Dynamic libraries for ASPIRE penalty functions and system models

Jeffrey A. Fessler

June 19, 2003

1 Introduction

Using any of a large variety of algorithms for any of several tomographic imaging problems, ASPIRE can perform image reconstruction by finding the minimizer of cost functions of the form

$$\Psi(\boldsymbol{x}) = J(\boldsymbol{A}\boldsymbol{x}) + R(\boldsymbol{x}),$$

where $\boldsymbol{A}$ is a "system matrix" that describes the physics of the imaging system, J is a data-fit term such as the negative log-likelihood or a weighted squared error, $R(\boldsymbol{x})$ is a roughness penalty, and $\boldsymbol{x}$ is the image (or volume) to be estimated.

ASPIRE now allows a user to provide custom routines for forward and backprojection, *i.e.*, multiplication by $\boldsymbol{A}$ or $\boldsymbol{A}'$, as well as for providing custom routines for the penalty function $R(\boldsymbol{x})$ and its gradient $\nabla R(\boldsymbol{x})$. The user provides these routines in dynamically loaded library files. (These are files that end in `.so` in unix, or `.dll` in some other inferior operating systems.) By this mechanism, ASPIRE does not need to be recompiled while the user debugs his or her routines. Likewise when a new release of ever-improving ASPIRE is available, the user need not recompile her routines, provided the prototypes do not change, which they probably will not. Well, hopefully not anyway. No promises.

2 Dynamic libraries for ASPIRE system models

This section describes the routines that must be provided by the user in the dynamic library for a user-defined system model $\boldsymbol{A}$. I say "model" rather than "matrix" because even though we write $\boldsymbol{A}$ on paper to denote a matrix, in most applications of this software, all that is needed is the ability to perform multiplications like $\boldsymbol{A}\boldsymbol{x}$ and $\boldsymbol{A}'\boldsymbol{y}$. Such operations can be performed by "forward projection" and "back projection" subroutines without ever explicitly constructing or storing a matrix $\boldsymbol{A}$.

All four of the routines described below must be provided; if you only need the forward projector, then a "do-nothing" backprojector can be supplied.

Often, rather than writing a forward / back projector "from scratch," you will want to modify an existing ASPIRE system model. For example, you might want to add the effects of scatter to the geometric effects computed by ASPIRE. This is easy to do since the user-defined projector routine can call any ASPIRE routine. This capability is illustrated in the example provided.

Most, if not all, of the algorithms supplied by ASPIRE were derived under the assumption that the backprojector $\boldsymbol{A}'$ is the exact transpose of the forward projector $\boldsymbol{A}$. Users who supply "mismatched" forward and backprojectors should be prepared for unexpected results.

The prototypes for the routines are defined in Appendix A.

Working (and tested!) examples for the four routines are available in `sys,dynlib,example.c`.

Here is what you must do.

- Write four routines, presumably by modifying the examples. Probably you want to start by using the examples unmodified to see if they work as well for you as they do for me. There is a test script `test,3u` provided that illustrates the use of a user-defined routine that does nothing other than call the ASPIRE routine!

- Compile the four routines into a dynamic library. This requires special instructions to the compiler to tell it to create a dynamic library with certain routines available to the outside world. For gcc/linux the file `linux-function.map` is needed. See the sample `Makefile` provided.

- Test your routines using

  `i proj3` *`data-out.fld image-in.fld`*
  `3u@`*`n1,n2,n3@mylibrary.so.1@user_args`*

  where *mylibrary.so.1* is the name of the dynamic library you compiled, *n1,n2,n3* are the data dimensions, and *user_args* are any optional arguments. Again, see `test,3u` for an example.

  For more thorough testing, type `i proj3` to see its additional optional arguments and also test the "subset" capability of your routines if relevant.

  Also testing the backproject using `i back3` would be wise before attempting iterative reconstruction.

- Finally, run `i pwls3` or `i empl3` or `i trpl3`, but replace the usual *sys_type* with

  `3u@`*`n1,n2,n3@mylibrary.so.1@user_args`*

  as the argument. No @s in the library name please.

2.1 Initialization: `sys_init`

Prior to iterating, ASPIRE will call the function `sys_init`. In this routine you can initialize any static variables that will be needed in the other routines. For example, you can read in files describing the tomographic geometry, allocate work memory, etc.

You can also parse the optional user arguments string, for example to extract parameters related to the system. See the example routine.

2.2 Cleanup: `sys_free`

Before ASPIRE exits, it will call the function `sys_free`. Here you should free any variables you allocated, if any. Why bother? Because it may help find bugs. By request, ASPIRE can be compiled with memory allocation testing features that make sure that every allocated pointer is freed.

2.3 Forward projector: `sys_proj`

In its simplest version, this routine must calculate $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, for a given $\boldsymbol{x}$, into an array $\boldsymbol{y}$.

The support `mask` is provided to this routine so that one can (sometimes) save computation by ignoring voxels outside this support. All voxel values that are outside of the support mask will be zero. The vector $\boldsymbol{y}$ will not be initialized to zero prior to calling `sys_proj`, so if your routine works by "incrementing" $\boldsymbol{y}$, then you are responsible for initializing $\boldsymbol{y}$ to zero.

A structure is also passed to this routine that controls which projection subsets are to be computed.

2.4 Gradient: `sys_back`

This routine must calculate $\boldsymbol{b} = \boldsymbol{A}' \operatorname{diag}\{w_i\} \boldsymbol{y}$, where $\boldsymbol{y}$ and the w_is are passed to the routine. In other words,

$$b_j = \sum_i a_{ij} w_i y_i.$$

The output vector $\boldsymbol{b}$ *will* be initialized to zero before calling your routine.

Again, a structure is also passed to this routine that controls which projection subsets are to be backprojected.

2.5 Subsets

The header file `def,subset.h` describes the subset control structure. For now, only uniformly space views (`SubsetUnif`) are implemented (or at least documented). The structure member `istart` indicates which is the starting view for the current subset; it will be an integer in the set $\{0, 1, 2, \ldots,$ `nsubset` $- 1\}$. The value of `nsubset` is specified by the user elsewhere as part of the argument describing the iterative `method`.

The trick part is determining which portions of the data vector correspond to the relevant subset. In fact (unfortunately) it depends on the type of system model.

2.5.1 2z system model

For a `2z` (or `2dsc`) system model, the data is `nb` $\times$ `na` $\times$ `nz`, where the number of radial bins varies fastest, then the number of projection angles, then the number of slices. So subsets are taken over the *second* dimension (`na`) in this case.

To set all of the data `yi` for this subset to zero, use code like the following, where `ss` is a pointer to the subset control structure that is passed into your `sys_proj` routine.

```
const int istart = ss->istart;
const int nsubset = ss->nsubset;
const int nx = n1_global;
const int ny = n2_global;
const int nz = n3_global;
for (iz=0; iz < nz; ++iz)
 for (ia=istart; ia < na; ia += nsubset)
  for (ib=0; ib < nb; ++ib)
   yi[ib + ia*nb + iz*na*nb] = 0
```

This probably is not a useful operation, but is intended to illustrate how the data is organized so that you can operate on the correct subset.

2.5.2 3s SPECT system model

For the `3s` system model for SPECT, and for some of the 3D PET system models, the data is of dimension `nu` $\times$ `nv` $\times$ `nview`, where each of the `nview` projection views is `nu` $\times$ `nv`. In this case, subsets are taken over the *third* dimension of the data! In this case the code fragment would be as follows.

```
const int istart = ss->istart;
const int nsubset = ss->nsubset;
```

```
const int nu = n1_global;
const int nv = n2_global;
const int nview = n3_global;
for (ia=istart; ia < nview; ia += nsubset)
 for (iu=0; iu < nu; ++iu)
  for (iv=0; iv < nv; ++iv)
   yi[ib + ia*nb + iz*na*nb] = 0
```

If this seems too confusing, then I recommend that you first code the 1-subset version of your projector and debug it, and then I should be able to help with the transition to ordered subsets.

3 Dynamic libraries for ASPIRE penalty functions

The following sections describe the routines that should be provided by the user in the dynamic library for a user-defined penalty function. If all four are not provided, ASPIRE will try to use built-in defaults, which probably will produce erroneous results since they will be incompatible with the user-defined routines. So please provide all routines!

The prototypes for the routines are defined in Appendix A.

Working (and tested!) examples for the four routines are available in `rp,dynlib,example.c` for the 1st-order standard quadratic penalty function described [1].

Here is what you must do.

- Write four routines, presumably by modifying the examples. Probably you want to start by using the examples unmodified to see if they work as well for you as they do for me.
- Compile the four routines into a dynamic library. See the sample `Makefile` provided.
- Test your routines using

  `i r3,test user:`*`mylibrary.so.1`*`:`*`user_args`*

  where *mylibrary.so.1* is the name of the dynamic library you compiled and *user_args* are any optional arguments. You can test example routines by

  ```
  i r3,test user:example,sol2.so.1:-3,-4
  ```

  where $\beta_{xy} = 2^{-3}$ and $\beta_z = 2^{-4}$ are the regularization parameters in this example. For the example routines with the above invocation, the output from the test routine includes the following

  ```
  iz=2    p0=13.4554 pp=13.4959 pm=13.4205
  dx=0.37685  d0=0.376849  dp=0.404975 dm=0.348724
  (-1.39631e-05%)
  ```

  This routine checks to see if the supplied gradient routine agrees with numerical differencing. In this case the two agreed within $2{\cdot}10^{-5}\%$, which means there probably is no inconsistency.
- Finally, run `i pwls3`, but replace the usual *penalty* argument in the method string with

  `user:`*`mylibrary.so.1`*`:`*`user_args`*

  as the argument. No colons in the library name please.

3.1 Initialization: `rp_init`

Prior to iterating, ASPIRE will call the function `rp_init`. In this routine you can initialize any static variables that will be needed in the other routines. For example, you can read in a file containing penalty information. You can also parse the optional user arguments string, for example to extract the regularization parameters. See the example routine.

3.2 Cleanup: `rp_clean`

Here you get to free any variables you allocated, if any. Why bother? Because it may help find bugs.

3.3 Penalty: `rp_penalty`

This routine must calculate $R(\boldsymbol{x})$, a nonnegative penalty value. *Only quadratic penalties are currently supported.* You can probably ignore the support `mask`. All image values that are outside of the support mask will be zero, unless your gradient routine has a bug.

3.4 Gradient: `rp_grad`

This routine must calculate $\nabla R(\boldsymbol{x})$. For a quadratic penalty, $R(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}'\boldsymbol{R}\boldsymbol{x}$ for some nonnegative definite matrix $\boldsymbol{R}$. Thus $\nabla R(\boldsymbol{x}) = \boldsymbol{R}\boldsymbol{x}$. If your penalty is of the form

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

where $w_{jk} = w_{kj}$, then

$$\frac{\partial}{\partial x_j} R(\boldsymbol{x}) = \sum_k w_{jk}(x_j - x_k).$$

In the example, $w_{jk} = \beta_{xy}$ if pixel j and k are in-plane horizontal or vertical neighbors, and $w_{jk} = \beta_z$ if pixel j and k are immediately above or below in adjacent slices, and $w_{jk} = 0$ otherwise.

Your routine must only compute $\frac{\partial}{\partial x_j} R(\boldsymbol{x})$ for pixels j that are within the support mask, just like in the example routine. For the other pixels, the gradient must be left at zero.

The gradient routine is used by the PCG algorithms.

3.5 1D Newton Step: `rp_newton1`

If you want coordinate descent rather than CG (*e.g.*, to get a nonnegativity constraint), then you will need to provide this routine. It is more work to explain. Maybe you can figure it out from the provided example? Currently you will just get an innocuous warning that it could not find this routine. Innocuous unless you try to run coordinate descent without providing this routine that is...

Essentially this routine implements the following equation, which is based on the long equation following (10) in [2]:

$$x_j^{\text{new}} - x_j^{\text{old}} = \frac{d_1 - \beta \sum_k w_{kj}(x_j - x_k)}{n_2 + \delta\beta \sum_k w_{kj}},$$

where δ is the "De Pierro" factor [3] which is 1 for coordinate descent, and 2 for parallelizable algorithms.

References

[1] 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`.

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

[3] 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.

A Appendix: Prototypes

Here are the prototypes for the user-defined functions.

A.1 System model

```
/*
* sys,dynlib,proto.h
* required prototypes for dynamic library forward / backprojectors
*
* Copyright 2003-6-14 Jeff Fessler The University of Michigan
*/

typedef int bool;
typedef unsigned char byte;
typedef const int cint;

#include "def,subset.h"

extern bool sys_init(
const void *DoNotTouch1,
const char *userinfo,
const int n1,/* measurement data dimensions */
const int n2,
const int n3,
const int nx,/* object dimensions */
const int ny,
const int nz,
const int nz_raw,/* = nz unless user asks for only some slices */
const int chat); /* 0 for silent, >0 for diagnostics */

extern bool sys_free(const void *DoNotTouch);


/*
* forward projector
```

```
*/
extern bool sys_proj(
const void *DoNotTouch,
float *proj,/* [n1,n2,n3] put projections of image here */
const float *image,/* [nx,ny,nz] image volume */
const byte *mask,/* [nx,ny,nz] binary support mask */
const Subsets *ss,/* subset control structure */
const int chat); /* 0 for silent, >0 for diagnostics */


/*
* back projector
*/
extern bool sys_back(
const void *DoNotTouch,
float *back,/* [nx,ny,nz] backproject into this */
const float *data,/* [n1,n2,n3] data to backproject */
const float *wi,/* [n1,n2,n3] weights */
const Subsets *ss,/* subset control structure */
const int chat); /* 0 for silent, >0 for diagnostics */



/*
* prototypes for ASPIRE projection / backprojection routines
*/

typedef void Sys; /* not going to document the internals! */

/*
* fi_read()
* read in and initialize aspire system model
*/
extern Sys *fi_read(
const char *sys_type,/* string describing system, e.g. 2z@... */
const char *file_mask,/* filename for mask */
const byte *mask,/* [nx,ny,nz] */
/* only one of mask or file_mask should be non-null */
const int nx,/* object size */
const int ny,
const int zmin,/* 0 */
const int zmax,/* nz-1 */
const int nz_raw,/* nz */
const int nthread,/* 1, unless you have multiple processors... */
const int chat); /* verbosity */


/*
```

```
* fi_free()
* free an aspire system model
*/
extern bool fi_free(Sys *sys);


/*
* fi_proj()
* forward projector
*/
extern bool fi_proj(
float *y,/* [n1,n2,n3] */
const float *x,/* [nx,ny,nz] */
const Sys *sys,
cSubsets *ss,/* subset control; NULL defaults to all views */
const int chat);


/*
* fi_back()
* weighted backprojector:
* back = G' * data if wi = NULL
* back = G' * D(wi) * data if wi non-NULL
*/
bool fi_back(
float *back,
const float *data,/* [[nd]] */
const Sys *sys,
const float *wi,/* [[nd]] optional weights, used if non-NULL */
cSubsets *ss,/* subset control; NULL defaults to all views */
const int chat);
```

A.2 Regularization

```
/*
* rp,dynlib,proto.h
* required prototypes for dynamic library penalty
*
* Copyright Jan 1998,Jeff Fessler University of Michigan
*/

extern int rp_init(
const void *DoNotTouch1,
const void *DoNotTouch2,
const int nx,
const int ny,
const int nz,
const int chat); /* 0 for silent, >0 for diagnostics */
```

```
extern int rp_free(const void *DoNotTouch);

extern double rp_penalty(
const float *image,/* [nx,ny,nz] image volume */
const void *DoNotTouch,
const unsigned char *mask,/* [nx,ny,nz] binary support mask */
const int nx,/* image dimensions */
const int ny,
const int nz);

extern int rp_grad(
float *rx,/* [nx,ny,nz] output gradient vector */
const float *image,/* [nx,ny,nz] input current image */
const void *DoNotTouch,
const unsigned char *mask,/* [nx,ny,nz] binary support mask */
const int nx,/* image dimensions */
const int ny,
const int nz);

extern double rp_newton1(
const double d1,/* d1 */
const double n2,/* n2 */
const float *image,/* [nx,ny,nz] image */
const int nx,/* image dimensions */
const int ny,
const int nz,
const void *DoNotTouch,
const double depierro); /* depierro factor */
```