FASTA: A GENERALIZED IMPLEMENTATION OF FORWARD-BACKWARD SPLITTING

TOM GOLDSTEIN

1. What is FASTA?

FASTA (Fast Adaptive Shrinkage/Thresholding Algorithm) is an efficient, easy-to-use implementation of the Forward-Backward Splitting (FBS) method for minimizing compound objective functions. FASTA targets problems of the form

$$\text{minimize} \quad f(Ax) + g(x), \tag{1}$$

where A is a linear operator, f is a differentiable function, and g is a "simple" function for which we can evaluate the proximal operator. Consider for example the ℓ_1-penalized least squares problem

$$\text{minimize} \quad \mu|x| + \frac{1}{2}\|Ax - b\|^2 \tag{2}$$

where $|\cdot|$ denotes the ℓ_1 norm, $\|\cdot\|$ denotes the ℓ_2 norm, A is a matrix, b is a vector, and μ is a scalar parameter. This problem is of the form (1) with $g(z) = \mu|z|$, and $f(z) = \frac{1}{2}\|z - b\|^2$. More generally, any problem of the form (1) can be solved by FASTA, provided the user can provide function handles to f, g, A and A^T.

The solver FASTA contains numerous enhancements of FBS to improve convergence speed and usability. These include adaptive stepsize choice, acceleration (i.e., of the type used by the solver FISTA), backtracking line search, and numerous automated stopping conditions, and many other improvements reviews in the article *A Field Guide to Forward-Backward Splitting with a FASTA Implementation.*

2. What does FASTA come with?

Your download comes with several folders. One folder is called `solvers`. This folder contains the file `fasta.m`, which is a self-contained solver for *any* problem of the form (1).

The `solvers` folder also contains numerous specialized solvers, each of which solves a *specific* problem of the form (1). For example, the code `fasta_sparseLeastSquares` solves the sparse least squares problem (2), and `test_sparseLogistic` solves ℓ_1 penalizes logistic regression problems. Each of these specialized solvers depends on the file `fasta`; they simply cook up a specific f, g, and A corresponding to a specific problem, and hand them off to `fasta`.

The top-level folder contains test scripts that demonstrate how to use each solver. For example, the script `test_sparseLeastSquares` builds a random instance of a sparse regression problem and solves it using `fasta_sparseLeastSquares`. Each of these scripts requires no setup by the user. Simply run them from the command line.

3. How to Install FASTA

After you download the code, simply add the **solvers** folder to your path and you're ready to go.

Technically, you only need to add the single file **fasta.m** to your path if you only want to use the general solver. However, many of the test/demo scripts call specialized methods from the **solvers** folder (or have other dependencies) so it is best to add the whole **solvers** folder to your current path.

4. How to use FASTA

Calling FASTA is easy. To use the solver, you will need to implement the functions $f(x)$ and $g(x)$ and the linear operators A and A^T. You will also need a function **grad**(x) that generates the gradient of f at x and the function **prox**(x,t) representing the proximal mapping of g at x with stepsize τ. For many problems of interest, a specialized solver is already in the **solvers** folder that does all this for you. However, if you are using the general solver, you call **fasta** with the following command.

```
solution = fasta(A, At, f, gradf, g, proxg, x0);
```

Here's a complete worked example to demonstrate the use of **fasta**. Suppose we want to solve (2). The script below builds a random test problem, and then solves the penalized least squares problem using **fasta**.

```
%% Build a simple (arbitrary) test problem
A = randn(5,10); % Define this matrix however you wish!
b = randn(5,1);  % Define this vector however you wish!
mu = 1;          % Define this scalar however you wish!

%% Build the ingredients for fasta
f = @(x)  0.5*norm(x-b)^2; % The smooth function, f
gradf = @(x)  x-b;         % The gradient of f
g = norm(x,1);             % The non-smooth function, g
proxg = @(x,t) sign(x).*max(abs(x)-mu*t,0);  % The proximal operator (shrinkage)
x0 = zeros(10,1);          % The initial guess

%% Call fasta to solve: minimize f(Ax)+g(x)
solution = fasta(A, At, f, gradf, g, proxg, x0, opts );
```

Note that for this particular problem, one could just use the built-in solver **fasta_sparseLeastSquares** by calling

```
fasta_sparseLeastSquares(A,A',b,mu,x0, opts);
```

rather than using the general solver. However, the above example demonstrates how one could build a custom solver using **fasta** in the event that a specialized solver were not already available.

5. Slightly More Advanced usage

A more advanced call to **fasta** would look like this:

```
[solution, outs] = fasta(A, At, f, gradf, g, proxg, x0, opts);
```

This method call looks a lot like what we've already seen, but with two key differences. First, we added the argument **opts**, which is a struct of options that control the behavior of `fasta`. Second, we captured the second return value `outs`, which is a struct of convergence information. We describe each of these structs below.

By setting fields in the struct `opts`, the user can control the behavior of `fasta`. Several of the most useful fields are described here:

`opts.verbose` Controls how much text output appears in the console. Set `opts.verbose=1` for some output, and `opts.verbose=2` to print convergence information after every iteration.

`opts.tol` The stopping tolerance of the method. The default value `tol=1e-3` works well for most problems. However you may choose a smaller value to achieve more precision, or a larger value to achieve shorter runtime.

`opts.maxIters` The maximum number of iterations the method will perform. The default value is 1000.

The struct `outs` contains information that can be use used to fine-tune performance. The most commonly used outputs are:

`outs.solveTime` The runtime of the algorithm.

`outs.residuals` A vector containing the residuals at each iteration. The residual is a derivative (or more generally sub-gradient) of the objective function, and should be nearly zero at a good approximate minimizer.

`opts.maxIters` The maximum number of iterations the method will perform. The default value is 1000.

6. Specialized Solvers

Lasso Regression. The Lasso regression is defined as follows:

$$\text{minimize} \quad \frac{1}{2}\|Ax-b\|^2 \;\; \text{subject to} \;\; \|x\|_1 \leq \lambda.$$

This problem is solved by calling

```
solution = fasta_lasso( A, At, b, lambda, x0);
```

where `At` is the transpose of `A`, `lambda` is the regularization parameter, and `x0` is an initial guess (usually an appropriately sized vector of zeros).

ℓ_1-Penalized Least Squares. The sparse least squares (or basis pursuit denoising) problem is

$$\text{minimize} \quad \mu\|x\|_1 + \frac{1}{2}\|Ax-b\|^2.$$

This problem is solved by the command

```
solution = fasta_sparseLeastSquares(A, At, b, mu, x0);
```

ℓ_1-Penalized Logistic Regression. When the vector $b \in \{0,1\}^M$ contains binary-valued entries one is interested in solving the sparse logistic regression problem

$$\text{minimize} \quad \mu\|x\|_1 + \text{logit}(Ax, b);$$

with the logit penalty function defined as

$$\text{logit}(z,b) = \sum_{i=1}^{M} \log(e^{z_i}+1) - b_i z_i.$$

This problem is solved using the following command:

```
solution = fasta_sparseLogistic(A, At, b, mu, x0);
```

Low-Rank (1-bit) Matrix Completion. FASTA can solve the matrix completion problem

$$\text{minimize} \quad \mu\|X\|_* + \text{logit}(X, Y),$$

where $\|X\|_*$ is the low-rank inducing nuclear norm of the matrix X and logit is the logistic loss function. This is done with the command

```
solution = fasta_logisticMatrixCompletion(B, mu);
```

Phase Retrieval. The PhaseLift algorithm solves phase retrieval problems of the form

$$\text{minimize } \|X\|_* \text{ subject to } \mathcal{A}(X) = b,\ X \succeq 0.$$

In the case where the measurement vector b is contaminated by additive noise, we choose the ℓ_2-norm penalty model

$$\text{minimize } \mu\|X\|_* + \|\mathcal{A}(X) - b\|^2 \text{ subject to } X \succeq 0,$$

which can be solved using FBS. The solution to this problem is found by calling

```
solution = fasta_phaselift(A, b, mu, X0);
```

Democratic Representations. Given a signal $b \in \mathbb{R}^M$, a low-dynamic range representation can be found by choosing a suitable matrix $A \in \mathbb{R}^{M\times N}$ with $M < N$, and by solving

$$\text{minimize} \quad \mu\|x\|_\infty + \frac{1}{2}\|Ax - b\|^2.$$

This problem is solved by the command

```
solution = fasta_democratic(A, At, b, mu, x0);
```

Total Variation Denoising. Given a noisy image f, we can find a denoised image by solving

$$\text{minimize} \quad \mu|\nabla x| + \frac{1}{2}\|x - f\|^2$$

where $|\nabla x|$ denotes the total-variation of x. Denoising if performed by the command

```
solution = fasta_totalVariation(f, mu);
```

Note: this solver works on “images” of dimension 1 or higher.

Appendix A. Complete list of options

`opts.verbose` Controls how much text output appears in the console. Set `opts.verbose=1` for some output, and `opts.verbose=2` to print convergence information after every iteration.

`opts.tol` The stopping tolerance of the method. The default value `tol=1e-3` works well for most problems. However you may choose a smaller value to achieve more precision, or a larger value to achieve shorter runtime.

`opts.maxIters` The maximum number of iterations the method will perform. The default value is 1000.

opts.recordObjective If `opts.recordObjective=true,` then the method will evaluate the objective function $f(x_k) + g(x_k)$ at every iteration and store the results in `outs.objective`. Computing the objective takes time, and so turning this option on may slow down computation for some problems. The default is `opts.recordObjective=false,`

opts.recordIterates If `opts.iterates=true`, then every iterate of the method is stored and returned in the cell array `outs.iterates`. This option is turned off by default. Turning it on may dramatically increase memory requirements.

opts.adaptive Determines whether adaptive stepsizes are used. By default `opts.adaptive=true.`

opts.accelerate Determines whether to use the accelerated method FISTA. By default this is turned off, but can be turned on by setting `opts.accelerate=true`. If this option is turned on, then the user may assign a boolean value to `opts.restart` to determine whether to use a “restart” rule (default behavior uses restart).

opts.function The user may supply a function that takes a single argument. On every iteration, the value of `opts.function(`x_k`)` is computed and stored in the cell array `outs.funcVals.`

opts.backtrack Determines whether backtracking is use to guarantee stability. If this option is set to `false`, then the user should either set the stepsize manually in `opts.tau`, or else supply a Lipschitz constant for ∇f in `opts.L`. By default `opts.backtrack=true,` and there is frequently no benefit in turning this option off.

opts.stopRule A string that determines which stopping condition is used. Choose a value from {`ratioResidual, normalizedResidual, hybridResidual`}. A hybrid residual strategy is used by default.

opts.stopNow The user may implement a custom stopping rule. At each iteration k, the function `opts.stopNow(`x_k`,k,residual,normalizedResidual,maximumResidual,opts)` is evaluated. Iteration stops when the returned value is `true`. When this `opts.stopNow` is defined, this function overrides the built-in stopping rules.

opts.stringHeader This string is appended to the front of all text output when `opts.verbose=true`. This option allows the user can add custom labels to text that is printed to the console.

Appendix B. Complete list of Outputs

The struct `outs`, contains convergence information that can be use used to fine-tuning convergence. The outputs are:

`outs.solveTime`	The runtime of the algorithm.
`outs.residuals`	A vector containing the residuals at each iteration.
`outs.stepsizes`	A vector containing the stepsize used at each iteration.
`outs.normalizedResiduals`	The normalized residuals at each iteration.
`outs.objective`	The objective function evaluated at each iterate. This is not recorded by default. Set `opts.recordObjective=true` to use this option.
`outs.funcValues`	Stores the values of `opts.function(`x_k`)` for each iterate x_k. If the user did not supply a value for `opts.function,` then this will be a vector of zeros.
`outs.backtracks`	The number of times backtracking was activated.
`outs.L`	The estimated Lipschitz constant for ∇f.
`outs.initialStepsize`	The initial stepsize used for the first iteration.
`outs.iterationCount`	The total number of iterations computed before termination.
`outs.iterates`	If `opts.recordIterates=true,` then this field is a cell array containing every iterate of the method.