ASPIRE 3.0 USER'S GUIDE:
A SPARSE ITERATIVE RECONSTRUCTION LIBRARY

Jeffrey A. Fessler

COMMUNICATIONS & SIGNAL PROCESSING LABORATORY
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

July 1995
Revised August 14, 2006

Technical Report No. 293
Approved for public release; distribution unlimited.

ASPIRE 3.0 User's Guide: A Sparse Iterative Reconstruction Library

Jeffrey A. Fessler
4240 EECS University of Michigan, Ann Arbor, MI 48109-2122
email: fessler@umich.edu, phone: 734-763-1434

August 14, 2006

Technical Report # 293
Communications and Signal Processing Laboratory
Dept. of Electrical Engineering and Computer Science
The University of Michigan

Abstract

ASPIRE 3.0 is a collection of ANSI C language programs for performing tomographic image reconstruction and image restoration using statistical methods. This user's guide describes how to compile and use the software.

Contents

This work was supported in part by DOE grant DE-FG02-87ER60561 and NIH grants CA-60711 and CA-54362.

Notice

This documentation will evolve over time in response to user questions and software updates. Please check the WWW site to verify you have the most recent version of the documentation before sending questions.

1 Introduction

A wide variety of inverse problems can be expressed roughly in the following form: find an approximate solution to the equation $\boldsymbol{y} = \boldsymbol{G}\boldsymbol{x}$ where $\boldsymbol{G}$ is a sparse matrix, $\boldsymbol{y}$ is related to the measurements, and $\boldsymbol{x}$ is the unknown image. This user's guide[2] describes how to install and use the ASPIRE 3.0 software to solve this type of problem using statistical methods.

The steps outlined below include:

- Downloading and compiling ASPIRE 3.0.
- Generating a "system matrix" $\boldsymbol{G}$ and storing it in a sparse binary format "weight file."
- Converting your data to the AVS `.fld` format.
- Tabulating the relationship between the smoothing parameter β and image resolution (*e.g.*, FWHM) for your system [1, 2].
- Running regularized iterative algorithms, using the value of β from the table that yields the desired resolution. (The user must choose the desired resolution, keeping in mind the resolution/noise tradeoff.)

By using an *unregularized* method such as the ordinary ML-EM algorithm, one could avoid the (easily performed) step of determining how β relates to resolution. However, unregularized methods give poorer quality images than regularized methods. You would also then need to decide "how many iterations?"

I have tested ASPIRE 3.0 extensively using the Insight software development package from ParaSoft Corp., so it should be relatively free of memory leaks, segmentation violations, etc. Therefore, most of the error messages will be due to problems with the input data or parameters.

2 Notation

This document adopts the conventional typography of using the `typewriter` font for things you will actually type literally, and *italics* for arguments that you will need to supply.

3 Installation

3.1 Getting software

Hopefully you have obtained the already compiled programs, `wt`, `op`, and `i`, by following the instructions on my web page. If so, you can skip the compiling instructions below, obviously.

3.2 Version information

Executing any of the three programs with no arguments (*e.g.*, just typing `op` at a unix prompt) will print the date and time the program was compiled, and a helpful list of the top-level arguments of that program. You will find many features in those list that are not documented here. If the compile date was a long time ago, then bug me to update your version!

[2] This is not a software developer's guide. Although you may have access to some form of source code for ASPIRE 3.0, you should consider it a "black box," except of course that the internal workings are described in publications. I would gladly document the key elements of the source code if contracted to do so, or as part of a commercial agreement. Appendix C has some brief information for developers.

3.3 Source code

In the event that you actually have the source code, (*e.g.*, if you are my student compiling a modified version), then you should have the following 9 files.

- `def.h` contains all the declarations that tend to be system-dependent, with the particularly variable ones defined as macros that you can redefine if necessary. ASPIRE 3.0 has been compiled on DEC Alphas running OSF, SUNs running SUNOS and Solaris, PC Linux boxes, and Mac OS X (my favorite). For other configurations you may need to modify `def.h`.
- `wt.c` with `wt.h` has the code for generating weight files.
- `io.c` has the input/output subroutines.
- `op.c` with `op.h` has a collection of utility operations.
- `i.c` with `i.h` has the collection of iterative reconstruction methods.
- You probably also have a `Makefile` with a huge number of options (ask me).
- You should also get the script `j` from ∼`fessler/l/src/script/j` (or ask me) which is a "jiffy" little display script that invokes `xv` based on Section 5 below.

3.4 Compiling

You must use an ANSI C compiler. To compile, put the files listed above in the same directory, and type something like the following.

```
cc -o wt wt.c -lm
cc -o op op.c io.c -lm
cc -o i -Dnomainwt -Dnomainop i.c wt.c op.c io.c -lm
```

You probably want to add optimization flags. This creates three programs: `wt`, `op`, and `i`.

For the `gcc` compiler, I use the following flags:

```
-O3 -ffast-math -fexpensive-optimizations -ansi -Wall -Wshadow -Wpointer-arith
     -Wcast-qual -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes
                         -Wmissing-declarations -Werror
```

4 Weight generation

To generate a system matrix[3] $\boldsymbol{G}$, you must first create a small ASCII file, called the *description file* that describes the imaging system geometry. The filename must have suffix `.dsc`. Several types of system geometries are implemented, as described in Appendix A. For historical reasons[4], we call the file containing the system matrix $\boldsymbol{G}$ a "weight file," and its suffix is usually `.wtf`.

To generate the weight file from a description file named `tomo.dsc`, type:

```
wt gen tomo
```

which will create a binary file named `tomo.wtf`. The top few lines are ASCII, followed by two form-feeds, followed by the binary data, so you can probably safely type `more tomo.wtf` if you are curious.

You can see the header of this file with the command

[3] $\boldsymbol{G}$ is mnemonic for *geometry*, because the tomograph geometry principally determines $\boldsymbol{G}$. In contrast, the matrix $\boldsymbol{A}$ (see below) includes both geometrical effects as well as attenuation, detector efficiency, etc.

[4] Note that the term "weights" is used in at least three different contexts in image reconstruction: for the elements of the matrix $\boldsymbol{G}$, for the diagonal elements of the inverse of the measurement covariance matrix, and for the w_{jk}'s that penalize neighboring pixels j and k.

```
wt head < tomo.wtf
```

which will also show a chunky picture of the support map.

For example, suppose the file `toy.dsc` contains the following lines (see Appendix A).

```
system  0
nx      6
ny      4
support all
scale   1
psf     5       3
1 2 3 2 1
5 7 9 7 5
1 2 3 2 1
```

We generate the weight file by typing `wt gen toy` at the command line. This may produce a few warning messages about "0 weights," which can be disregarded. (It is a bit inefficient, but generally harmless, to store a few 0's in a sparse matrix.) The output of `wt printfull toy.wtf` is the following.

```
 0: 9 7 5 0 0 0 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1: 7 9 7 5 0 0 2 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 2: 5 7 9 7 5 0 1 2 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 3: 0 5 7 9 7 5 0 1 2 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0
 4: 0 0 5 7 9 7 0 0 1 2 3 2 0 0 0 0 0 0 0 0 0 0 0 0
 5: 0 0 0 5 7 9 0 0 0 1 2 3 0 0 0 0 0 0 0 0 0 0 0 0
 6: 3 2 1 0 0 0 9 7 5 0 0 0 3 2 1 0 0 0 0 0 0 0 0 0
 7: 2 3 2 1 0 0 7 9 7 5 0 0 2 3 2 1 0 0 0 0 0 0 0 0
 8: 1 2 3 2 1 0 5 7 9 7 5 0 1 2 3 2 1 0 0 0 0 0 0 0
 9: 0 1 2 3 2 1 0 5 7 9 7 5 0 1 2 3 2 1 0 0 0 0 0 0
10: 0 0 1 2 3 2 0 0 5 7 9 7 0 0 1 2 3 2 0 0 0 0 0 0
11: 0 0 0 1 2 3 0 0 0 5 7 9 0 0 0 1 2 3 0 0 0 0 0 0
12: 0 0 0 0 0 0 3 2 1 0 0 0 9 7 5 0 0 0 3 2 1 0 0 0
13: 0 0 0 0 0 0 2 3 2 1 0 0 7 9 7 5 0 0 2 3 2 1 0 0
14: 0 0 0 0 0 0 1 2 3 2 1 0 5 7 9 7 5 0 1 2 3 2 1 0
15: 0 0 0 0 0 0 0 1 2 3 2 1 0 5 7 9 7 5 0 1 2 3 2 1
16: 0 0 0 0 0 0 0 0 1 2 3 2 0 0 5 7 9 7 0 0 1 2 3 2
17: 0 0 0 0 0 0 0 0 0 1 2 3 0 0 0 5 7 9 0 0 0 1 2 3
18: 0 0 0 0 0 0 0 0 0 0 0 0 3 2 1 0 0 0 9 7 5 0 0 0
19: 0 0 0 0 0 0 0 0 0 0 0 0 2 3 2 1 0 0 7 9 7 5 0 0
20: 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 2 1 0 5 7 9 7 5 0
21: 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 2 1 0 5 7 9 7 5
22: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 2 0 0 5 7 9 7
23: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 0 0 0 5 7 9
```

Note that there are $6 \cdot 4 = 24$ rows and 24 columns, since this $\boldsymbol{G}$ matrix is 24×24.

I recommend you try the above now to verify your installation.

The output of `wt printsparse toy.wtf` is the following.

```
0     0 9
0     1 7
```

```
    0     2 5
    0     6 3
    0     7 2
    0     8 1
    1     0 7
    1     1 9
    1     2 7
    1     3 5
...
   22    20 5
   22    21 7
   22    22 9
   22    23 7
   23    15 1
   23    16 2
   23    17 3
   23    21 5
   23    22 7
   23    23 9
```

The first column is the column index j in $\boldsymbol{G}$, the second column is the row index i, and the third column is g_{ij}. (Columns, rows, image slices, etc. are all numbered starting from 0, as usual for C programs.) (Note that in fact the zero entries have been removed, so there is no storage inefficiency. Software can assume the (stored) g_{ij}'s are nonzero.)

4.1 Converting to a custom sparse format

For realistic sizes of tomographic systems, there will be millions of nonzero entries in $\boldsymbol{G}$, so printing them will usually be impractical. Nevertheless, if you need to extract an ASPIRE 3.0 system matrix and convert it into your own custom format, using `wt printsparse` is the simplest way.

4.2 Converting custom system matrix to ASPIRE 3.0

If you want to make your own `.wtf` but you do not have Matlab, then you can create an ASCII file, say *file*`.dat`, containing the same number of lines as there are nonzero entries in $\boldsymbol{G}$, where each line is of the form

$$j \quad i \quad g_{ij}$$

(which is equivalent to the output of `wt printsparse`). Then type

`wt load` *file*`.wtf` *file*`.dat` *nx ny nb na*

where the image and sinogram sizes are the last 4 arguments (see Appendix). This will create an ASPIRE 3.0 format `.wtf` file.

4.3 Interface with Matlab

If you need it, I have a Matlab `mex` file

`[G, nx,ny, nb,na] = wtfmex('load', '`*file*`.wtf')`

for reading `.wtf` format files into Matlab sparse matrices, as well as

`wtfmex('`*file*`.wtf',` *G, nx, ny, nb, na*`)`

for writing a Matlab sparse matrix `G`=$\boldsymbol{G}$ to a `.wtf` file.

5 Data format

ASPIRE 3.0 automatically determines the file format from the (required) three letter extension. Currently, the primary input/output data format supported by the released version of ASPIRE 3.0 is the `.fld` format of AVS (Application Visualization System). This format is particularly simple to describe and use (see Appendix B). For example, it is very easy to write an M-file for reading `.fld` files into Matlab.

In the past, ASPIRE 3.0 could also read and write Matlab `.mat` files if compiled appropriately. However, Mathworks made their file I/O interface a ridiculous moving target, so this option is no longer available. However, my Matlab Tomography Toolbox has routines `fld_read.m` and `fld_write.m` that can read and write `.fld` files from Matlab, so there is no longer any real need for ASPIRE 3.0 to read Matlab's format.

If you have X windows and the shareware program `xv`, you can display 2D `.fld` files by typing:

```
op conv tmp.pgm file.fld byte - 1
xv tmp.pgm &
```

For 3D `.fld` files, something like

```
op vol2mat tmp.fld file.fld
op conv tmp.pgm tmp.fld byte - 1
xv tmp.pgm &
```

will arrange the stack of slices as a grid and then display them. The `pgm` format is a simple binary 8-bit 2d format that is supported by `xv`. You will probably want aliases or scripts to simplify performing the above frequently.

Alternatively, if you type `op display` *file*, then ASPIRE 3.0 will call `xv` for you. Type `op disp` to see the many (and evolving) display features .

There is another format partially supported. If you name an output file with the extension `.raw`, then ASPIRE 3.0 will write out raw binary data with no header. You cannot use the extension `.raw` for *input* files though, since ASPIRE 3.0 needs to know the image dimensions etc. However, you can provide an AVS "external" header for such raw files so that ASPIRE 3.0 can read them: see Appendix B.

Finally, your version may be able to read files in the pre-7.0 CTI "matrix" format ending in `.scn .nrm .atn .img`, and may be able to write CTI `.img` files, albeit with very limited header information entered. Post-7.0 CTI format files ending in `.s` etc. are also supported to various degrees, thanks to considerable pain (and help from Christian Michel).

6 Tabulating β

Penalized-likelihood methods for image reconstruction maximize objective functions of the form

$$\Phi(\boldsymbol{x}) = L(\boldsymbol{y}; \boldsymbol{x}) - \beta R(\boldsymbol{x}),$$

where $L(\boldsymbol{y}; \boldsymbol{x})$ is the log-likelihood of the measurements given a hypothetical image $\boldsymbol{x}$, $R(\boldsymbol{x})$ is a measure of the roughness of the image $\boldsymbol{x}$, and β controls the tradeoff between resolution and noise. You have to choose β, which some people consider to be a big disadvantage of penalized-likelihood methods. This is a little unfair, since *all* reconstruction methods have some fiddle-factor that controls the resolution/noise tradeoff, including filtered backprojection. But the problem with β historically has been that it is effectively unitless, so it is not obvious even where to begin (good values are probably somewhere between 2^{-20} and 2^{20}, depending on your units, etc.). If you are going to be experimenting with many different system geometries for only a short amount of time with each, then choosing β by trial-and-error might be expedient enough. But if you are going to consistently use a particular system geometry for several data sets, then the tabulation method described in this section will be helpful in the long run.

The basic idea is that the local impulse response of penalized likelihood methods is approximately

$$[\boldsymbol{G}'\boldsymbol{G} + \beta \boldsymbol{R}]^{-1} \boldsymbol{G}' \boldsymbol{p}$$

where $\boldsymbol{p}$ is the (noiseless) projection of a point source at some location [1–3]. What you need to do is compute the local impulse response given above (using ASPIRE 3.0) for several values of β. Each computation gives an "image" which looks like a small "bump" [2]. You calculate your favorite measure of resolution of these bumps (*e.g.*, FWHM), thereby producing a table relating β to FWHM. In subsequent studies then, you simply decide what resolution appeals to you, and then look up β from the table. This only works with my modified penalty [1, 2], by the way, *not* with the conventional penalties in the literature.

6.1 Simulations

This tabulation process is particularly easy for simulations since one can simply compute projections of a point source:

$$\boldsymbol{p} = \boldsymbol{G}\boldsymbol{e}_j.$$

At least for spatially-invariant systems and for a pixel j not too close to the image edge, we can compute

$$[\boldsymbol{G}'\boldsymbol{G} + \beta \boldsymbol{R}]^{-1} \boldsymbol{G}'\boldsymbol{G}\boldsymbol{e}_j \tag{1}$$

to within a very close approximation using FFT's [2]. ASPIRE 3.0 includes programs that do most of the work for you. After creating your description file, say `tomo.dsc`, type

```
i gtg2 gtgraw.fld - tomo.dsc
```

This will create a `nx` by `ny` image in the file `gtgraw.fld` of $\boldsymbol{G}'\boldsymbol{G}\boldsymbol{e}_j$ for a pixel at the "center" of the image. You can look at this image using the display method described in Section 5. It should look something like the well-known $1/r$ response. To ensure real results, the FFT approximation to (1) must use a symmetric kernel (PSF). The following command makes a symmetric version:

```
op psfsym gtgsym.fld gtgraw.fld
```

Then type

```
op psfpls psf.fld - gtgsym.fld -6 1
```

which will compute a Fourier-based approximation to (1) for $\beta = 2^{-6}$ and for $\boldsymbol{R}$ corresponding to a quadratic penalty with a 1st-order neighborhood (horizontal and diagonal neighbors only) [2]. You can change the `1` to a `2` for a 2nd-order neighborhood, but you probably will not see much difference (in the final reconstructed image), for quadratic penalties. The `op psfpls` program above will print out the FWHM for the PSF saved in `psf.fld`. The output will look something like:

```
N      Right Left  Up    Down  Horiz Vert  Avg
16  16 16.13 17.50 14.13 15.50 16.82 15.82 16.32       psfpls
```

which means that the FWHM is 16.82 pixels horizontally and 15.82 pixels vertically. If you prefer some other measure of resolution than FWHM, just analyze `psf.fld` using your favorite method. Now if you repeat this for several values of $\log_2 \beta$, you can generate a table like Table 1, which corresponds to the following `.dsc` file:

$\log_2 \beta$	FWHM of $[\boldsymbol{G}'\boldsymbol{G} + \beta\boldsymbol{R}]^{-1}\boldsymbol{G}'\boldsymbol{G}\boldsymbol{e}_j$
-13	1.20
-12	1.21
-11	1.26
-10	1.33
-9	1.44
-7	1.81
-5	2.52
-3	3.40
-1	4.80
0	5.74
1	7.00

Table 1: Relationship between β and resolution for the system matrix described in the text, for a 1st order quadratic penalty.

```
system  2
nx      64
ny      64
nb      64
na      60
support ellipse 0 0 30 30
orbit           180
orbit_start     0
pixel_size      1
ray_spacing     1
strip_width     1
scale           1
```

I recommend you regenerate Table 1 to test your installation. Note that as $\beta \to 0$, the FWHM goes to 1 pixel, which is the lower limit. As a rough guess, I suggest first using the value of β that corresponds to a spatial resolution of about 3 pixels FWHM, and then adjust up or down depending on the noise.

6.2 Real Systems

For a real tomographic system, the projections of a point source would be something like:

$$\boldsymbol{p} = \boldsymbol{G}_{\text{true}}\delta_{(x_p,y_p)}$$

where $\boldsymbol{G}_{\text{true}}$ is the true (imperfectly known) system, and $\delta_{(x_p,y_p)}$ is a point source at spatial location (x_p, y_p). If you think you have specified a system matrix $\boldsymbol{G}$ that is very close to $\boldsymbol{G}_{\text{true}}$, then you can probably proceed with the recipe given above for simulations. Or you could use high-count *measured* projections of a "point" source. Or you can use the trial-and-error method to choose β. Or contact me to discuss further. This is an active research area in my group [3–5] and will continue to be until at least 2003 since simplifying the process of choosing β is probably essential to achieving wider use of regularized methods.

7 SPECT attenuation (2d)

For SPECT, the effects of attenuation must be built into the system matrix by an element-by-element multiplication. ASPIRE 3.0 includes routines for performing this multiplication for certain system geometries. You must first create a 2D attenuation map, say `mu.fld`, whose pixel values have units inverse length. (If you have a transmission scan, then you can use the reconstruction methods described below to estimate this attenuation map.) Then running

`i atten out.wtf in.wtf mu.fld` *pixel_size*

performs the element-by-element multiplication and creates a new system matrix that includes the affect of attenuation. The fourth argument should scale `mu.fld` to make it unitless. So if your attenuation map has units inverse centimeters, then *pixel_size* should be in centimeters. The algorithm for computing the necessary line lengths is fairly crude, but should be adequate for attenuation maps that are fairly smooth.

The above approach means that it will be somewhat inconvenient to use this version of ASPIRE 3.0 for routine processing of multiple SPECT slices. (PET is no problem since the attenuation affects the matrix $\boldsymbol{A}$ differently.) The 3D users manual [6] describes software for reconstructing multiple SPECT slices with attenuation correction and 3D depth-dependent detector response compensation.

8 Initial image

Any iterative reconstruction method needs an initial guess. For penalized-likelihood methods, an FBP image with the appropriate spatial resolution [1, 2, 7] is an ideal choice. (For unregularized methods, a uniform image is used conventionally.)

8.1 Transmission FBP Reconstruction

For FBP reconstruction from raw transmission scan data (*i.e.*, intensity measurements to which no logarithm has been applied), the following command does it all

`i fbp2t dsc` *image.out sino.out yi bi bi_factor ri ri_factor tomo.dsc window*

The assumed measurement model here is:

$$y_i = b_i e^{-l_i} + r_i + \text{noise}, \quad \text{where } l_i = [\boldsymbol{G}\boldsymbol{\mu}]_i = \sum_j g_{ij}\mu_j,$$

and the b_i's denote the blank-scan (or air-scan) factors.

- The argument *yi* is the filename containing the $n_b \times n_a \times n_z$ transmission data y_i.
- If the argument *bi* is just the default dash `-`, then b_i is taken to be the (positive) value given by *bi_factor*. Otherwise b_i is taken to be the values in the file *bi* multiplied by *bi_factor*.
- If the argument *ri* is just the default dash `-`, then r_i is taken to be the (nonnegative) value given by *ri_factor*. Otherwise r_i is taken to be the values in the file *ri* multiplied by *ri_factor*. Usually one will use the default dash `-` for *ri* and `0` for *ri_factor*. Exceptions include PET transmission scans with prompts/delays acquired separately, or X-ray CT scans for which scatter estimates are available.
- If *sino.out* is not the default dash `-`, then this file is written with the values

$$\hat{l}_i = -\log(1 + \text{fix\_negatives}(\text{smooth}((y_i - r_i)/b_i)) - 1)).$$

- The "fix_negatives" operation (enabled by default) checks for any residual non-positive sinogram values after the smoothing, and tries to interpolate neighboring positive values (if any) to "fill in" a positive value, so that the logarithm will work. For any remaining non-positive values (*i.e.*, if all 4 neighbors in the sinogram are non-positive), the log-value is set to zero, which will make streaks. Those streaks indicate the need for more smoothing! This option can be disabled (to make even more streaks). Type `i fbp2t` to see all the arguments.

- *tomo.dsc* is the name of the geometry description file.
- The *window* argument specifies the type of smoothing, and takes just the same arguments as the FBP window. Type `op fbp` to see all of the options. A reasonable choice of *window* for PET transmission scans would be something like `3d@gauss,7,3,6@gauss,1,8,1` which does z-smoothing using a Gaussian kernel with `7`mm FWHM on `3`mm slice spacing discretized using $13 = 2 \times 6 + 1$ samples, along with a ramp filter apodized by a Gaussian filter corresponding to an `8`"mm" transaxial FHWM, assuming that the `.dsc` file used "mm" units.

8.2 Emission FBP Reconstruction

For FBP emission reconstruction, use

`i fbp2e dsc` *image.out sino.out yi ci ci_factor ri ri_factor tomo.dsc window*

This method is based on the measurement model

$$y_i = c_i [\boldsymbol{G}\boldsymbol{x}]_i + r_i,$$

where y_i is the raw sinogram measurements, r_i is an estimate of randoms and scatter contribution, and c_i is a calibration sinogram that includes survival probabilities (inverse of attenuation correction factors), deadtime etc. Typically in PET the randoms are precorrected, in which case one should use the default hyphen `-` for *ri* and `0` for *ri_factor*. In typical cases where attenuation etc. has also already been corrected (or ignored), then use a hyphen for *ci* and a `1` for *ci_factor*.

If an attenuation map is available, then one can compute the c_i's by reprojecting that attenuation map and then exponentiating its negative as follows:

```
i proj2 line_integrals.fld attenuation_map.fld tomo.dsc
op nonlin exp ci.fld line_integrals.fld -1 1
```

If in addition the sinogram normalization factors are available, then those can be incorporated into `ci.fld` using `op mul` or `op div`.

If *sino.out* is not the default hyphen, then it will write

sino = smooth((yi-ri)/ci)

to the *sino.out* file. The output image will be the ramp-filtered reconstruction of the *smoothed* sinogram *sino*.

If everything is precorrected, one can use the following simpler command:

`i fbp dsc imageout.fld sinoin.fld tomo.dsc` *window*

will apply FBP to the input sinogram. Type `op fbp` for a list of *window* options. To match to penalized-likelihoood, try the following window: `cls3sinc,`*log2beta*`,1,1` where *log2beta* is replaced by the numerical value of $\log_2 \beta$, *e.g.*, -6, that you plan to use for iterative reconstruction.

9 Image reconstruction

You have converted your data to `.fld` format, generated an appropriate weight file, and read some papers on image reconstruction. Now you are ready to reconstruct images. You must make several decisions, namely, whether to use a penalized weighted least squares or a penalized-likelihood objective function, what type of penalty to use, and which optimization algorithm to use. In my WWW site (address at end of bibliography) I have a page of opinions and recommendations about cost functions and algorithms.

9.1 Penalized Weighted Least Squares

The simplest regularized method uses the penalized least-squares cost function:

$$\Phi(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{G}\boldsymbol{x}\|^2 + \beta R(\boldsymbol{x}),,$$

where the roughness penalty $R(\boldsymbol{x})$ has a form like

$$R(\boldsymbol{x}) = \frac{1}{2}\sum_j \sum_k w_{jk}\, \psi(x_j - x_k),$$

where ψ is a convex function. For a "1st-order neighborhood" w_{jk} is 1 for horizontal and diagonal neighbors and zero otherwise, and for a "2nd-order neighborhood" w_{jk} is also $1/\sqrt{2}$ for diagonal neighbors. Note each pair of pixels is counted twice by the double sum, hence the $\frac{1}{2}$ out front.

In PET and SPECT imaging, the measurements have different variances, so a PWLS cost function is preferable:

$$\Psi(\boldsymbol{x}) = \frac{1}{2}\,(\boldsymbol{y} - \boldsymbol{G}\boldsymbol{x})' \,\mathsf{diag}\{u_i\}\,(\boldsymbol{y} - \boldsymbol{G}\boldsymbol{x}) + \beta R(\boldsymbol{x}),$$

where the u_i are "weights" (inverse of the variance of Y_i, see [8]).

If you type `i pwls2` you will get the syntax of how to minimize this Ψ. The output should include something like:

```
Usage: pwls2 out init- yi nder1- nder2-
wtf mask- method
[saver- flag_obj(0) flag_nonneg(1) pix_max scale_init(0) slices-]
```

(The argument order is fixed.) The arguments followed by a dash "`-`" are optional; using the dash will give sensible defaults. Here is what each argument means.

- *out* is the name of the output image file.
- *init* is the name of the initial image file. The default is a uniform image, but I highly recommend FBP (corrected by Chang for SPECT), for convergence rate reasons detailed in [8, 9].
- *yi* is the input sinogram $\boldsymbol{y}$, typically corrected for attenuation, scatter, randoms, etc. (The statistical effects are accounted for in the weights u_i [8].)
- *nder1* should always just be the default hyphen.
- *nder2* is the sinogram-sized set of u_i's. Default is $u_i = 1$, which is uniform weighting. (Not recommended: see figures in [8]!) See appendix of [8] for details on computing u_i.
- *wtf* is the name of the weight file containing the system matrix $\boldsymbol{G}$ (or in the SPECT case, the modified $\boldsymbol{G}$ that includes attenuation).

- *mask* is a 2D binary file that can override the `support` in the `.wtf`. I recommend using the default dash unless you are feeling brave.
- *method* specifies how many iterations of what algorithms using which penalties. See below.
- *saver* should usually be `-`. There are additional options that allow saving the intermediate iterations. See output when you type `i pwls2`.
- *flag_obj*: if `1`, will compute Φ every iteration and print. Use `0` except for debugging.
- *flag_nonneg*: if `1`, enforce nonnegativity constraint $\boldsymbol{x} \geq \boldsymbol{0}$. If `0`, unconstrained.
- *pix_max*: maximum allowable pixel value, which can be useful for transmission images if you know the maximum attenuation coefficient. Use a big number like `1e9` otherwise.
- *scale_init*: If you have used `i fbp2e` then the initial FBP image *should* be properly scaled, in which case use `0`. If you are *not* sure that the initial image is properly scaled, then use `1`, and ASPIRE 3.0 will scale your initial image to best fit the date before iterating. This requires an extra projection operation, so it is best to match scaling of FBP with the $\boldsymbol{G}$ matrix by careful bookkeeping (*i.e.*, preserving counts in the emission case). ASPIRE 3.0 will print out the scale factor it applied to the initial image. If your initial image is scaled correctly, it should print a value within a few percent of 1.
- *slices*: Use, say, `7,12` to only reconstruct slices 7 through 12 (numbered from 0). The default (dash) is to do all slices. This is 2D reconstruction, but it can work slice-by-slide with all or part of a stack of sinograms stored in a "3D" input file.

The generic syntax of the *method* argument looks like

@niter1@algorithm1@penalty1@niter2@algorithm2@penalty2... (2)

This allows you to run *niter1* iterations of *algorithm1* for an cost function that includes *penalty1*, followed by *niter2* iterations of *algorithm2* for a cost function that includes *penalty2*, etc. Usually you will just have one algorithm. For example,

`@2@cg,diag@-6,quad,2,-@10@ca,0.6,raster1,@-6,quad,1,-`

means 2 iterations of conjugate gradient with a diagonal preconditioner [10] (other preconditioners may be documented later), followed by 10 iterations of coordinate ascent using the conventional raster scan ordering, and the under-relaxation parameter of successive over-relaxation[5] [8] is $\omega = 0.6$. In this example, the *penalty* is

`-6,quad,1,-`

which would be a quadratic penalty ($\psi(t) = t^2/2$), with a 1st-order neighborhood, with $\beta = 2^{-6}$, and with the conventional choice for w_{jk}.

`4,quad,2,b2info`

would be the quadratic penalty with a 2nd-order neighborhood, with $\beta = 2^4$, and with the w_{jk} modified as in [1, 2] to yield nearly uniform resolution. I recommend using that modified penalty (and you must if you want the β tabulation method described in Section 6 to work).

There are more complicated ψ functions implemented and partially implemented. See the output of `i pwls2` for all of the options. Please discuss with me if interested in anything particular.

Here is a complete example of how you would run 10 coordinate ascent iterations (say, starting from an FBP image *init*`.fld`) to minimize Φ:

[5] I used to recommend 0.6, but I may not have been using the best initial image in the experiments used to draw that conclusion. Values between 0.6 and 1.0 all seem to yield pretty fast convergence. Please let me know what your experience is if you experiment with this.

```
i pwls2 out.fld init.fld sino.fld - invvar.fld tomo.wtf - \
  @10@ca,0.6,raster1@-6,quad,1,b2info 0 1 1e9 1
```

(The backslash is a Unix way of splitting long lines.) Obviously you will want to create scripts rather than typing all that on the command line!

9.2 Penalized Likelihood: Transmission Case

The statistical model for transmission tomography is:

$$Y_i \sim \mathsf{Poisson}\Big\{ b_i \, \mathrm{e}^{- \sum_j g_{ij} x_j} + r_i \Big\},$$

where b_i is the blank scan or air scan rate (properly scaled for scan-time differences between blank scan and transmission scan), r_i is the background events (*e.g.*, random coincidences, scatter, or crosstalk), Y_i is the transmission measurement, x_j is the linear attenuation coefficient (units inverse length) of the jth pixel, and g_{ij} has units of length. I recommend estimating $\boldsymbol{x}$ by maximizing a penalized-likelihood objective. The differences between a reconstructed FBP attenuation map and a penalized-likelihood reconstruction can be very dramatic!

Typing `i trpl2` will show the arguments for 2D penalized-likelihood transmission reconstruction.

```
Usage: trpl2 out {init|-|0} yi bi- bi_scale ri- ri_scale wtf mask- method
[saver- flag_obj(0) flag_nonneg(1) pix_max scale_init(0) slices-]
```

Many of these arguments are identical to those for PWLS, so below I only describe the new ones.

- *yi* is the transmission scan sinogram $\boldsymbol{y}$.
- *bi* is the blank scan sinogram b_i.
- *bi_scale* is for scaling the blank scan by a constant, usually the ratio of the transmission scan time over the blank scan time. You could also include (relative) dead time effects here. Use `1` if the b_i's are already scaled by the relative scan times.
- *ri* is the sinogram of background events r_i. Default (if hyphen is used) is the scalar value *ri_scale*.
- *ri_scale* scales r_i by a constant.
- *flag_nonneg* should usually be `1` to enforce the nonnegativity constraint.

The *method* argument has the same syntax as for PWLS (see (2)). Since penalized-likelihood image reconstruction has been one of my favorite research topics, there are several different *algorithm*s that are supported. Typing `i trpl2` shows the whole set. For 2D PET and SPECT transmission scans, my current favorite (in terms of speed of convergence and monotonicity) is the paraboloidal-surrogates coordinate-ascent (PSCA) methd developed by Erdoğan [11]. This algorithm has several variations depending how one chooses the parabola curvatures.

Here are the possible choices for *algorithm* in the *method* string.

- `psca,od,1,raster1` uses the "optimal" curvature of [11] which ensures monotonicity. This algorithm cannot diverge!
- `psca,fd,1,raster1` uses the "fast precomputed" curvature of [11], which does *not* ensure monotonicity, but is *usually* monotonic anyway. I recommend you start with this approach, and then revert to the optimal curvatures if problems arise. (And tell me if they do!) This is what I use 99% of the time.

The `1` indicates a single subiteration of CA before updating the surrogates, which seems the most efficient approach. The `raster1` specifies that CA visits the pixels in conventional lexicographic ordering.

My favorite penalty function for transmission scans is the Huber penalty, because we know what the range of values is in transmission scans, so we can choose the breakpoint δ in the Huber function intelligently. The `penalty` string for a Huber function penalty with $\delta = 0.001$/mm with a 2nd-order neighborhood looks like the following.

log2beta,`huber,2,-,0.001,ih,3`

Type `i trpl2` to see more details.

9.3 OSTR

If you have a large sinogram and image, like in X-ray CT, or a slow computer (or just limited patience, like me), then even PSCA may seem to slow to you and you will want to try the *ordered-subsets transmission reconstruction* (OSTR) algorithm of Erdoğan [12]. *Please please do not apply the emission OSEM algorithm to transmission scans! It works poorly, as shown in [12].* To use the OSTR algorithm with 4 subsets, use the following *algorithm* string `osc,4,1.0`.

Unlike the OSEM world, which seems to be dominated by unregularized work, I *highly* recommend including good regularization with OSTR to get the best results. Here is an example of a complete *method* string for 5 iterations of OSTR with 4 subsets with the Huber penalty with $\beta = 2^{-6}$.

`@5@osc,4,1.0@-6,huber,2,-,0.001,ih,3`

9.4 Penalized Likelihood: Emission Case

The statistical model for emission tomography is:

$$Y_i \sim \mathsf{Poisson}\left\{ \sum_j a_{ij} x_j + r_i \right\}, \tag{3}$$

where r_i is the background events (*e.g.*, random coincidences, scatter, or crosstalk), Y_i is the emission measurement, x_j is the emission density of the jth pixel, and $a_{ij} = c_i g_{ij}$ where c_i are ray-dependent calibration factors (such as detector efficiency and PET photon absorption survival probabilities) and g_{ij} represents the geometric portion of the system matrix. I recommend estimating $\boldsymbol{x}$ by maximizing a penalized-likelihood objective. The fastest monotonic algorithm I know for performing this maximization is the PML-SAGE-3 algorithm [13], which I will document here along with the hideously slow ML-EM algorithm for comparison.

Typing `i empl2` will show the arguments for this method:

```
Usage: empl2 out init- yi ci- ri- ri_scale wtf mask- method
[saver- flag_obj(0) pix_max scale_init(0) slices-]
```

Again, most of these are identical to those for PWLS and `trpl2`, so below I only describe the new ones.

- *yi* is the "prompt" emission coincidences $\boldsymbol{y}$.
- *ci* is the factors c_i. Default is $c_i = 1$.
- *ri* is the background events r_i. Default (if dash is used) is *ri_scale*.
- *ri_scale* scales r_i by a constant.

The *method* argument has the usual syntax in (2). There are several choices for the *algorithm* argument. Type `i empl2` to see all the choices, since only some are described below.

9.4.1 ML-EM

`em,1` is the standard ML-EM-1 algorithm (which is unregularized, so just use a "`-`" for the *penalty*, *i.e.*,

`@30@ml,1@-`

would be the method argument for the ubiquitous 30 iterations of ML-EM.

9.4.2 SAGE

`sage,3,raster1` applies the PML-SAGE-3 algorithm [13], with a large collection of penalty functions available. For now, I recommend the penalty choice `quad,1,b2info` since it gives more uniform resolution than conventional regularization methods [2], and works with the β tabulation described in Section 6. In the near future I hope to document other choices due to the work of Stayman [3–5]; ask me if interested. I have reservations about applying nonquadratic penalty functions to emission data, but please go ahead and try for yourself; `i empl2` will list the many choices.

This implementation of SAGE assumes that $r_i > 0$. If you do not provide a `ri` file with positive elements, or if your `ri_scale` argument is 0, then ASPIRE 3.0 will print a warning and you will probably get a floating point exception due to division by zero. I cannot think of *any* real data that has a zero background: there is always randoms, scatter, or just room background. If you are using simulated data with no randoms or scatter, then just use a small number like `1e-5` for `ri_scale` and the effect should be negligible.

9.5 OSEM

Use `osemc,8` for OSEM with 8 subsets (with subsets chosen as far apart as possible, as recommended in the literature).

As much as it pains me, you are probably going to try OSEM on your data. And quite possibly you are going to "correct" your data for everything before so doing. Here is an example of how to use ASPIRE 3.0 for OSEM on precorrected data:

```
i empl2 out.fld - sino.fld - -shift 0.01 tomo.wtf - @10@osema,8@- - 1 1e9 0 -
```

This will run 10 full iterations of unregularized OSEM with 8 subsets on the `sino.fld` sinogram, starting from a uniform image. The `-shift 0.01` implements a very simple version of the *shifted Poisson* method described in [14–17]. Basically we add 0.01 to the Y_i's and also set the r_i's to 0.01 in (3). (If you have a better estimate of the mean random events per sinogram bin, you should use that instead.)

Since the OSEM method uses blocks of rays, it requires a `row` grouped system matrix. After generating the `col.wtf` using `wt gen`, call `wt row2col row.wtf col.wtf` to form a row-grouped system matrix `row.wtf` which should be passed to `i empl2` for OSEM.

9.6 OSDP: ordered subsets regularized modified EM algorithm of De Pierro

In 1995, De Pierro published a clever modified EM algorithm [18] for handling the regularized case. It seems not to have earned the attention by practitioners that it deserves. It inherits the slow convergence of ML-EM, but this can be largely overcome by applying the ordered-subsets principle.

Using `osdpc,8` invokes the regularized, ordered-subsets version of De Pierro's modified EM algorithm, aka OSDP for lack of a better acronym. *This is the method-of-choice* if you want both regularization *and* the fast "convergence" of OS algorithms. Like OSEM, OSDP will not converge in general, but we're working on fixing that [19].

9.7 Preprocessing for PWLS: Emission Case

As explained in [8], the PWLS cost function is for pre-processed data. To pre-process sinogram data from a CTI scanner, type:

```
op pre emis corr yicorr.fld nder2.fld yiraw.fld nrm.fld atn.fld 2 10 0
```

where the inputs are: `prompt.fld` is the measured coincidences, `nrm.fld` is the detector normalization factors, and `atn.fld` is the attenuation correction factors. The outputs: `pivot.fld` and `nder2.fld` are the corresponding inputs to `i pwls2`. Or better yet, use combination of `op mul`, `op sub`, and `op div`, to apply the corrections yourself so that you know exactly what is going on. *However*, thanks to the shifted-Poisson developments [14–17], I virtually never use the PWLS approach of [8] anymore.

9.8 Shifted Poisson statistical model

PET measurements are usually precorrected for accidental coincidences, destroying the Poisson statistics. Yavuz [14–17], showed that such precorrected measurements can be well approximated by Poisson statistics *if* they are appropriately "shifted" so that the mean matches the variance.

In practice, the ideal shift $(2r_i)$ for each sinogram bin is unknown. Remarkably, however, Yavuz showed that even a uniform constant shift works well in practice [17]. A principled approach would be to look up the total number of delayed coincidences in the PET sinogram file header, and then divide this by the number of sinogram bins, and then multiply by 2 and use that as the shift factor. In practice I am usually lazier than that. I just shift by a small value like 3 or 4 counts. This will usually eliminate 90% of the negative values in a typical PET body scan.

To apply such a shift in `i empl2` or `i trpl2`, simply replace the *ri* argument with `-shift` and the *ri_scale* argument with the scalar value to be used for the shift (*e.g.*, 4).

Since precorrection for accidental coincidences is one of the biggest discrepancies between the "theory" of PET reconstruction (which usually is based on the Poisson model) and the routine practice of PET, I highly recommend that you at least skim the papers by Yavuz to see how the shifted Poisson model bridges this gap.

10 Examples

Here is a complete and tested example of using `op`, `wt`, and `i` to generate simulated transmission measurements and reconstruct via FBP and penalized likelihood. The very last line of the script converts the output images to postscript, and these very postscript figures are shown in Fig. 1. You should be able to cut-and-paste these lines from the PDF file in Acrobat reader (or ask me to email them to you) so that you can replicate this "test." By exploring the built-in documentation in these programs you will discover the wealth of features available. This script is only about 60 lines long, including blank lines and comments, and goes from synthesizing the phantom and scans through reconstruction and display.

Happy reconstructing!

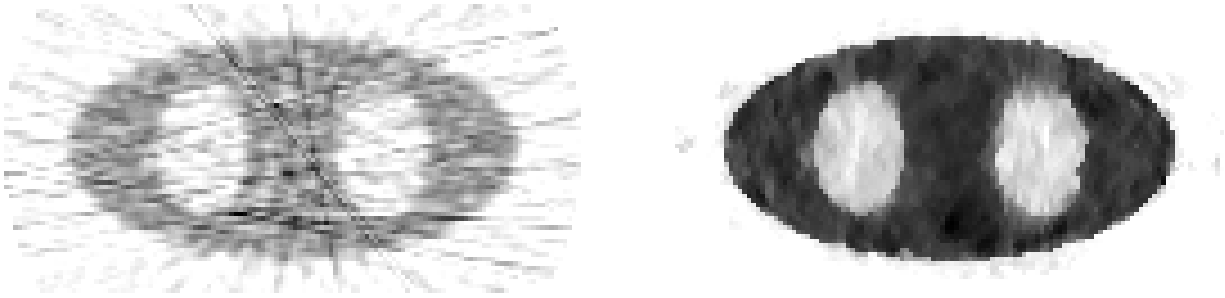

Figure 1: Results of a low-count transmission simulation. The left image is FBP, the right image is penalized-likelihood.

```
#!/bin/csh
# demo,tran
# demonstrate simulated-PET transmission reconstructions using ASPIRE

# generate system description file and system matrix files
# (xrad and yrad set the support ellipse radii)
if !(-e tomo.dsc) then
        wt -chat 0 dsc 2 nx 128 ny 64 nb 160 na 192 \
                pixel_size 4.2 ray_spacing 3.4 strip_width 3.4 \
                scale 0 xrad 62 yrad 30 tiny 0 >! tomo.dsc
endif
```

```
if !(-e tomo.wtf) wt gen tomo.dsc
if !(-e tomo.wtr) wt col2row tomo.wtr tomo.wtf

# make "true" thorax-like attenuation map out of ellipses
if !(-e mumap.fld) then
        op ellipse mumap.fld 128 64 \
                0 0     50 25   0 0.01 3 \
                20 0    10 15   0 -0.008 3 \
                -20 0   10 15   0 -0.008 3 \
                0 -15   5 5     0 0.003 3
endif

# transmission noiseless sinogram
if !(-e proj.fld) i -nthread 2 proj2 proj.fld mumap.fld tomo.wtf

# create blank scan with artificial nonuniform detector efficiencies
if !(-e bi.fld) \
        op sim blank trues.fld bi.fld proj.fld 0.3 5e5 0

# noisy transmission scan with 5% precorrected accidental coincidences
if !(-e yi.fld) \
        op sim pet yi.fld - trues.fld   1 0 - 5 -1 1

# FBP reconstruction, followed by setting negatives to zero
set fbpwin = gauss,1,10,1
if !(-e tfb.fld) then
        i fbp2t dsc tfb.fld - yi.fld bi.fld 1 - 0 tomo.dsc $fbpwin -
        echo y | op nonlin max tfb.fld tfb.fld 0 0
#       j --red -a tfb.fld mumap.fld    # compare fbp to true
endif

# determine support mask from fbp image
if !(-e mask.fld) then
        op pre attn mask mask.fld tfb.fld 0.001 1 "f e1-+3,3 d2-+3,3"
        # make sure within .wtf support
        i support t0 tomo.wtf
        echo y | op mul mask.fld mask.fld t0
#       j -s -m2 t0 mask.fld mumap.fld
endif

#set alg = psca,fd,1,raster1
set alg = osc,24,1.0                    # OSTR algorithm
set penal = 18,huber,2,-,0.0002,ih,3    # Huber penalty
set method = @8@$alg@$penal

# penalized-likelihood transmission reconstruction
if !(-e tpl.fld) then
        set flag_obj = 1
        i trpl2 tpl.fld tfb.fld yi.fld bi.fld 1 -shift 3 tomo.wtr mask.fld \
                $method - $flag_obj 1 1e9 0 -
#       j --red -a tpl.fld tfb.fld mumap.fld
endif

# make eps files from final figures
```

```
op -chat 0 eps tfb.eps tfb.fld 72 72 144 1 0
op -chat 0 eps tpl.eps tpl.fld 72 72 144 1 0
```

11 General options

11.1 Verbosity

All three programs produce a fair amount of "chatter." To eliminate the chatter, use

`op -chat 0` *arguments*

You will want to do this when piping the output into a file, such as with `op ascii` *filename*.

Before reporting any bugs, it is helpful for you to *increase* the chatter and email me *all* of the output

`i -chat 999` *arguments*

Obviously, bigger numbers means more chatter.

11.2 Threads

I am starting to support POSIX threads for the more computationally intense routines such as forward and backprojection. If you have a multi-core computer, then the invocation

`i -nthread 2` *arguments*

will tell ASPIRE 3.0 to try to run 2 threads, which should nearly halve your execution time in many cases. The option is harmless when applied to routines that are not thread-enabled, so it cannot hurt if you have a dual-processor machine. (Of course, if you are *sharing* a dual-processor machine with others, then you will now be using both processors, which may affect your popularity.)

If you have a one-processor machine, then invoking 2 threads will incur a slight operating system overhead, because one processor will have to serve both threads.

A Geometry descriptions

ASPIRE 3.0 supports several system geometries, including the following.

- Spatially-invariant image domain blur (for image restoration).
- Parallel tomographic geometry with uniformly spaced strip- or line-integrals and uniformly spaced angles.
- Fan-beam tomographic geometry with equi-detector spacing (for fan-beam collimated SPECT systems and flat-panel X-ray CT systems).
- Depth-dependent blur (2D) for SPECT.

Complications like the circular geometry in PET are partially implemented, but not documented because they have not been adequately tested.

A.1 Common properties

The user specifies the relevant properties of the system geometry of interest in an ASCII description file that, by convention, has the suffix `.dsc`. There are some properties that are common to all `.dsc` files.

- Each `.dsc` file must include a line of the form

  `system` *system_number*

  where *system_number* could be one of several integers, indicating which type of system geometry is described in the file. To see a list of all the system geometry types, enter `wt gen`. (The integers reflect the historical order in which the system types were implemented.)
- Comment lines are allowed, and must begin with a # sign.
- Each characteristic must be on a separate line.
- There will always be lines of the form

```
system  0
nx      6
ny      4
support all
scale   1
```

  - `nx` and `ny` are the number of image columns and rows.
  - `scale` is an optional scaling factor applied to all elements

  If the above `support all` line is used (not recommended!) then weights are generated for *all* pixels, even though some of the pixels at the corners of the FOV may have partly truncated projections in the sinogram. Instead, I recommend something like the following:

```
support ellipse 0 0 62 57
```

  In this case, weights are generated *only* for pixels lying wholly within an ellipse centered (in this example) at (0,0) (dead center of the pixel matrix) having horizontal,vertical radii 62,57 pixels (in this example). This saves lots of memory, but you must make sure the ellipse is big enough! The default (if no `support` line) is a centered ellipse with radii (`nx`/2-2), (`ny`/2-2).

  There is also a "`support file` *file*" option if you want an irregular support specified in some binary file.

A.2 Spatially-invariant image restoration

In this case, the $\boldsymbol{G}$ matrix represents a discrete 2D convolution with some space-invariant point spread function. A typical `.dsc` file for this case looks like:

```
system  0
nx      6
ny      4
support all
scale   1
psf     5       3
0 0 1 0 0
2 3 4 3 2
0 0 1 0 0
```

Here,

- `system 0` indicates the image restoration geometry.
- `nx` and `ny` are the number of image columns and rows. These should be even integers.
- The two digits following `psf` give the size of the support of the PSF. These must be odd integers.

- The next $15 = 5 \cdot 3$ entries are real numbers representing the PSF.

A.3 Parallel strip-integral geometry

The measurements from many tomographic instruments can be approximated by line-integrals or strip-integrals. In this case, element g_{ij} of $\boldsymbol{G}$ is proportional to the area of intersection between the jth pixel and the ith strip (Fig. 2). I very strongly recommend strip-integrals over line-integrals.

The absolutely most minimal `.dsc` file you could use looks like the following:

```
system  2
nx 64
nb 80
na 60
support all
```

This generates weights for a 64×64 image projected onto a sinogram with `nb`=80 radial samples whose spacing (and width) is the same as the pixel size, and `na`=60 angular samples distributed over 180°. The "`system 2`" line indicates the parallel strip/line integral geometry.

An example of a `.dsc` file that uses virtually all of the options of system 2 is the following.

```
#       931,thorax,emis,2.dsc
system  2
nx      128
ny      128
nb      192
na      256
support ellipse 0 0 62 57
orbit           180
orbit_start     0
bin_min 18
bin_max 174

offset_even     0.5
offset_odd      0.5
center_x        -0.5
center_y        0.5
flip_y          -1

pixel_size      4.69398
ray_spacing     3.12932
strip_width     3.12932
scale           0
```

(I use this one for reconstructing PET thorax images from a CTI 931 scanner). Here is an explanation of the arguments.

- The image dimensions are `nx` by `ny`. If not specified, `ny` defaults to `nx`.
- The sinogram dimensions are `nb` (radial bins) by `na` (angles).
- The uniformly-spaced projection angles are computed in degrees as:

$$\texttt{orbit\_start} + \texttt{orbit} \cdot i/\texttt{na},$$

  for $i = 0, \ldots, (\texttt{na} - 1)$. I use 0° as straight up along the y axis, and `orbit_start` adds a counter-clockwise angular offset. Defaults are 0 and 180.

- Weights are generated *only* for pixels lying wholly within an ellipse centered (in this example) at (0,0) (dead center of the pixel matrix) having horizontal,vertical radii 62,57 pixels (in this example). This saves lots of memory, but you must make sure the ellipse is big enough! The default (if no `support` line) is a centered ellipse with radii (`nx`/2-2), (`ny`/2-2).

- Only sinogram bins in the range [`bin_min, bin_max`) = [18,174) are used (they are 0 outside of this on our CTI 931 due to our normalization method). You probably should not include this option; the defaults are `bin_min` = 0 and `bin_max` = `nb`.

- Normally the pixel matrix dead center would be the center of rotation, but I have found that CTI images are off by half a pixel, so for consistency I use `center_x -0.5` and `center_y 0.5`. These have units of pixels, and you can use other values, but you will have to experiment to determine if you need positive or negative shifts. Defaults are 0.

- In many tomographs (such as SPECT with proper center-of-rotation correction), dead-center of the image will project dead-center on the sinogram. Due to the interleaving of the projections by CTI, there are half-bin offsets, hence the `offset_...` lines. Again, whether it is left or right is too painful to document. Defaults are 0.

- CTI images also seem to be upside down relative to my coordinate system, so I use `flip_y = -1.` The default is `1`, which does no flipping. You could also use other values if you wanted to stretch or shrink the vertical direction, but you probably do not want to do that.

- The `pixel_size` is the width of each pixel (in any units, but the units of `ray_spacing` and `strip_width` must match). Default is 1.

- `ray_spacing` is the center-to-center spacing of the radial samples. Defaults to `pixel_size`.

- `strip_width` is the width of the strip, which should usually not be smaller than `ray_spacing`, or you will have gaps between your strips. For a 931, it might be more realistic to use set `strip_width` to about 6mm, which is approximately the detector width. If you set `strip_width` to 0, then you will get line integrals, and probably lousy images. Defaults to `ray_spacing`.

- Using the argument `scale 0` causes strip-integral areas to be normalized by the strip width, so the g_{ij}'s will have the same *units* as line-integrals, and the reconstructed pixel values will have units of inverse length, which is exactly what is needed for transmission tomography. These days I use the same choice for emission tomography, even though there the natural units are something like counts per unit area, because to get absolute quantification in emission tomography one must apply some type of global scale factor based on a well counter measurement, and this usually done after reconstruction. In principle, you could use `scale` to include scalar effects such as deadtime, decay, etc., although personally I would include those somewhere else. The default `scale` value is `1` for historical reasons, so I strongly recommend over-riding this default by explicitly choosing `scale 0`. That way you can use the same `.wtf` for both transmission and emission reconstruction.

A.4 Fan-beam geometry

Several groups now have line sources opposing fan-beam collimators for SPECT transmission scans. Because of how the Anger camera works, this geometry corresponds to the "equally spaced detectors" version of fan-beam data. Here is an example of a complete `.dsc` file for this geometry:

```
system  8
nx      64
ny      64
nb      110
```

```
na      60
support ellipse 0 0 30 30
orbit           360
orbit_start     0
pixel_size      7.12
ray_spacing     3.56
strip_width     3.56
src_det_dis     650
obj2det_x     219
obj2det_y     219
```

Most of the arguments have the same meaning as above. The idea of a "strip" here is an approximation, since if you think of the source as a point, and the detector as having a certain width, then the beam is more of a very thin triangle than a rectangular strip. However, the obliqueness of the beam over the size of a pixel is usually negligible, so we simply approximate the triangle locally by a rectangle. Anyway, the final units will be inverse length, since this geometry is only for transmission imaging.

- `obj2det_x, obj2det_y` denotes the distance from the center of rotation to the detector plane in x and y directions (for elliptical orbit).
- `src_det_dis` denotes the distance from source to the detector (*e.g.*, 650 mm focal length, give or take the thickness of the collimator).

Note that these last two are changed from an earlier version!

I have concerns about the accuracy of this approximation. I recommend using `system 13` instead which has the same options (plus more, see `wt dsc 13`). The `system 13` version does an *exact* analytical calculation of the area of intersection between the wedge and the pixel.

If you have an "arc" detector geometry, like 3rd generation CT systems, then use `system 14` which has similar arguments. (Bug me for more documentation).

A.5 Depth-Dependent Gaussian Blur for 2D SPECT

This system matrix assumes the PSF for SPECT has a Gaussian shape with the following model for FWHM:

$$\mathrm{FWHM} = \sqrt{(z\mathrm{FWHM}_s + \mathrm{FWHM}_0)^2 + \mathrm{FWHM}_d^2},$$

where z is the distance from a pixel's center to the detector, FWHM_d is the intrinsic spatial resolution of the detector (often about 3mm), FWHM_0 is a constant that partially determines the FWHM for a point source adjacent to the collimator, and FWHM_s is the "slope" of the FWHM versus depth.

An example of a `.dsc` file that uses all of the options of system 12 is the following.

```
system  12
nx      64
ny      64
nb      68
na      60
support ellipse 0 0 30 30
orbit           180
orbit_start     0
bin_min 0
bin_max 64
```

```
pixel_size      7.2
ray_spacing     7.2
scale           7.2

obj2det_x     219
obj2det_y     219

fwhm_detector           3.2
fwhm_collimator0        1.76
fwhm_slope              0.0568
fwhm_factor             1
```

Many of the arguments are the same as for system 2. I only explain those that differ.

- `scale` must be nonzero (no "transmission" scaling). Default is 1.
- `obj2det_x`, `obj2det_y` denote the distance from the center of rotation to the detector plane in x and y directions (for elliptical orbit).
- `fwhm_detector` specifies the FWHM of the intrinsic detector response, in the same units as `pixel_size` and `ray_spacing`.
- `fwhm_collimator0` specifies FWHM_0 in the above equation, also in the same units as `pixel_size`.
- `fwhm_slope` specifies the FWHM_s term in the above equation, which must be unitless.
- `fwhm_factor` specifies how far out to sample the Gaussian on each side of the peak. If you use the default, which is `fwhm_factor 1`, then it will be sampled 1 FHWM on each side, which covers 0.98% of the area. The software automatically corrects scales up the g_{ij}'s for each pixel for each angle so that no counts are lost.

If you are serious about SPECT reconstruction with compensation for depth-dependent blur, then you probably really want the full restoration provided in the 3D reconstruction method `i empl3`. See [6].

B AVS data format

The AVS `.fld` data format comes in two flavors. In the "internal" format, the ASCII header is at the top of the file, the header is followed by two "form-feed" characters, which are then followed by the data in binary format. Form feed characters often appear as `(^L)` in Unix, and are created using the `'\f'` character in C.

In the "external" format, the header and the data are in separate files, and the ASCII header file includes a pointer to the data file. The data file can contain either ASCII or binary data.

Suppose you have a 128×64 (first dimension (radial samples) varies fastest) sinogram consisting of short integers. Then the format of the "internal" header would be:

```
# AVS field file
ndim=2
dim1=128
dim2=64
nspace=2
veclen=1
data=short
field=uniform
```

followed by the two form feeds and then the 128×64 short integers in binary format. If you have a 3D stack of, say, 20 sinograms (or images), then you would use

```
# AVS field file
ndim=3
dim1=128
dim2=64
dim3=20
nspace=3
veclen=1
data=short
field=uniform
```

ASPIRE 3.0 supports up to 4 dimensions.

All output data from ASPIRE 3.0 is stored in the "internal" format. *AVS filenames must end with the extension* `.fld`.

Now suppose you have stored the above sinogram data in a binary file named, say, `sino.dat` with some home-brew header in it that consists of, say, 1999 bytes. And suppose you do not want to convert from home-brew format to "internal" format. Then you can use the "external" format by creating an ASCII file named, say, `sino.fld` containing:

```
# AVS field file
ndim=2
dim1=128
dim2=64
nspace=2
veclen=1
data=short
field=uniform
variable 1 file=sino.dat filetype=binary skip=1999
```

You can add additional comments to these headers using lines that begin with `\#`. The `skip=1999` indicates that there is a 1999 byte header to be skipped before reading the binary data. *This format does not allow for additional headers buried within the data, so you cannot usually read Siemens/CTI format data without doing some file conversion, since their format includes embedded directories.* (Complain to CTI.) If there is no binary header, then you can omit the `skip=0` altogether. If your data is in ASCII format (I hope not), then you can change `filetype=binary` to (you guessed it) `filetype=ascii`. However, for ASCII data, the `skip=` option refers to ASCII entries, not bytes.

The allowed types in the `data=...` line include: `byte`, `short`, `int`, `float`, `double`. The `byte` format is unsigned 8 bits. The output from ASPIRE 3.0 is virtually always of the `float` variety. Your input data can be any of the above; ASPIRE 3.0 will convert to whatever type it needs internally.

More information about the AVS program is available from `http://www.avs.com/`. Personally, I do not use AVS much anymore because it does not support "batch" processing very well, but it was useful in the early stages of my software development, when interactive use was more important. The complete AVS `.fld` format includes other features which almost certainly are not supported by ASPIRE 3.0.

Actually, I have added other features to ASPIRE 3.0 that are nonstandard AVS but very handy, like a single 3D header file that points to multiple 2D files that get treated as a single entity. Ask me if interested!

C Information for developers

To access the internal subroutines of ASPIRE 3.0, compile using

```
cc -c -Dnomainwt wt.c
```

which will create `wt.o`, which you then combine with your own `main` routine:

```
cc -o myprog mymain.c wt.o -lm
```

Here is a fragment of code that illustrates how to use ASPIRE 3.0 to read in a weight file and calculate $G'Gx$ for an image.

```
void example_function(float *proj, float *image, char *file_wtf)
{
        void *sp;

        /*
        * Read weight file
        */
        if ( !(sp = sp_read_file(file_wtf, NULL, 0)) )
                Fail("error reading file")

        /*
        * forward projection
        */
        sp_project(proj, image, sp, 0);

        /*
        * back projection
        */
        sp_back_project(image, proj, sp);

        /*
        * Free
        */
        sp_free(sp);
}
```

The above fragment is enough information to implement most of the popular reconstruction algorithms (ML-EM, WLS-CG, etc.). To implement the coordinate-ascent algorithms efficiently you need several more routines which are present in ASPIRE 3.0 but are not documented here. You could figure some of them out by examining the subroutines `sp_project` and `sp_back_project` in `wt.c`.

D Acknowledgement

The author gratefully acknowledges support of grants from DOE, NIH, and the Whitaker foundation. He also is very grateful to Christian Michel for help decoding the CTI file formats. Portions of the I/O code used for CTI scan files are based on software from Michel's ftp site. The author also thanks numerous colleagues and students for their feedback and bug reports etc.

References

[1] J A Fessler and W L Rogers. Uniform quadratic penalties cause nonuniform image resolution (and sometimes vice versa). In *Proc. IEEE Nuc. Sci. Symp. Med. Im. Conf.*, volume 4, pages 1915–1919, 1994.

[2] J A Fessler and W L Rogers. Spatial resolution properties of penalized-likelihood image reconstruction methods: Space-invariant tomographs. *IEEE Tr. Im. Proc.*, 5(9):1346–58, September 1996.

[3] J W Stayman and J A Fessler. Regularization for uniform spatial resolution properties in penalized-likelihood image reconstruction. *IEEE Tr. Med. Im.*, 19(6):601–15, June 2000.

[4] J W Stayman and J A Fessler. Penalty design for uniform spatial resolution in 3d penalized-likelihood image reconstruction. In *Proc. of the 1999 Intl. Mtg. on Fully 3D Im. Recon. in Rad. Nuc. Med.*, 1999.

[5] J W Stayman and J A Fessler. Nonnegative definite quadratic penalty design for penalized-likelihood reconstruction. In *Proc. IEEE Nuc. Sci. Symp. Med. Im. Conf.*, 2001. To appear.

[6] J A Fessler. Users guide for ASPIRE 3D image reconstruction software. Technical Report 310, Comm. and Sign. Proc. Lab., Dept. of EECS, Univ. of Michigan, Ann Arbor, MI, 48109-2122, July 1997. Available from `http://www.eecs.umich.edu/~fessler`.

[7] J A Fessler. Resolution properties of regularized image reconstruction methods. Technical Report 297, Comm. and Sign. Proc. Lab., Dept. of EECS, Univ. of Michigan, Ann Arbor, MI, 48109-2122, August 1995.

[8] J A Fessler. Penalized weighted least-squares image reconstruction for positron emission tomography. *IEEE Tr. Med. Im.*, 13(2):290–300, June 1994.

[9] K Sauer and C Bouman. A local update strategy for iterative reconstruction from projections. *IEEE Tr. Sig. Proc.*, 41(2):534–548, February 1993.

[10] S D Booth and J A Fessler. Combined diagonal/Fourier preconditioning methods for image reconstruction in emission tomography. In *Proc. IEEE Intl. Conf. on Image Processing*, volume 2, pages 441–4, 1995.

[11] H Erdoğan and J A Fessler. Monotonic algorithms for transmission tomography. *IEEE Tr. Med. Im.*, 18(9):801–14, September 1999.

[12] H Erdoğan and J A Fessler. Ordered subsets algorithms for transmission tomography. *Phys. Med. Biol.*, 44(11):2835–51, November 1999.

[13] J A Fessler and A O Hero. Penalized maximum-likelihood image reconstruction using space-alternating generalized EM algorithms. *IEEE Tr. Im. Proc.*, 4(10):1417–29, October 1995.

[14] M Yavuz and J A Fessler. Objective functions for tomographic reconstruction from randoms-precorrected PET scans. In *Proc. IEEE Nuc. Sci. Symp. Med. Im. Conf.*, volume 2, pages 1067–71, 1996.

[15] M Yavuz and J A Fessler. New statistical models for randoms-precorrected PET scans. In J. Duncan and G. Gindi, editors, *Information Processing in Medical Im.*, volume 1230 of *Lecture Notes in Computer Science*, pages 190–203. Springer Verlag, Berlin, 1997.

[16] M Yavuz and J A Fessler. Statistical image reconstruction methods for randoms-precorrected PET scans. *Med. Im. Anal.*, 2(4):369–378, 1998.

[17] M Yavuz and J A Fessler. Penalized-likelihood estimators and noise analysis for randoms-precorrected PET transmission scans. *IEEE Tr. Med. Im.*, 18(8):665–74, August 1999.

[18] A R De Pierro. A modified expectation maximization algorithm for penalized likelihood estimation in emission tomography. *IEEE Tr. Med. Im.*, 14(1):132–137, March 1995.

[19] S Ahn and J A Fessler. Globally convergent ordered subsets algorithms: Application to tomography. In *Proc. IEEE Nuc. Sci. Symp. Med. Im. Conf.*, 2001. To appear. MI-2.

Most of my papers are available on WWW: `http://www.eecs.umich.edu/~fessler`

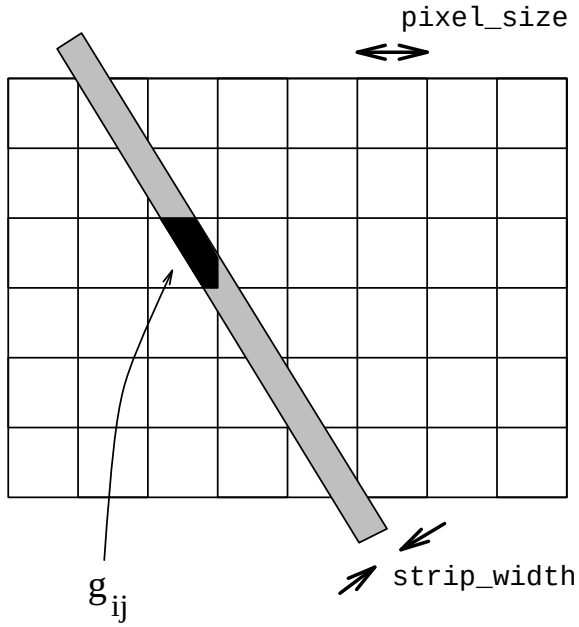

Figure 2: Strip integral.