRegPrintOpt(const tvregopt *Opt);
```

`TvRegPrintOpt` prints the current options to stdout. It also prints a brief description of the algorithm that `TvRestore` will use to solve the restoration problem. This is mostly for debugging purposes to verify that `TvRestore` will receive the expected options.

TvRegGetAlgorithm

```
const char *TvRegGetAlgorithm(const tvregopt *Opt)
```

`TvRegGetAlgorithm` returns a string containing a brief description of the algorithm that `TvRestore` will use to solve the restoration problem described by `Opt`.

3.2 Image segmentation

To use Chan-Vese image segmentation in C/C++ programs, copy the files `chanvese.h`, `chanvese.c`, and `num.h` into your project and `#include` the `chanvese.h` header file.

The file `num.h` defines a typedef `num`, which you may configure as either double or float. By default it is double. To use float, add the statement `#define NUM_SINGLE` before including `chanvese.h` (or add `-DNUM_SINGLE` to the project compile flags). The FFTW library is not required for Chan-Vese segmentation.

The main computation routine is `ChanVese`, and options are set by creating a `chanveseopt` object. The following runs Chan-Vese segmentation with length penalty $\mu = 0.8$, maximum iterations set to 100, and otherwise default arguments.

```
chanveseopt *Opt = ChanVeseNewOpt();      /* Create a new chanveseopt object */

ChanVeseSetMu(0.8);                       /* Set mu = 0.8                    */
ChanVeseSetMaxIter(100);                  /* Set maximum iterations to 100   */

ChanVeseInitPhi(Phi, Width, Height);      /* Default initialization for phi  */
ChanVese(Phi, f, Width, Height, 1, Opt);  /* Run segmentation                */

ChanVeseFreeOpt(Opt);                     /* Free chanveseopt object         */
```

ChanVeseInitPhi

```
void ChanVeseInitPhi(num *Phi, int Width, int Height)
```

`ChanVeseInitPhi` fills array `Phi` with the generic default starting function

$$\varphi(x, y) = \sin \tfrac{\pi}{5}x \, \sin \tfrac{\pi}{5}y.$$

If an approximate segmentation of the object of interest is known (or just its rough location), then this information can be used to make a better initialization than `ChanVeseInitPhi`.

ChanVese

```
void ChanVese(num *Phi, const num *f,
    int Width, int Height, int NumChannels, const chanveseopt *Opt)
```

`ChanVese` performs Chan-Vese two-phase segmentation on image `f` with options `Opt`. The image `f` should be a contiguous array of size `Width`×`Height`×`NumChannels` in planar row-major order,

$$\texttt{f[}x\texttt{ + Width*(}y\texttt{ + Height*}k\texttt{)]} = k\text{th component of pixel } (x, y).$$

Array `Phi` is both an input and output of the routine. It should be set by the caller to an array of size `Width`×`Height` representing the initial level set function. The initial segmentation is defined by the sign of `Phi`, where positive is inside the curve and negative is outside the curve. A reasonable generic initialization for `Phi` is provided with `ChanVeseInitPhi`. After the computation, `Phi` is overwritten with a level set function representing the final segmentation.

Other options are specified through `Opt`. If `Opt = NULL`, then `ChanVese` does segmentation with default parameters. For practical use, you will probably at least want to call `ChanVeseSetMu` to tune the length penalty.

First use `chanveseopt Opt = ChanVeseNewOpt()` to create a new options object with default options. Then use any of the following functions.

`void ChanVeseSetMu(chanveseopt *Opt, num Mu)`	length penalty μ
`void ChanVeseSetNu(chanveseopt *Opt, num Nu)`	area penalty ν
`void ChanVeseSetLambda1(chanveseopt *Opt, num Lambda1)`	fit weight inside the curve λ_1
`void ChanVeseSetLambda2(chanveseopt *Opt, num Lambda2)`	fit weight outside the curve λ_2
`void ChanVeseSetTol(chanveseopt *Opt, num Tol)`	convergence tolerance
`void ChanVeseSetMaxIter(chanveseopt *Opt, int MaxIter)`	maximum number of iterations
`void ChanVeseSetDt(chanveseopt *Opt, num dt)`	time step parameter dt

RegionAverages

```
void RegionAverages(num *c1, num *c2, const num *Phi, const num *f,
    int Width, int Height, int NumChannels)
```

`RegionAverages` computes the average values inside and outside of the curve described by `Phi`. Arguments `c1` and `c2` should be arrays with space for at least `NumChannels` elements. The function computes `c1[k]` as the average value of `f[`x` + Width*(`y` + Height*`k`)]` over the points where `Phi[`x` + Width*`y`]` ≥ 0 and `c2[k]` as the average value over points where `Phi[`x` + Width*`y`]` < 0.

ChanVeseSetPlotFun

```
void ChanVeseSetPlotFun(chanveseopt *Opt,
    int (*PlotFun)(int, int, num, const num*, const num*, const num*,
        int, int, int, void*), void *PlotParam)
```

For control over how `ChanVese` outputs information, a custom plotting function may be set with `ChanVeseSetPlotFun`. Setting `PlotFun = NULL` disables normal display. An example `PlotFun` is

```
int ExamplePlotFun(int State, int Iter, num Delta,
    const num *c1, const num *c2, const num *Phi,
    int Width, int Height, int NumChannels, void *PlotParam)
{
    switch(State)
    {
    case 0: /* Running */
        fprintf(stderr, " RUNNING   Iter=%4d, Delta=%7.4f\r", Iter, Delta);
        break;
    case 1: /* Converged successfully */
        fprintf(stderr, " CONVERGED Iter=%4d, Delta=%7.4f\n", Iter, Delta);
        break;
    case 2: /* Maximum iterations exceeded */
        fprintf(stderr, " Maximum number of iterations exceeded!\n");
        break;
    }
    return 1;
}
```

The `State` argument is either 0, 1, or 2, and indicates `TvRestore`'s status. `Iter` is the number of iterations completed, and `Delta` is the change in the solution defined in terms of the number of pixels where `Phi` has changed sign,

$$\texttt{Delta} = \frac{\#\{x : \varphi^{\text{cur}}(x)\varphi^{\text{prev}}(x) < 0\}}{\text{number of pixels}}.$$

Arguments `c1`, `c2`, `Phi` give pointers to the current solution. The last argument `PlotParam` is a void pointer that can be used to pass additional information to `PlotFun` if necessary.

The plot function return value tells `ChanVese` whether to continue the computation. A nonzero value tells `ChanVese` to continue and zero value means stop.

ChanVesePrintOpt

```
void ChanVesePrintOpt(const tvregopt *Opt);
```

`ChanVesePrintOpt` prints the current options to stdout. This is mostly for debugging purposes to verify that `ChanVese` will receive the expected options.

4 Usage in MATLAB

tvreg can be used to perform image restoration and segmentation directly in MATLAB. For better computational performance, the main computation functions can be compiled as MATLAB MEX functions, see section 7. However, in case compiling is not possible, (slower) pure M-code equivalents are also provided. MATLAB will use the MEX versions if they are compiled and automatically fall back to the M-code versions otherwise.

4.1 Image restoration

tvreg image restoration can performed in MATLAB with the following functions.

Denoising	`u = tvdenoise(f,lambda)`
Deconvolution	`u = tvdeconv(f,lambda,K)`
Inpainting	`u = tvinpaint(f,lambda,D)`

In these three functions, `f` is the input image, `lambda` is a positive parameter tuning the denoising strength, and `u` is the restored image.

Image `f` may be either an `M`×`N` matrix for a grayscale image or more generally an `M`×`N`×`P` array, where usually `P` = 3 for a color image, but `P` may be any positive integer for an arbitrary multichannel image. The image `f` should be scaled so that the maximum intensity range of the true clean image is from 0 to 1. It is allowed that `f` have values outside of $[0, 1]$ (the effects of noise and blur may cause `f` exceed this range), but it should be scaled so that the restored image is in $[0, 1]$ (and not $[0, 255]$). This scaling is especially important for the Poisson noise model.

The parameter `lambda` is a positive value specifying the fidelity weight where smaller `lambda` implies stronger denoising. A spatially-varying fidelity weight can be specified by setting `lambda` as an `M`×`N` matrix.

Denoising The function `u = tvdenoise(f,lambda)` performs TV-regularized denoising, where `f` is a noisy image and `lambda` is a parameter controlling the denoising strength as described above.

Deconvolution The function `u = tvdeconv(f,lambda,K)` performs TV-regularized deconvolution, where `K` should be a matrix representing the impulse response of the blur operation K. `tvdeconv` solves for `u` such that approximately $\mathtt{u} = \mathtt{K} * \mathtt{f}$.

```
K = gaussian(0.7);
K = K(:)*K(:)';
f = double(imread('turtles.png'))/255;
u = tvdeconv(f,500,K);
```

Beware that the coefficients of `K` are not automatically normalized to sum to one, nor is it required. Normalizing `K` can be done conveniently in MATLAB as `K = K/sum(K(:))`.

```
K = [0,1,0;
     1,4,1;
     0,1,0];
K = K/sum(K(:)); % Normalize to sum to one
u = tvdeconv(f,lambda,K);
```

Inpainting The function `u = tvinpaint(f,lambda,D)` performs TV-regularized inpainting, where `D` should be an `M`×`N` logical array specifying an inpainting domain. Pixels where `D` is true are considered unknown and are interpolated (inpainted). Pixels where `D` is false are denoised. To inpaint domain `D` but keep pixels outside of `D` approximately unchanged, set `lambda` to a large value.

```
f = double(imread('rabbit.png'))/255;
D = (imread('rabbit-D.png') ~= 0);
u = tvinpaint(f,1e4,D);
```

Since inpainting is actually a special case of spatially-varying fidelity weight, inpainting can be performed using `tvdenoise` with the arguments `tvdenoise(f,lambda*(~D))`.

Other parameters The Gaussian noise model is used by default. The noise model may be specified as

```
tvdenoise(...,model)
tvdeconv(...,model)
tvinpaint(...,model)
```

where `model` is a case-insensitive string equal to `'Gaussian'` (or `'L2'`), `'Laplace'` (or `'L1'`), or `'Poisson'`. Additionally, the tolerance and maximum number of iterations specified as

```
tvdenoise(...,model,tol,maxiter)
tvdeconv(...,model,tol,maxiter)
tvinpaint(...,model,tol,maxiter)
```

These options do not change the minimization problem, only the accuracy of the algorithm.

Two further parameters may be given to any of these functions:

`(...,model,tol,maxiter,plotfun,u0)`

Argument `plotfun` is a plot callback function to customize the display of the solution progress. Argument `u0` is the initial guess of the solution, which should have the same size as the input image. By default, `u0 = f` is used as the initial guess. A better initial guess may allow the solution to be found in fewer iterations.

For the plot callback, `plotfun` is the name of a function or a function handle. An example `plotfun` is

```
function myplot(state, iter, delta, u)
switch state
    case 0  % Running
        fprintf('  Iter=%4d  Delta=%7.3f\n', iter, delta);
    case 1  % Converged
        fprintf('Converged successfully.\n');
    case 2  % Maximum iterations exceeded
        fprintf('Maximum number of iterations exceeded.\n');
end

if size(u,3) == 3
    % Display color image
    image(min(max(u,0),1));
else
    % Display grayscale image
    image(u*255);
    colormap(gray(256));
end

axis image
axis off
title(sprintf('Iter=%d  Delta=%.4f', iter, delta));

shg;  % Force figure to foreground
```

Optionally, the plot function can return a scalar value to tell whether computation should continue. Returning a nonzero value tells the computation to continue and a zero value means stop.

4.2 Image segmentation

Chan-Vese two-phase segmentation can be performed in MATLAB with the `chanvese` function.

```
f = double(imread('toad.jpg'))/255;
phi = chanvese(f);
imagesc(phi > 0);
axis image
```

The `chanvese` function has a number of optional arguments which can be passed either as additional arguments or as an options struct.

`phi = chanvese(f,phi0,opt)` segments image `f` where `phi0` is the initial level set function, and `opt` is a struct containing all or any subset of the following parameters:

`opt.tol`	convergence tolerance (default 10^{-4})
`opt.maxiter`	maximum number of iterations (default 500)
`opt.mu`	length penalty parameter (default 0.25)
`opt.nu`	area penalty (default 0)
`opt.lambda1`	inside fit penalty (default 1)
`opt.lambda2`	outside fit penalty (default 1)
`opt.dt`	timestep parameter (default 0.5)
`opt.plotfun`	plotting function
`opt.verbose`	show verbose information

Unfortunately, the implemented stopping condition is not always effective, it is fooled into stopping prematurely if the evolution is slow while in other cases not stopping until the maximum iteration limit. To override the stopping condition, set `opt.tol` to 0 and use `opt.maxiter` to specify a fixed number of iterations.

`phi = chanvese(f,phi0,tol,maxiter,mu,nu,lambda1,lambda2,dt,plotfun)` is the alternative syntax. Passing empty set `[]` or omitting an argument specifies the default value. For example,

```
% Run 50 iterations with length penalty mu
phi = chanvese(f,[],0,50,mu);
```

The plot function has similar syntax as for the image restoration functions.

```
function myplot(state, iter, delta, phi)
switch state
    case 0  % Running
        fprintf('  Iter=%4d  Delta=%7.3f\n', iter, delta);
    case 1  % Converged
        fprintf('Converged successfully.\n');
    case 2  % Maximum iterations exceeded
        fprintf('Maximum number of iterations exceeded.\n');
end

imagesc(phi);
colormap(gray(256));
axis image
axis off
title(sprintf('Iter=%d  Delta=%.4f', iter, delta));

shg;  % Force figure to foreground
```

Optionally, the plot function can return a scalar value to tell whether `chanvese` should continue. Returning a nonzero value tells `chanvese` to continue and a zero value means stop.

5 Mathematical Background

denotes Laplacian, and $\|\cdot\|_p$ denotes the L^p norm on Ω. Variable x will be used to denote a point in two-dimensional space.

5.1 Image restoration

The image restoration methods in tvreg are based on the Rudin-Osher-Fatemi [25] total variation (TV) denoising technique, see for example Chan and Shen's book [11]. A general model for TV-regularized denoising, deblurring, and inpainting is to find an image u that minimizes

$$\min_{u\in BV(\Omega)} \int_\Omega |\nabla u(x)|\, dx + \int_\Omega \lambda(x) F\big(Ku(x), f(x)\big)\, dx, \tag{1}$$

where the integrals are over a two-dimensional bounded set $\Omega \subset \mathbb{R}^2$ and $|\nabla u(x)|$ denotes the gradient magnitude of u at $x \in \mathbb{R}^2$. Function f is the given noise and blur corrupted image, K is the blur operator, $\lambda(x)$ is a nonnegative function specifying the regularization strength, and F determines the type of data fidelity:

$$F\big(Ku(x), f(x)\big) = \begin{cases} \frac{1}{2}\big(Ku(x) - f(x)\big)^2 & \text{Gaussian noise,} \\ \big|Ku(x) - f(x)\big| & \text{Laplace noise,} \\ Ku(x) - f(x)\log Ku(x) & \text{Poisson.} \end{cases}$$

This general model can be used to perform image denoising, deconvolution, and inpainting as special cases. For simplicity, $\lambda(x)$ is usually specified as a positive constant, $\lambda(x) \equiv \lambda$. For inpainting problems, where f is considered unknown on a region $\mathcal{D} \subset \Omega$, the fidelity strength is

$$\lambda(x) = \begin{cases} 0 & \text{if } x \in \mathcal{D}, \\ C & \text{otherwise,} \end{cases}$$

where $C > 0$ is a constant parameter.

In other words, the TV-regularization strategy is to search over all possible functions[†] to find a function $u : \Omega \to \mathbb{R}$ that minimizes (1). In the discrete setting, the minimization has one dimension of freedom in each pixel of u. So even for a 256×256-pixel image, the minimization is over $256^2 = 65\,536$ dimensions. The search domain of the minimization is indeed impressively vast.

A technical remark: the gradient magnitude $|\nabla u|$ should actually be interpreted in a distributional sense; functions u with jump discontinuities are allowed. The total variation of an image is a Radon measure

$$\|u\|_{\mathrm{TV}} := \int_\Omega |Du| := \sup\Big\{ \int_\Omega u \operatorname{div} \vec{g}\, dx \;:\; \vec{g} \in C_c^1(\Omega, \mathbb{R}^2),\ \||\vec{g}|\|_\infty \le 1 \Big\}.$$

The supremum is over all vector fields $\vec{g}$ that are continuously differentiable, have support compactly contained in Ω, and have bounded magnitude $|\vec{g}(x)| \le 1$. When u is smooth, $\|u\|_{\mathrm{TV}} = \int_\Omega |\nabla u|\, dx$.

[†]More precisely, the minimization is over all functions of bounded variation $BV(\Omega)$.

Theoretical properties for grayscale restoration

- For denoising with the Gaussian model, if $f \in L^2$, then the minimizer u exists and is unique and is stable in L^2 with respect to perturbations in f [18].
- For denoising with the Laplace model, if $f \in L^1$, then minimizers u exist but are generally not unique and are not continuous with respect to f [10].
- For denoising with the Poisson model, if f is positive and bounded, then the minimizer u exists and is unique [21].
- For deblurring with the Gaussian model, if $f \in L^2$, the exact solution is in BV, and $K : L^1 \to L^2$ is bounded, injective, and satisfies $K[1] \equiv 1$, then the minimizer u exists and is unique [3].
- For inpainting with the Gaussian model, if the exact solution is in BV and takes values in $[0, 1]$, then minimizers u exist but are generally not unique [9].

Algorithms There are many algorithms for solving TV minimization problems, especially for denoising or inpainting with the Gaussian noise model. To name just a few, there are semi-implicit gradient descent [30], the discrete TV filter [7], Chambolle's dual algorithm [4], FTVd splitting with fixed-point continuation [34], graph cuts [5], and Haar frame shrinkage [27].

tvreg uses the split-Bregman method [17], which thanks to its operator splitting enable a modular approach to solving the problems. This is what allows tvreg to tackle such a wide variety of TV-regularized problems. Furthermore, split-Bregman has state-of-the-art competitive efficiency and reliability. I don't claim that tvreg/split-Bregman is the best possible method. The other methods listed above each have their advantages, and moreover TV minimization algorithms continue to be an active area of research.

5.2 Image segmentation

The Chan-Vese segmentation model [8] finds a local minimizer of

$$\begin{aligned} \min_{c_1,c_2,C} \; & \mu \,\mathrm{Length}(C) + \nu \,\mathrm{Area}(C) \\ & + \lambda_1 \int_{\mathit{inside}(C)} \bigl(f(x) - c_1\bigr)^2 \, dx + \lambda_2 \int_{\mathit{outside}(C)} \bigl(f(x) - c_2\bigr)^2 \, dx, \end{aligned}$$

where C is a closed curve and c_1 and c_2 are the average values inside and outside of C. The weights $\mu, \nu, \lambda_1, \lambda_2$ control respectively the penalties on the curve's length, area, and the fit inside and outside of C. The Chan-Vese model is a simplification of the Mumford-Shah segmentation model [23].

The problem is solved by representing C with a level set function φ through the relationship $C = \{x : \varphi(x) = 0\}$ (see section 2.2). By defining the Heaviside function and the Dirac measure,

$$H(t) = \begin{cases} 1 & \text{if } t \geq 0, \\ 0 & \text{if } t < 0, \end{cases} \qquad \delta(t) = \frac{d}{dt} H(t),$$

the curve's length and enclosed area can be expressed as

$$\operatorname{Area}(\varphi \geq 0) = \int_\Omega H\bigl(\varphi(x)\bigr)\,dx, \qquad \operatorname{Length}(\varphi = 0) = \int_\Omega \delta\bigl(\varphi(x)\bigr)\,dx.$$

This allows the minimization to be rewritten in terms of the level set function,

$$\begin{aligned}\min_{c_1,c_2,\varphi}\ &\mu \int_\Omega \delta\bigl(\varphi(x)\bigr)\,dx + \nu \int_\Omega H\bigl(\varphi(x)\bigr)\,dx \\ &+ \lambda_1 \int_\Omega H\bigl(\varphi(x)\bigr)\bigl(f(x) - c_1\bigr)^2\,dx + \lambda_2 \int_\Omega \bigl[1 - H\bigl(\varphi(x)\bigr)\bigr]\bigl(f(x) - c_2\bigr)^2\,dx.\end{aligned}$$

As developed in [8], the problem is then to find a locally optimal φ, which can be done by evolving a PDE on φ until reaching steady state. The segmentation is obtained from the sign of the resulting φ as $\{x : \varphi(x) > 0\}$ and $\{x : \varphi(x) < 0\}$.

5.3 The state of the art

TV regularization is a flexible technique. Over the past two decades, it has led to successful methods for image denoising, deblurring, inpainting, and a variety of other applications. Its strengths are a well developed theoretical understanding and a number of fast algorithms.

There are recent works in particular applications that surpass TV regularization. This list is not exhaustive, but a few highlights of suggested reading.

- **Tensor-driven diffusion.** The image structure tensor is defined at every point by the 2×2 matrix formed from the outer product $\nabla u \otimes \nabla u$. The structure tensor can be used to obtain robust estimates of the local edge orientation. Weickert [35] and Tschumperlé [31, 32] developed methods for inpainting, denoising, and other problems using diffusion processes that are steered by the structure tensor.

- **Patch-based processing and self-similarity.** Efros and Leung [15] proposed a patch-based inpainting method capable of texture synthesis. This work inspired the nonlocal means method [2], a simple and surprisingly effective denoising method based on patch similarity. Nonlocal means has been extended to apply in other inverse problems, see for example the frameworks proposed [16] and [24].

- **Sparsity.** Fueled by the interest in compressed sensing, sparse optimization has been found successful for inverse problems in imaging. Block-matching and 3D filtering (BM3D) [14] is a highly effective method for denoising. In an exciting recent work, Yu, Sapiro, and Mallat [38] proposed a framework based on a union of subspaces model of sparsity to develop computationally efficient methods for denoising and other problems.

Regarding Chan-Vese segmentation, its strength is its simplicity. The idea and level set based approach can be extended to for example segmentation based on texture cues [26], nested segmentation curves [13], and multiphase segmentation [20]. An improvement to Chan-Vese is to minimize under a Sobolev gradient instead of the L^2 gradient, leading to a PDE with better stability properties [19, 29].

To mention just a couple other works, segmentation by weighted aggregation (SWA) [28] is a flexible method using a multiscale approach and Boykov and Jolly proposed an interactive segmentation method using graph cuts [1]. See also Unnikrishnan et al. [33], which develops a methodology toward the objective evaluation of segmentation methods.

6 Solution with Split Bregman

Methods for efficiently solving TV regularized minimizations is a topic of ongoing research. The implementation in this package uses the recent *split Bregman* method of Goldstein and Osher [17]. At the time of this writing, split Bregman is one of the fastest methods for TV minimization, and it is one of the few methods with the flexibility to handle all of our problem parameters and noise models in a unified approach.

The split Bregman method solves a minimization problem by operator splitting and then applying Bregman iteration to solve the split problem. For (1), the split problem is

$$\begin{aligned} &\min_{\vec{d},z,u} \int_\Omega \big|\vec{d}(x)\big|\,dx + \int_\Omega \lambda(x) F\big(z(x), f(x)\big)\,dx \\ &\text{subject to } \vec{d} = \nabla u,\ z = Ku \end{aligned}$$

At first glance, it may appear that the split problem is no different from the original (1). The point is that the two terms of the objective have been split: the first term $\int |\vec{d}|$ only depends directly on $\vec{d}$ and the second term $\int \lambda F(z,f)$ only on z. Of course, $\vec{d}$ and z are still indirectly related through the constraints $\vec{d} = \nabla u$, $z = Ku$.

Bregman iteration is used to solve the split problem. In each iteration, Bregman iteration calls for the solution of the following problem:

$$\begin{aligned} &\min_{\vec{d},z,u} \int_\Omega |\vec{d}|\,dx + \int_\Omega \lambda F(z,f)\,dx \\ &+\frac{\gamma_1}{2}\|\vec{d} - \nabla u - \vec{b}_1\|_2^2 + \frac{\gamma_2}{2}\,\|z - Ku - b_2\|_2^2 \end{aligned} \tag{2}$$

where the additional terms are quadratic penalties enforcing the constraints and $\vec{b}_1$ and b_2 are variables related to the Bregman iteration algorithm.

The solution of (2), which minimizes jointly over $\vec{d}$, z, u, is approximated by alternatingly minimizing one variable at a time, that is, fixing z and u and minimizing over $\vec{d}$, then fixing $\vec{d}$ and u and minimizing over z, and so on. This leads to three variable subproblems.

The $\vec{d}$ subproblem Variables z and u are fixed, and the subproblem is

$$\min_{\vec{d}} \int |\vec{d}|\,dx + \frac{\gamma_1}{2}\|\vec{d} - \nabla u - \vec{b}_1\|_2^2.$$

Its solution decouples over x and is known in closed form:

$$\vec{d}(x) = \frac{\nabla u(x) + \vec{b}_1(x)}{|\nabla u(x) + \vec{b}_1(x)|} \max\big\{|\nabla u(x) + \vec{b}_1(x)| - 1/\gamma_1, 0\big\}.$$

This is the key subproblem that drives the TV minimization.

The z subproblem Variables $\vec{d}$ and u are fixed, and the subproblem is

$$\min_z \int_\Omega \lambda F(z,f)\,dx + \frac{\gamma_2}{2}\left\|z - Ku - b_2\right\|_2^2.$$

The solution decouples over x. The optimal z satisfies

$$\lambda \partial_z F(z,f) + \gamma_2(z - Ku - b_2) = 0.$$

We solve this equation for z for each noise model F.

- For Gaussian noise with $F(z,f) = \frac{1}{2}(z-f)^2$,

$$z(x) = \frac{Ku(x) + b_2(x) + \frac{1}{\gamma_2}\lambda(x)f(x)}{1 + \frac{1}{\gamma_2}\lambda(x)}.$$

- For Laplacian noise with $F(z,f) = |z-f|$,

$$\begin{aligned} z(x) &= f(x) + \frac{s(x)}{|s(x)|}\max\bigl\{|s(x)| - \tfrac{1}{\gamma_2}\lambda(x), 0\bigr\}, \\ s &= Ku - f + b_2. \end{aligned}$$

- For the Poisson model with $F(z,f) = z - f\log z$,

$$\begin{aligned} z(x) &= s(x)/2 + \sqrt{\bigl(s(x)/2\bigr)^2 + \tfrac{1}{\gamma_2}\lambda(x)f(x)}, \\ s &= Ku - \tfrac{1}{\gamma_2}\lambda + b_2. \end{aligned}$$

In the special case that $\lambda(x) = 0$ (i.e., when the problem has an inpainting domain and x is in the unknown region), the solution reduces to

$$z(x) = Ku(x) + b_2(x).$$

The u subproblem Variables $\vec{d}$ and z are fixed, and the subproblem is

$$\min_u \frac{\gamma_1}{2}\|\nabla u - \vec{d} + \vec{b}_1\|_2^2 + \frac{\gamma_2}{2}\left\|Ku - z + b_2\right\|_2^2.$$

For denoising and inpainting, K is identity and the optimal u satisfies

$$\tfrac{\gamma_2}{\gamma_1}u - \Delta u = \tfrac{\gamma_2}{\gamma_1}(z - b_2) - \operatorname{div}(\vec{d} - \vec{b}_1),$$

which is a sparse, symmetric positive definite linear system. The solution u can be efficiently approximated by Gauss-Seidel iteration.

For general K, the optimal u satisfies

$$(\tfrac{\gamma_2}{\gamma_1}K^*K - \Delta)u = \tfrac{\gamma_2}{\gamma_1}K^*(z - b_2) - \operatorname{div}(\vec{d} - \vec{b}_1), \tag{3}$$

where K^* is the adjoint of K. The optimality equation is symmetric positive definite and may be solved for example by preconditioned conjugate gradients. In the special case that Ku is a convolution, $Ku := \varphi * u$, the equation is efficiently solved in the Fourier domain:

$$\hat{u} = \frac{\frac{\gamma_2}{\gamma_1}\bar{\hat{\varphi}} \cdot (z - b_2)\hat{} - \big(\operatorname{div}(\vec{d} - \vec{b}_1)\big)\hat{}}{\frac{\gamma_2}{\gamma_1}\bar{\hat{\varphi}} \cdot \hat{\varphi} - \hat{\Delta}},$$

where $\hat{}$ denotes Fourier transform and $\cdot$ is pointwise multiplication. To avoid boundary artifacts, the image should first be doubled along each dimension with its symmetric extension.

If φ is symmetric in both dimensions, the computation can be done more efficiently with discrete cosine (DCT) transform. As developed by Martucci [22], convolution with symmetric boundary handling can be done as

$$\varphi * f = \mathcal{C}_{2e}^{-1}\big(\mathcal{C}_{1e}(\varphi) \cdot \mathcal{C}_{2e}(f)\big),$$

where $\mathcal{C}_{1e}$ and $\mathcal{C}_{2e}$ are the DCT-I and DCT-II transforms of the same period length. In this way, the data does not need to be padded; symmetric boundaries are implied by the transforms. Noting also that the transformed data is real, the memory cost with the DCT is several times lower than with the Fourier transform. So if φ is even in each dimension, a computationally efficient strategy to obtain u is

$$u = \mathcal{C}_{2e}^{-1}\left[\frac{\mathcal{C}_{2e}\big(\frac{\gamma_2}{\gamma_1}\varphi * (z - b_2) - \operatorname{div}(\vec{d} - \vec{b}_1)\big)}{\mathcal{C}_{1e}\big(\frac{\gamma_2}{\gamma_1}\varphi * \varphi - \Delta\big)}\right].$$

In tvreg, φ is tested for symmetry so that DCT can be used instead of Fourier when possible.

The full algorithm The minimization (1) is solved with the following iteration:

```
Initialize $u = z = b_2 = 0, \vec{d} = \vec{b}_1 = 0$
while "not converged"
    Solve the $u$ subproblem
    Solve the $\vec{d}$ subproblem                    (4)
    Solve the $z$ subproblem
    $\vec{b}_1 := \vec{b}_1 + \nabla u - \vec{d}$
    $b_2 := b_2 + Ku - z$
```

When solving the subproblems, the xth subproblem solution is computed from the current values of all other variables and overwrites the previous value of variable x. Convergence may be checked for example by testing the maximum difference from the previous iterate: $\|u^{\text{cur}} - u^{\text{prev}}\|_2 < \mathit{Tol}$.

Simplified algorithm The problem simplifies somewhat with the Gaussian noise model $F(z, f) = \frac{1}{2}(z - f)^2$. In this case, the z auxiliary variable is unnecessary, reducing the problem to

$$\min_{\vec{d},u} \int_\Omega |\vec{d}|\, dx + \lambda \int_\Omega \tfrac{1}{2}(Ku - f)^2\, dx + \frac{\gamma_1}{2}\|\vec{d} - \nabla u - \vec{b}_1\|_2^2.$$

With this simplification there are only two variable subproblems. The $\vec{d}$ subproblem is the same as previously. The u subproblem is

$$\min_u \lambda \int_\Omega \tfrac{1}{2}(Ku - f)^2\,dx + \frac{\gamma_1}{2}\|\nabla u - \vec{d} + \vec{b}_1\|_2^2.$$

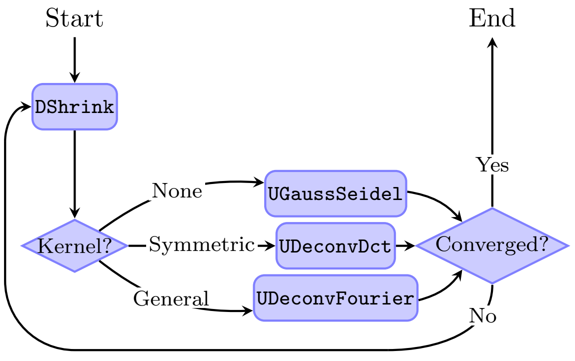
The optimality equation is

$$(\tfrac{\lambda}{\gamma_1}K^*K - \Delta)u = \tfrac{\lambda}{\gamma_1}K^*f - \operatorname{div}(\vec{d} - \vec{b}_1).$$

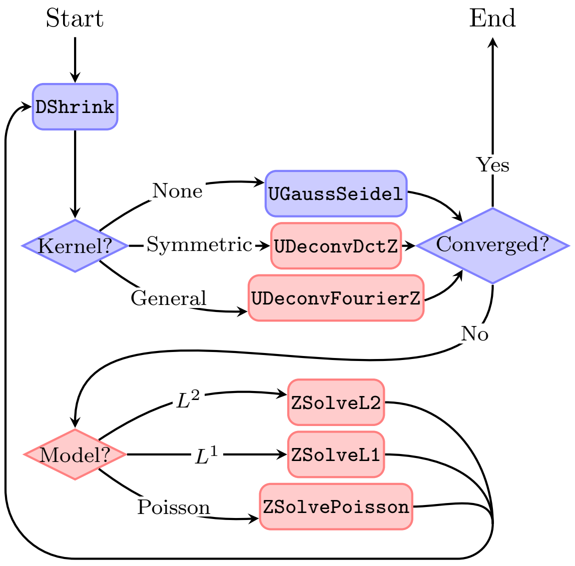
Depending on K, the optimality equation may be solved in the Fourier domain or by iterative matrix techniques. The split Bregman algorithm is the same as previously (4) but without the steps "Solve the z subproblem" and "$b_2 := b_2 + Ku - z$."

Implementation Details In the implementation, the variables $\tilde{\vec{d}} := \vec{d} - \vec{b}_1$ and $\tilde{z} := z - b_2$ are used instead of $\vec{b}_1$, b_2. The main computation routine is `TvRestore`, which runs the Bregman iteration and calls subroutines that solve the variable subproblems. The diagrams below illustrate the main loop.

Simplified algorithm (`UseZ = 0`)



General algorithm (`UseZ = 1`)



7 Compiling Instructions

The compilation is configurable. At a minimum, the FFTW library is needed to compile. Optionally, to read and write image formats other than BMP, the programs can be compiled with libraries for JPEG, PNG, and TIFF support.

Format	Library	Add preprocessor flag
JPEG	libjpeg	`LIBJPEG_SUPPORT`
PNG	libpng	`LIBPNG_SUPPORT`
TIFF	libtiff	`LIBTIFF_SUPPORT`
BMP	(native)	—

Also, tvreg can be compiled to use single precision (float) instead of double precision by adding the preprocessor flag `NUM_SINGLE`.

7.1 C/C++ on Linux or Mac OSX

On Ubuntu and other Debian-based Linux systems, the libraries can be installed by running the following line in a terminal:

```
    sudo apt-get install build-essential libfftw3-dev libjpeg8-dev libpng3-dev libtiff4-dev
```

On Fedora:

```
    sudo yum install gcc fftw-devel libjpeg-devel libpng-devel libtiff-devel
```

On Mac OSX, the libraries can be installed with Fink:

```
    sudo fink install fftw libjpeg libpng libtiff
```

Once the libraries are installed, use the GCC makefile to compile the programs. Open a terminal in the directory of the tvreg sources and run the following line:

```
    make -f makefile.gcc
```

This should produce two executables `tvrestore` and `chanvese`.

Troubleshooting The GCC makefile will try to use libjpeg, libpng, and libtiff. If linking with these libraries is a problem, they can be disabled by commenting their line at the top of the makefile.

```
# The following three statements determine the build configuration.
# For handling different image formats, the program can be linked with
# the libjpeg, libpng, and libtiff libraries.  For each library, set
# the flags needed for linking.  To disable use of a library, comment
# its statement.  You can disable all three (BMP is always supported).
LDLIBJPEG=-ljpeg
LDLIBPNG=-lpng
LDLIBTIFF=-ltiff
```

The makefile will automatically set the corresponding preprocessor symbols; only these lines need to be changed. For example, to disable libjpeg and libtiff but to keep libpng support, comment the first and third lines

```
#LDLIBJPEG=-ljpeg
LDLIBPNG=-lpng
#LDLIBTIFF=-ltiff
```

While libjpeg, libpng, and libtiff are optional, the FFTW library is required.

7.2 C/C++ on Windows

The code can be compiled using Microsoft Visual C++ (MSVC). Microsoft Visual Studio Express

`http://www.microsoft.com/express/windows`

can be downloaded for free.

FFTW The FFTW library is required to compile. To obtain FFTW, download the precompiled Windows DLL files from

`http://www.fftw.org/install/windows.html`

In order for MSVC to link with FFTW, we must create LIB import libraries. Open a Visual Studio Command Prompt by selecting Start Menu → Programs → Microsoft Visual Studio → Visual Studio Tools → Visual Studio Command Prompt. (Alternatively, open a regular command prompt and run `vcvarsall.bat`.) Then, navigate to the FFTW DLL files and run the commands

```
lib /def:libfftw3-3.def
lib /def:libfftw3f-3.def
lib /def:libfftw3l-3.def
```

This should produce the LIB files `libfftw3-3.lib`, `libfftw3f-3.lib`, and `libfftw3l-3.lib`.

Building tvreg To build tvreg, perform the following steps.

1. Copy the FFTW DLL files to the tvreg folder.
2. Edit `makefile.vc` to specify where the header file `fftw3.h` and the lib files `libfftw3-3.lib` are:

   ```
   # Please specify the locations of fftw3.h and the FFTW libs
   FFTW_DIR     = "D:/libs/fftw-3.2.2.pl1-dll32"
   FFTW_INCLUDE = -I(FFTW_DIR)
   FFTW_LIB     = $(FFTW_DIR)/libfftw3-3.lib $(FFTW_DIR)/libfftw3f-3.lib
   ```

3. In a Visual Studio Command Prompt, run

   ```
   nmake -f makefile.vc
   ```

   to build tvreg. This should produce programs `tvrestore` and `chanvese`.

libjpeg and libpng (optional) It is optionally possible under Windows to compile the program with libjpeg and libpng to add support for JPEG and PNG images (libtiff should be possible as well, but it is not explored here). To avoid incompatibility problems, the reliable way to compile is to build the libraries from source using the same compiler.

First, download the libjpeg, libpng, and also the zlib library sources. The zlib library is needed to compile libpng.

- libjpeg sources: `http://www.ijg.org/files/jpegsr8b.zip`
- libpng sources: `http://download.sourceforge.net/libpng/lpng143.zip`
- zlib sources: `http://gnuwin32.sourceforge.net/downlinks/zlib-src-zip.php`

Create a folder to contain the libraries, `C:\libs` for instance. Unzip the library sources into the libs folder so that they are structured as

```
libs
  ├── jpeg-8b
  ├── lpng143
  └── zlib
```

This structure will help keep the code organized. Take care to rename the folder for zlib to "zlib" since libpng will look for it. Below are the steps to build each library. If you want JPEG support, build libjpeg. For PNG support, build zlib first and then build libpng.

Building libjpeg

1. Rename `jconfig.vc` to `jconfig.h`.
2. Open a Visual Studio Command Prompt (under Start Menu → Programs → Microsoft Visual Studio → Visual Studio Tools), go to `libs\jpeg-8b`, and run

   ```
   nmake -f makefile.vc libjpeg.lib
   ```

   This should produce `libjpeg.lib`.

Building zlib

1. Change `zconf.h` line 287, which should have a comment about `HAVE_UNISTD_H`, to "`#if 0`."
2. Open a Visual Studio Command Prompt, go to `zlib\projects\visualc6`, and run

   ```
   vcbuild -upgrade zlib.dsp
   vcbuild zlib.vcproj "LIB Release|Win32"
   ```

   This should produce a folder "Win32_LIB_Release" containing `zlib.lib`.
3. Copy `zconf.h`, `zlib.h`, and `zlib.lib` to `libs\zlib` (libpng will look here).

Building libpng

1. First build zlib.
2. Change `-MD` to `-MT` in the CFLAGS line of `lpng143\scripts\makefile.vcwin32`

   ```
   CFLAGS = -nologo -DPNG_NO_MMX_CODE -MT -O2 -W3 -I..\zlib
   ```

3. From a Visual Studio Command Prompt, go into `lpng143` and run

   ```
   nmake -f scripts\makefile.vcwin32
   ```

   This should produce `libpng.lib`.

Building tvreg with JPEG and PNG support Once the libraries are built, tvreg can be compiled with JPEG and/or PNG support by adjusting tvreg's makefile. Uncomment and edit the lines at the top of `tvreg\makefile.vc` to reflect the locations of libjpeg, libpng, and zlib:

```
# Uncomment and edit the following lines for JPEG support.
LIBJPEG_DIR     = "C:/libs/jpeg-8b"
LIBJPEG_INCLUDE = -I$(LIBJPEG_DIR)
LIBJPEG_LIB     = $(LIBJPEG_DIR)/libjpeg.lib

# Uncomment and edit the following lines for PNG support.
ZLIB_DIR     = "C:/libs/zlib"
ZLIB_INCLUDE = -I$(ZLIB_DIR)
ZLIB_LIB     = $(ZLIB_DIR)/zlib.lib
LIBPNG_DIR     = "C:/libs/lpng143"
LIBPNG_INCLUDE = -I$(LIBPNG_DIR)
LIBPNG_LIB     = $(LIBPNG_DIR)/libpng.lib
```

The makefile will automatically add the corresponding preprocessor symbols based on which libraries are defined. From a Visual Studio Command Prompt, compile with

```
nmake -f makefile.vc
```

7.3 MATLAB MEX

For MATLAB use, compiling is not required, however, the main computational components can be compiled as MEX functions to significantly improve performance. If your system has a C compiler and MEX has been configured to use it, then the functions can be compiled by running the included function `compile_mex` from the MATLAB console,

```
>> compile_mex
```

If MEX has not been configured, run `mex -setup` on the MATLAB console. MEX will try to detect compilers on your system and configure for them automatically. For more information, see

`http://www.mathworks.com/support/tech-notes/1600/1605.html`

or `help mex`. In case compiling MEX is not possible, (slower) M-code implementation is also included. MATLAB automatically uses the MEX versions if they are compiled and falls back to the M-code versions if not.

FFTW

Compiling the MEX functions requires the FFTW3 library (`http://www.fftw.org`). For most Linux distributions, this library is available through the package management system with the name `libfftw3-dev` or similar. On Mac OSX, the library can be obtained through Fink.

Under Windows, the following instructions have been successful with Microsoft Visual C++ (MSVC). First download the precompiled Windows .dll files for FFTW from

```
http://www.fftw.org/install/windows.html
```

Open a Visual Studio Command Prompt by selecting Start Menu → Programs → Microsoft Visual Studio → Visual Studio Tools → Visual Studio Command Prompt. Then, navigate to the FFTW DLL files and run the commands

```
lib /def:libfftw3-3.def
lib /def:libfftw3f-3.def
lib /def:libfftw3l-3.def
```

which should produce three files with .lib extension. Copy the FFTW DLL files into the folder containing the tvreg files (or anywhere else in the MATLAB path).

Next, edit the lines at the top of `compile_mex.m` to tell MEX where to find the FFTW header file `fftw3.h` and the FFTW .lib files. On my system, I put these files in a folder `D:\libs\fftw`, so the configuration is

```
% Configure linking with the FFTW3 library (http://www.fftw.org)
% If FFTW3 is not in MEX's search path, use the following variables.  Set
%    "libfftw3include"     to the location of fftw3.h,
%    "libfftw3"            compiler options for linking with libfftw3
%    "libfftw3f"           compiler options for linking with libfftw3f
libfftw3include = '"D:\libs\fftw"';
libfftw3 = '"D:\libs\fftw\libfftw3-3.lib"';
libfftw3f = '"D:\libs\fftw\libfftw3f-3.lib"';
```

I have enclosed the path names in double quotes, this is required if a path contains spaces. Run `compile_mex` to compile the MEX functions.

Troubleshooting

- `Cannot open include file: 'fftw3.h'`
  Set variable `libfftw3include` in `compile_mex.m` to the directory containing `fftw3.h`.

- `unresolved external symbol _utIsInterruptPending`
  Edit the libut configuration in `compile_mex.m`. Alternatively, set `matlabctrlc = false` to disable the Ctrl-C detection discussed below.

- (Windows) `unresolved external symbol __imp__fftw_plan_many_r2r`
  There is a problem linking with FFTW. Make sure that the variables `libfftw3` and `libfftw3f` in `compile_mex.m` are set to the locations of the FFTW .lib files. For more help, see

  ```
  http://www.fftw.org/install/windows.html
  ```

- (Windows) `Invalid MEX-file 'tvreg.mexw32'`
  Copy the FFTW DLL files into the folder containing the tvreg files.

Ctrl-C detection

Matlab MEX has an unfortunate limitation: there is currently no official method for handling Ctrl-C correctly in a MEX function.

Normally, the keypress Ctrl-C should stop the program. However, if the keypress occurs during the execution of a MEX function, then Matlab does not terminate the MEX function, nor is there an official method that the MEX function can use to detect that Ctrl-C has been signaled. Worse yet, callbacks called from the MEX function (with `mexCallMATLAB`) will still happen, but may produce strange results and run slowly.

tvreg uses the undocumented `utIsInterruptPending` function to detect and abort computation when Ctrl-C is pressed [36]. This should allow Ctrl-C to work as expected. However, in case this method causes problems, Ctrl-C detection can be disabled. This is done by changing the `matlabctrlc` variable in `compile_mex.m` to false.

Thanks

I have received much positive feedback and encouragement as a result of the tvreg project, which has led to many interesting discussions. Thanks to everyone for your support!

References

[1] Y. Boykov and M.-P. Jolly. "Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-D images." *International Conference on Computer Vision*, vol. I, pp. 105–112, 2001.

[2] A. Buades, B. Coll, and J.-M. Morel. "A review of image denoising algorithms, with a new one." *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005.

[3] A. Chambolle and P.L. Lions. "Image Recovery via Total Variation-Based Restoration." *SIAM J. Sci. Comput.* 20, pp. 1964–1977, 1999.

[4] A. Chambolle. "An Algorithm for Total Variation Minimization and Applications." *Journal of Mathematical Imaging and Vision*, 20(1-2), 2004

[5] A. Chambolle and J. Darbon. "On Total Variation Minimization and Surface Evolution using Parametric Maximum Flows," *IJCV*, vol. 84, no. 3, pp. 288–307, 2009.

[6] T.F. Chan, B. Sandberg, and L. Vese. "Active Contours Without Edges for Vector-Valued Images." *Journal of Visual Communication and Image Representation*, vol. 11, pp. 130–141, 2000.

[7] T.F. Chan, S. Osher, and J. Shen. "The digital TV filter and nonlinear denoising" *IEEE Transactions on Image Processing*, vol. 10, no. 2, pp. 231–241, 2001.

[8] T.F. Chan and L.A. Vese. "Active Contours Without Edges." *IEEE TIP*, vol. 10, no. 1, pp. 266–277, 2001.

[9] T.F. Chan, S.-H. Kang and J. Shen. "Euler's Elastica and Curvature-Based Inpainting." *SIAM J. Appl. Math.*, 63, pp. 564–592, 2002.

[10] T.F. Chan and S. Esedoglu. "Aspects of total variation regularized L^1 function approximation." *UCLA CAM Report* 04–07, 2004.

[11] T.F. Chan and J. Shen. *Image Processing and Analysis: Variational, PDE, wavelet, and Stochastic Methods.* SIAM, 2005.

[12] T.F Chan and C.-K. Wong. “Total variation blind deconvolution.” *IEEE Transactions on Image Processing*, vol. 7, no. 3, pp. 370–375, 1998.

[13] G. Chung and L. Vese. “Energy Minimization Based Segmentation and Denoising Using a Multilayer Level Set Approach.” *Energy Minimization Methods in Computer Vision and Pattern Recognition*, vol. 3757/2005, pp. 439–455, 2005.

[14] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian. “Image denoising by sparse 3D transform-domain collaborative filtering.// *IEEE Trans. Image Process.*, vol. 16, no. 8, pp. 2080–2095, 2007.

[15] A. Efros and T. Leung. “Texture Synthesis by Non-parametric Sampling.” *Int. Conf. on Computer Vision*, pp. 1033–1038, 1999.

[16] G. Gilboa and S. J. Osher. “Nonlocal Linear Image Regularization and Supervised Segmentation.” *SIAM Multiscale Modeling & Simulation*, vol. 6, no. 2, pp. 595–630, 2007.

[17] T. Goldstein and S. Osher. “The Split Bregman Method for L^1 Regularized Problems.” *UCLA CAM Report* 08–29, 2008.

[18] A. Haddad. “Stability in a Class of Variational Methods.” *Appl. Comput. Harmon. Anal.* 23, pp. 57–73, 2007.

[19] M. Jung, G. Chung, G. Sundaramoorthi, L. Vese, A. Yuille. “Sobolev gradients and joint variational image segmentation, denoising and deblurring.” *IS&T/SPIE on Electronic Imaging*, vol. 7246, 2009.

[20] M. Keegan, B. Sandberg, and T. Chan. “A Logic Framework for Multiphase Multichannel Image Segmentation.” *UCLA CAM Report* 10-60, September 2010

[21] T. Le, R. Chartrand, and T. Asaki. “A Variational Approach to Constructing Images Corrupted by Poisson Noise,” *JMIV*, vol. 27(3), pp. 257–263, 2007.

[22] S. Martucci, “Symmetric convolution and the discrete sine and cosine transforms,” *IEEE Trans. Sig. Processing* SP-42, p. 1038–1051, 1994.

[23] D. Mumford and J. Shah. “Optimal approximation by piecewise smooth functions and associated variational problems.” Commun. Pure Appl. Math, vol. 42, pp. 577–685, 1989.

[24] G. Peyré, S. Bougleux, and L. Cohen. “Non-local Regularization of Inverse Problems.” *ECCV08*, III, pp. 57–68, 2008.

[25] L.I. Rudin, S. Osher, and E. Fatemi. “Nonlinear total variation based noise removal algorithms.” *Physica D*, vol. 60, p. 259–268, 1992.

[26] B. Sandberg, T. Chan, L. Vese. “A Level-Set and Gabor-Based Active Contour Algorithm for Segmenting Textured Images.” *UCLA CAM Report* 02-39, 2002.

[27] S. Setzer. “Operator Splittings, Bregman Methods and Frame Shrinkage in Image Processing,” *International Journal of Computer Vision*, accepted.

[28] E. Sharon, M. Galun, D. Sharon, R. Basri, and A. Brandt. “Hierarchy and adaptivity in segmenting visual scenes.” *Nature*, vol. 442, no. 7104, pp. 719–846, 2006.

[29] G. Sundaramoorthi, A Yezzi, A. Mennucci, and G. Sapiro. “New Possibilities With Sobolev Active Contours.” *International Journal of Computer Vision*, vol. 84, no. 2, pp. 113–129, 2009.

[30] E. Tadmor, S. Nezzar, and L. Vese. “A multiscale image representation using hierarchical (BV, L^2) decompositions.” *Multiscale Modeling & Simulation*, vol. 2, no. 4, pp. 554–579, 2004.

[31] D. Tschumperlé and R. Deriche. “Vector-Valued Image Regularization with PDE’s: A Common Framework for Different Applications.” *IEEE Int. Conf. on Computer Vision and Pattern Recognition*, vol. 1, pp. 651–659, 2003.

[32] D. Tschumperlé. “Fast Anisotropic Smoothing of Multi-Valued Images using Curvature-Preserving PDE’s.” *International Journal of Computer Vision*, vol. 68, no. 1, pp. 65–82, 2006.

[33] Ranjith Unnikrishnan, Caroline Pantofaru, Martial Hebert, ”Toward Objective Evaluation of Image Segmentation Algorithms,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 29, no. 6, pp. 929-944, June 2007, doi:10.1109/TPAMI.2007.1046

[34] Y. Wang, J. Yang, W. Yin, and Y. Zhang. "A new alternating minimization algorithm for total variation image reconstruction." *SIAM Journal on Imaging Sciences*, vol. 1, no. 3, pp. 248–272, 2008.

[35] J. Weickert. *Anisotropic diffusion in image processing.* ECMI Series, Teubner-Verlag, Stuttgart, Germany, 1998.

[36] W. Yin. "Ctrl-C Detection in MATLAB MEX Files." Retrieved December 12, 2010 from `http://www.caam.rice.edu/~wy1/links/mex_ctrl_c_trick`

[37] Y. You and M. Kaveh. "A regularization approach to joint blur identification and image restoration." *IEEE Transactions on Image Processing*, vol. 5, pp. 416–428, 1996.

[38] G.Yu, G. Sapiro, and S. Mallat. Solving Inverse Problems with Piecewise Linear Estimators: From Gaussian Mixture Models to Structured Sparsity." Submitted to IEEE Trans. on Image Processing, 2010.