Adaptive and Accelerated Convex Optimization with a Sequential Paradigm for Iterative Numerical Algorithms

Noah Singer

November 2017

Abstract

FASTA is a framework for the optimization of convex functions which uses the forward/backwards splitting method augmented with backtracking stepsize selection, the Barzilai-Borwein adaptive heuristic, and FISTA-style acceleration. I develop an extensible and readable Python implementation of FASTA, intended for research, industrial, and educational purposes. It contains visualization and comparison routines for the algorithms (e.g. contour plots and convergence graphs) and has several adjustable parameters, including tolerance, backtracking window size, backtracking condition, and stopping condition. I construct a novel programming paradigm, called FLOW, based on the sequential functional combination of discrete imperative *flows*, that facilitates the development of FASTA and other iterative numerical algorithms. I use FASTA to solve 11 optimization problems, including total-variation denoising, max-norm optimization, the multiple measurement vector problem, and non-negative matrix factorization. Finally, I examine and quantify the impact of the condition number of the measurement matrix in linear least squares on the performance gap between the adaptive heuristic and the accelerated method.

1 Introduction

In mathematics and computer science, an *optimization problem* is a task whose goal is to find inputs $\mathbf{x}^*$ that *optimize*, or attain minimal values of, some function f, called the *objective*. Sometimes, these functions are optimized over all possible inputs $\mathbf{x}$: for instance, all the integers $\mathbb{Z}^n$, reals $\mathbb{R}^n$ or complex numbers $\mathbb{C}^n$. These problems are called *unconstrained* optimization problems. Other times, we seek to optimize functions only over the $\mathbf{x}$'s that satisfy a specific set of *constraints*; the set of all possible $\mathbf{x}$'s satisfying the constraints is called the *feasible set*. We may consider unconstrained optimization to be a special case of constrained optimization, where the feasible set is all possible inputs.

Most, if not all, important problems in theoretical computer science can be formulated as optimization problems. In general, optimization is computationally intractable, especially for problems depending on a large number of variables n. Many classes of optimization problems, such as optimizing linear functions over the integers (integer linear programming, ILP) optimizing Boolean functions over Boolean values (the Boolean satisfiability problem, SAT), are well-known to be NP-hard or NP-complete. However, for certain extremely important classes of well-behaved functions, optimization is much simpler. For instance, many well-known algorithms exist to efficiently optimize linear functions or quadratic forms over the real numbers. In this research, I consider the general class of *convex functions*, which have specific properties that make their optimization tractable.

Intuitively, convex functions are those which "curve up" everywhere. We can in general distinguish two types of optima: *local* and *global* optima. Global optima are optimal over the entire feasible set, whereas local optima are only optimal over some neighborhood. Although local optima may sometimes be "good enough" for practical purposes, in the theory of optimization we generally seek the global optima of functions. Convex functions have the key property that all local optima are also global optima. Therefore, in order to optimize a convex function, we need only seek a local optimum. One classic approach to doing so is to follow the function's contours "downward" to a minimum; this approach, when formalized, is called *gradient descent*. As an analogy: suppose you are in an unknown landscape, of which you can only see a small area around you, and you are seeking the highest point (or lowest point, in the case of finding a minimum). One approach might be to repeatedly find the direction of steepest ascent and then take a small step in that direction. If the landscape is well-behaved, you will always converge to a peak, but if the landscape is also convex, then you are guaranteed that that peak is the highest peak in the entire landscape. This powerful idea is the basis for many optimization algorithms. While gradient descent itself is a relatively simple algorithm, incorporating modifications that improve its customizability and efficiency makes the algorithm significantly more complex.

My goal in this research was to implement a Python version of Goldstein et al.'s Fast Adaptive Shrinkage/Thresholding Algorithm (FASTA) [5] for optimizing convex functions with constraints, using many of the latest gradient descent techniques and improvements from the literature. Specifically, FASTA incorporates two dramatic improvements over the classical gradient descent algorithm, *adaptive stepsize selection* [2, 3] and *FISTA-style acceleration* [1]. FASTA also features a variety of

available backtracking and stopping conditions, further parameters of the algorithm that can be adjusted and analyzed.

The original implementation of FASTA is in MATLAB, which is not generally considered a production-level programming language (instead, it is more of a mathematical prototyping language). My Python version of FASTA has a high-quality production codebase that is readable and extensible, one that can be distributed to students and researchers alike. I reimplemented 11 of the 12 original example optimization problems used in [5] to test the correctness of FASTA and serve as readable examples. I also developed visualizations to monitor FASTA's progress, such as convergence plots or contour maps, that are useful for understanding and debugging the algorithm.

FASTA is an algorithm with many "moving parts," individual components that in many cases can be easily deactivated or switched out for other components. In order to facilitate development, testing, and analysis of the algorithm, instead of the traditional imperative paradigm, FASTA is implemented in a novel hybrid imperative-functional paradigm for iterative numerical algorithms that I invented in the course of this research called Functional Lightweight Operator Networks, or FLOW for short. Every FLOW program consists of a collection of imperative *flows*, each of which perform a specific function by modifying a global *state* in-place. These flows are strung together in sequences called *chains* à la functional programming.

Finally, I studied certain parameters of convex optimization problems that makes them slower or faster for FASTA to solve. Specifically, I examined the impact of the *condition number*, or ratio of largest to smallest singular value, of the Hessian matrix of the objective f—which can be intuited as a measure of the "convexity" of the function for reasons that will become clear in Section 2—on how many iterations it takes various configurations of FASTA to converge to a minimum of f.

In Section 2, I review convex functions and related notions; in Section 3, I examine the various algorithms included in the FASTA package; in Section 4, I examine the FLOW paradigm and its applications; in Section 5, I numerically analyze the impact of the condition number of a problem on the performance gap between various configurations of FASTA; and in Section 6, I conclude my research and cite areas of further study.

2 Preliminaries

A *homogeneous* polynomial is a polynomial whose terms all have identical degree. A *quadratic form* is any homogeneous second-degree polynomial in some number of variables n. Any real, $n \times n$, symmetric matrix A determines a quadratic form $q(\mathbf{z}) = \mathbf{z}^\mathsf{T} A\mathbf{z}$. The level curves $\mathbf{z}^\mathsf{T} A\mathbf{z} = k$ for $k \in \mathbb{R}_+$ of this quadratic form are quadric surfaces (generalized conic sections) in n dimensions centered at the origin.

In the $n = 2$ case, it is easy to show that if $A = \begin{bmatrix} a & c \\ c & b \end{bmatrix}$ and $z = \begin{bmatrix} x & y \end{bmatrix}^\mathsf{T}$, then the quadratic forms

$$\mathbf{z}^\mathsf{T} A\mathbf{z} = ax^2 + by^2 + 2cxy = k$$

are ellipses if and only if $ab > c^2$, and hyperbolas otherwise.

In the general case, the quadric surfaces are ellipsoids if and only if the matrix A is *positive semidefinite*; that is, all its eigenvalues are positive. If A is positive semidefinite, then the semiaxes of the ellipsoid that it generates in n-space have lengths corresponding to the reciprocal square roots of the n eigenvalues.

In many applications, especially in optimization, we are interested not just in the fact that A generates ellipsoids, but in how "spherical" these ellipsoids are. The *condition number* c of a matrix A is defined as the ratio of its largest singular value to smallest singular value. When c is close to 1, all the semiaxes are roughly of the same length, so the ellipsoid is roughly spherical, while when c is very large, the ellipsoid is roughly linear, or "degenerate", in at least one dimension.

Now, we may introduce the concept of a convex function.

Definition 2.1 (Convex function)**.** *A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if and only if*

$$\forall \theta \in [0,1], \mathbf{x}, \mathbf{y} \in \mathbb{R}^n, \quad f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y}).$$

Geometrically, this is equivalent to requiring that the secant line segment between any two points $(\mathbf{x}, f(\mathbf{x}))$ and $(\mathbf{y}, f(\mathbf{y}))$ on f's graph lies entirely above the graph of f. This definition is also known as *Jensen's inequality*.

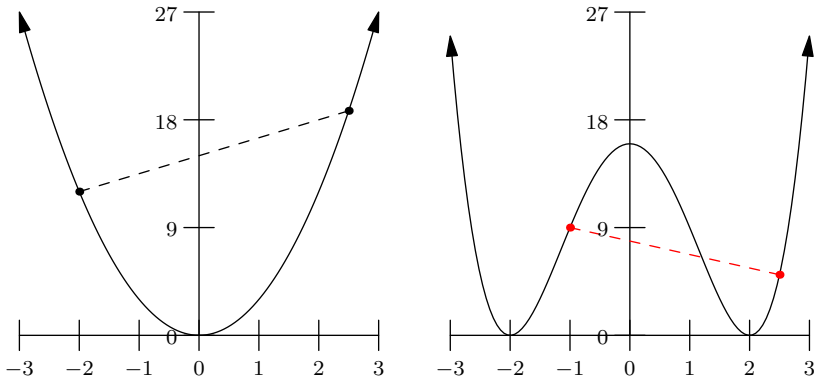

Figure 1: The functions $f(x) = 3x^2$ (left) and $g(x) = (x-2)^2(x+2)^2$ (right). All secants between pairs of points on f's graph do not lie at all below f's graph, so f is convex by Jensen's inequality. On the other hand, the secant in red between two particular points on g's graph lies partially below g's graph, so by Jensen's inequality, g is non-convex.

In the single variable case, the first derivative of $f : \mathbb{R} \to \mathbb{R}$ tells us about its rate of change, and the second derivative of f tells us about its curvature—if f'' is positive at x, then f "curves up" at x. If $f''(x) > 0$ for all x on the domain, then f is convex.

For the multivariable case, let's consider the second-degree Taylor expansion of f at some point $\mathbf{x}$ centered at $\mathbf{x_0}$:

$$f(\mathbf{x}) \approx f(\mathbf{x_0}) + \nabla f(\mathbf{x_0})(\mathbf{x} - \mathbf{x_0}) + \frac{1}{2}(\mathbf{x} - \mathbf{x_0})^\mathsf{T} \nabla^2 f(\mathbf{x_0})(\mathbf{x} - \mathbf{x_0}), \tag{2.1}$$

where the *gradient* of f, ∇f, is the vector of f's first partials, and the *Hessian* of f, $\nabla^2 f$,

is the matrix of f's second partials. Knowing that the Hessian of any twice-differentiable f is a symmetric matrix by Clairaut's theorem, we can recognize the second-degree term as the quadratic form determined by $\nabla^2 f$ and evaluated at $(\mathbf{x} - \mathbf{x_0})$.

An important theorem of convex analysis, the so-called second-order convexity condition, states that a twice-differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if and only if $\nabla^2 f$ is positive semidefinite everywhere [4], the multivariable equivalent of the second-derivative convexity condition for single variable functions. Further, since the level curves of second-degree Taylor approximations to convex functions are ellipsoids, the level curves of convex functions must be approximately ellipsoids.

The quantitative concept of the condition number of a single matrix to can be extended to qualitatively describe the conditioning of an entire convex optimization problem (which is equivalent the conditioning of the Hessian of the objective): Some convex optimization problems are *well-conditioned*, meaning that the contours of the function to be optimized are roughly circular, whereas poorly-conditioned problems have contours that are much more irregular (highly eccentric, approaching linear). We shall see in Section 3 that poorly-conditioned objectives are much more time-consuming to optimize. In fact, one key difference between the accelerated and adaptive modes of FASTA is that, while the adapted method essentially always outperforms the accelerated method in practice, the accelerated method is guaranteed to converge with a strong performance guarantee regardless of the conditioning of the problem, whereas the adaptive method is an extremely useful heuristic that fails for very poorly-conditioned problems. An additional goal of my research was to quantify the relationship between the condition number of a problem and the relative performances of the accelerated and adaptive methods.

3 FASTA: Advanced Gradient Descent

Goldstein et. al [5] developed the FASTA algorithm as a framework for convex optimization that combines several improvements on and modifications to the classical gradient descent algorithm. I review naïve gradient descent, and its several optimizations and modifications, in this section.

3.1 Gradient Descent

Gradient descent is the classical algorithm for unconstrained optimization of convex functions. The essential idea behind gradient descent is as follows: at any point $\mathbf{x}$, the gradient $\nabla f(\mathbf{x})$ points in steepest ascent direction, and therefore, in order to attempt to minimize f, we simply repeatedly move a small amount in the direction exactly opposite the gradient of f. The algorithm is *iterative*; we start from some *initializer* $\mathbf{x}^{(0)}$ and produce a sequence of *iterates* $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ which get progressively closer to $\mathbf{x}^*$. Gradient descent also depends on some *stepsize* τ which controls the distance between each successive pair of iterates, which must be sufficiently small for the algorithm to converge (the bound depends on the Lipschitz constant of f's gradient).

When the objective is well-behaved, gradient descent always converges to a local minimum. As long as the objective is also convex, this local minimum will converge to a global minimum of f.

Algorithm 1 Gradient descent

Require: $\mathbf{x}^{(0)}, \tau$
1: **for** $k \in \{0, 1, \ldots, N\}$ **do**
2: $\quad \mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \tau \nabla f(\mathbf{x}^{(k)})$
3: **end for**

Theorem 3.1 (Convergence of gradient descent [4]). *Let the objective $f : \mathbb{R}^n \to \mathbb{R}$ be convex*[1] *and differentiable, ∇f be L-Lipschitz, $\mathbf{x}^*$ be the unique global minimizer of f, and $\tau \in (0, \frac{2}{L}]$. The gradient descent algorithm (1) converges in time $\mathcal{O}(1/k)$, where k is the number of iterations; that is,*

$$||f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*)|| \in \mathcal{O}(1/k).$$

Gradient descent must be coupled with some *stopping condition* that informs the algorithm when to halt. Generally, this condition is based on the iterative *residual*, defined as

$$r = \frac{||\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}||}{\tau},$$

the basic principle being that if f is well-behaved, then r should decrease continually, and when r drops below some *tolerance*, then the algorithm halts.

3.2 Backtracking Gradient Descent

When optimizing an objective f, we want the stepsize τ to be as large as possible, since the smaller τ is, the longer the algorithm takes to converge, but at the same time, τ cannot be too large (larger than $2/L$, where L is the Lipschitz constant of ∇f). Thus, without knowing L, we may not be able to choose a good value of τ, and in general, when optimizing a convex, differentiable objective, we may have no analytic or computational knowledge of the corresponding L. One method to effectively approximate L is to generate two random vectors $\mathbf{x_1}, \mathbf{x_2}$ and then compute

$$L_{est} = \frac{||\nabla f(\mathbf{x_1}) - \nabla f(\mathbf{x_2})||}{||\mathbf{x_1} - \mathbf{x_2}||}.$$

L_{est} must be an underestimate of the true L because L is defined as the maximum over all such L_{est}. Unfortunately, if we begin by significantly underestimating L and thus significantly overestimating $\tau_{max} = 2/L$, the algorithm is no longer guaranteed to converge.

One approach to remedying this problem is to introduce an additional step into the gradient descent called *backtracking*. In backtracking gradient descent algorithms, when the stepsize is too large, we simply shrink it by some condition β until the so-called *backtracking condition* is satisfied. Thus, instead of a single stepsize τ, we have a sequence of stepsizes $\tau^{(0)}, \tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(n)}$. Each stepsize $\tau^{(k)}$ is shrunk some integer $m_k \geq 0$ times, to yield a new stepsize $\tau^{(k+1)} = \beta^{m_k} \tau^{(k)}$.

[1] We actually need a slightly stronger condition: f must be *strongly convex*, which means that there is a finite positive lower bound on the eigenvalues of the Hessian. See [4] for details.

3.3 Forward-Backward Splitting

So far, the gradient descent algorithm we have considered is only capable of optimizing differentiable functions. However, many important functions in optimization are non-differentiable or even discontinuous, such as the ℓ_1-norm or the indicator function χ_S of a given set S. Gradient descent fails for these functions because the gradient simply does not exist at certain points.

One approach to optimizing non-differentiable objectives is to introduce the *proximal operator* of a function g, centered at a point $\mathbf{v}$ with stepsize τ. Intuitively, the proximal operator yields an approximate minimizer of g that does not stray too far from the point $\mathbf{v}$. This regularization is controlled with the parameter τ.

Definition 3.1 (Proximal operator)**.** *The proximal operator* $\mathbf{prox}_g(\mathbf{v}, \tau)$ *is defined as*

$$\mathbf{prox}_g(\mathbf{v}, \tau) = \arg\min_{\mathbf{x}} \left(f(\mathbf{x}) + \frac{\tau}{2} ||\mathbf{x} - \mathbf{v}||^2 \right).$$

While in general the proximal operator may be difficult to compute, for many important non-smooth optimization problems, it is quite simple. For example, the proximal operator of the characteristic function χ_S for some set $S \subseteq \mathbb{R}^n$ is simply the Euclidean projection onto of $\mathbf{v}$ onto S, and the proximal operator of the ℓ_1-norm is the *shrink operator* (also known as the soft-thresholding operator) $\text{sgn}(\mathbf{v}) \max(|\mathbf{v}| - \tau, 0)$.

Many non-differentiable objectives can be expressed as the sum of two functions f and g, where f is convex and differentiable, and g is convex and non-differentiable but has a proximal operator that may be easily computed:

$$\arg\min_{\mathbf{x}} h(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x}).$$

Correspondingly, there exists an important class of modified gradient descent algorithms called *splitting algorithms* which may be used to optimize non-differentiable objectives. The *forward/backward splitting (FBS)* algorithm is an iterative algorithm which repeatedly executes a *forward step*, or gradient step, which is a simple classical gradient descent on f, and then a *backward step*, which evaluates the proximal operator of g.

Algorithm 2 FBS

Require: $\mathbf{x}^{(0)}, \tau$
1: **for** $k \in \{0, 1, \ldots, N\}$ **do**
2: $\quad \hat{\mathbf{x}}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \tau \nabla f(\mathbf{x}^{(k)})$
3: $\quad \mathbf{x}^{(k+1)} \leftarrow \mathbf{prox}_g(\hat{\mathbf{x}}^{(k+1)}, \tau)$
4: **end for**

3.4 Descent with Acceleration

The above algorithms always converge to the global maximizer for convex functions $f : \mathbb{R}^n \to \mathbb{R}$ with worse-case speed $\mathcal{O}(1/k)$. In practice, gradient descent algorithms often converge faster than this, but this worst-case bound remains for certain pathological functions f.

Recall that if $A \in \mathbb{R}^{n \times n}$ is positive semidefinite, then the curve formed by $\mathbf{x}^\mathsf{T} A \mathbf{x} = k$ for $k \in \mathbb{R}_+$ is an n-ellipsoid, with lengths as the square roots of the corresponding eigenvalues. If A is well-conditioned, that is, its condition number is almost 1, then the ratio between the lengths of the semimajor and semiminor axes of the ellipsoid is almost 1, and the ellipsoid is approximately a sphere. On the other hand, if A is *ill-conditioned*, then the ellipsoid is approximately linear in some dimsensions.

It is a well-known theorem of multivariable calculus that the gradient of a function f evaluated at any point is orthogonal to the tangent line to the level curve of f at that point. Therefore, if the Hessian $\nabla^2 f$ is ill-conditioned, then gradient descent will take a long time to converge: at each iterate $\mathbf{x}^{(k)}$, the component of the gradient in the direction of the minimizer $\mathbf{x}^*$ is very small, and so the gradient update makes very little progress towards the minimizer.

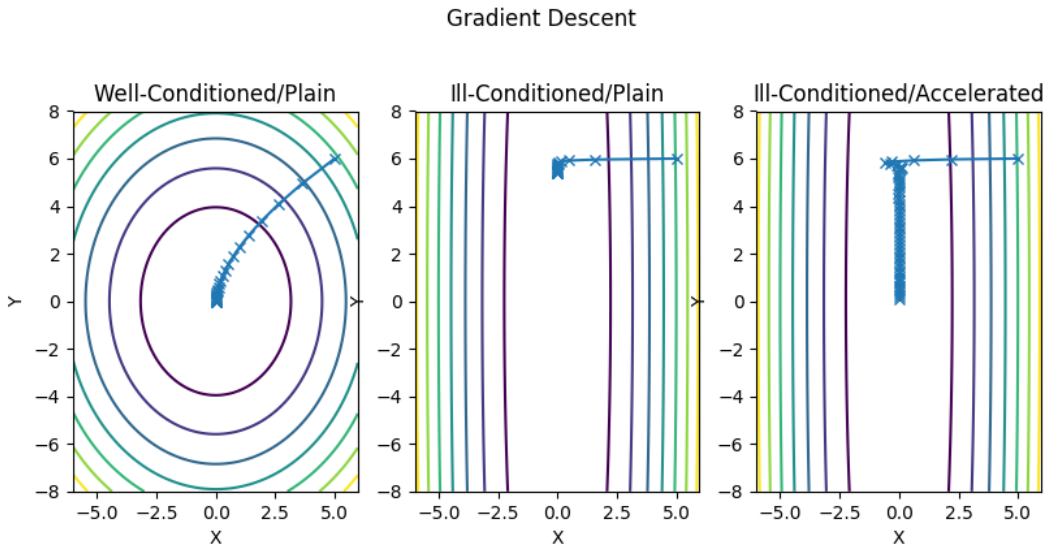

Figure 2: 50 iterations of both plain and accelerated gradient descent executed on a well-conditioned ($f(x, y) = x^2 + 0.8y^2$) and ill-conditioned ($g(x, y) = x^2 + 0.1y^2$) quadratic form (visualized using FASTA). The colored contours are level curves of the functions; the blue paths are the gradient descent iterates. Observe that the plain gradient descent converges rapidly to the minimum (in the center of the curves) on the well-conditioned problem, but performs poorly on the poorly-conditioned problem. The iterates advance only perpendicular to the "lines" formed by the level curves of g in the $-x$ direction and fail to advance downward in the $-y$ direction. Accelerated gradient descent converges significantly more rapidly than plain gradient descent on the ill-conditioned problem.

[1] introduced the Fast Iterative Shrinkage/Thresholding Algorithms (FISTA), which uses acceleration to guarantee a convergence rate of $\mathcal{O}(1/k^2)$, even for ill-conditioned problems. It relies on the principle of *momentum*: each gradient descent iteration is forced to "overshoot", so it makes significant progress towards the minimum even if the direction does not move very directly towards the minimum. The level of overshooting is controlled by the factor $\dfrac{\alpha^{(k)} - 1}{\alpha^{(k+1)}}$, where $\alpha^{(k)}$ is a parameter

that increases every iteration under a certain regime:

$$\alpha^{(k+1)} = \frac{1 + \sqrt{1 + 4(\alpha^{(k)})^2}}{2}$$

3.5 Adaptive Stepsize Selection

An important class of heuristic methods that often achieves dramatic performance benefits in practical applications over traditional gradient methods are the *adaptive methods*. In this paper, we consider one such adaptive heuristic, the *Barzilai-Borwein method* as applied in the solver SpaRSA [3] with adaptive stepsize selection [2]. At each step in the descent, the algorithm solves analytically for an ideal stepsize $\tau^{(k)}$ by approximating the objective as a perfectly conditioned quadratic optimization problem. In particular, f is approximated as

$$q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathsf{T} A\mathbf{x} - \mathbf{b}$$

where A is $\nabla^2 f(\mathbf{x})$. Immediately,

$$\nabla q(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$$

We can solve analytically for a stepsize τ which optimizes q. In the Barzilai-Borwein method, two least squares problems are formulated [6],

$$\min_\beta ||\beta(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) - (\nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)}))||^2$$

and

$$\min_\alpha ||(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) - \alpha(\nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)}))||^2$$

and by solving for α and β analytically and combining the solutions adaptively, we may can heuristically compute a τ which minimizes the resulting adjusted Taylor approximation for f. The particular adaptive stepsize selection that we use is

$$\tau = \begin{cases} \alpha & 2\alpha > \beta \\ \beta - \frac{1}{2}\alpha & \text{otherwise} \end{cases}$$

This optimization can fail when the quadratic approximation is poor: in other words, when the Hessian is poorly conditioned.

3.6 The FASTA Algorithm and Problems

FASTA is an efficient and reusable implementation of the adaptive and accelerated methods for backtracking FBS [5]. I distinguish three general *operating models* of FASTA: plain, adaptive, and accelerated.

Both the original MATLAB FASTA and this Python implementation of FASTA ship with the facilities to construct and solve example optimization problems. Each test problem serves both to verify that FASTA functions correctly and as readable examples for users seeking to use the software. The following optimization problems, distinguishable in **bold**, are all from the original FASTA paper. Each has important applications in various fields; consult [5] for details. Importantly, my implementation of FASTA uses `matplotlib`'s plotting capabilities in order to visualize the example problems.

Many of the example problems follow some variant of the **linear least squares** pattern

$$\arg\min_{\mathbf{x}} ||A\mathbf{x} - \mathbf{b}||^2,$$

with some additional constraints or penalties imposed on $\mathbf{x}$. A is the *measurement operator* which is used to measure the *unknown signal* $\mathbf{x}$, yielding the *observation vector* $\mathbf{b}$. The goal is to find the signal $\mathbf{x}$ which, when measured, produces the closest match to the given observation vector (minimal error). Many FASTA problems consist simply of the least squares problems combined with various constraints: the **LASSO** problem restricts the ℓ_1-norm of $\mathbf{x}$ to confine it into an ℓ_1-ball, the **sparse least squares/basis pursuit denoising** problem adds an ℓ_1 penalty term to the objective, the **democratic representation** problem adds an ℓ_∞ penalty term, and the **non-negative least squares** problem adds a non-negativity constraint on $\mathbf{x}$. Still other problems replace the ℓ_2-norm with the logistic log-odds function; for example, the **sparse logistic least squares** problem uses the logit funtion with ℓ_1 penalty, and the **logistic matrix completion** problem combines the logit function and the matrix nuclear norm. The **multiple measurement vector (MMV)** problem is similar to the sparse least squares problem but is for matrices; it uses the Frobenius norm of the difference between measured and observed signals and a group sparsity prior instead of the ℓ_1-norm. In each of these cases, the objective can be represented as the sum of a differentiable distance function and a non-differentiable constraint function and therefore FBS may be used.

Duality is a mathematical tool that enables conversion between a function f of $\mathbf{x}$ and a function f^* of f's Lagrange multipliers $\boldsymbol{\lambda}$. For several important problems, although their proximal operators cannot be easily evaluated, the proximal operators of their dual problems can. Therefore, we can solve the dual problems with FBS and convert back to the original form to find a minimum. Examples of this class of problems include the **total-variation denoising** problem (minimizing the total difference in intensities of adjacent pixels in order to remove noise from an image) and **support vector machine** problem (finding the hyperplane which maximally separates two classes of data in n dimensions).

Although FASTA is an algorithm for solving convex optimization problems, in practice FASTA can also give useful results for many non-convex problems. The solution is not guaranteed to be optimal, but it often suffices for practical purposes. Two such problems included with FASTA are **non-negative matrix factorization**, which factors a given matrix into the product of two matrices with non-negative entries, and **max-norm optimization**, which can be used to compute max-cuts in graphs and therefore has important applications in clustering.

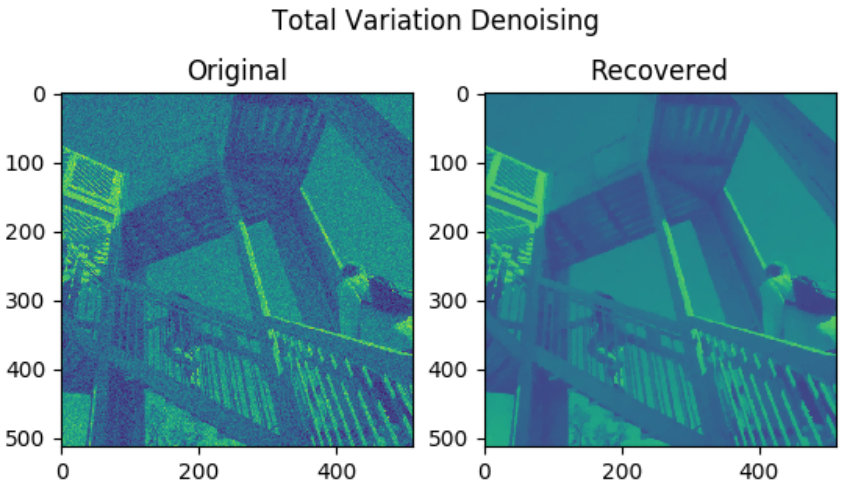

Figure 3: Total-variation denoising with FASTA. On the left is the original, noisy image; on the right is the image after the total variation between adjacent pixels has been minimized (visualized with FASTA).

4 FLOW: A Paradigm for Iterative Numerical Algorithms

One common theme in many past implementations of optimization algorithms, such as the original MATLAB implementation of FASTA, is redundant and non-meaningful code. Often, their authors implement them for analysis and comparison, not necessarily intending for their code to be read or understood, and therefore rapid prototyping is generally valued over elegance. On the other hand, my Python implementation of FASTA is designed for clarity.

I introduce the programming framework FLOW, a paradigm that abstracts the sequential nature of iterative numerical algorithms into discrete blocks called *flows*. FLOW allows the programmer to express the control flow of the program explicitly, while in usual imperative programs it is implicit in the code itself. By abstracting away the structural details of iterative computation, FLOW saves precious program space by:

- Tracking the values of each loop variable for the current and previous round.
- Optionally recording the history of each loop variable throughout the entire room.
- Splitting the algorithm into discrete pieces (*flows*) which can be activated or deactivated as necessary.

Each flow transforms a *state* in-place, which contains the assigned values of several *flow variables*. The flow itself is an imperative sequence of steps. However, algorithms implemented in the FLOW paradigm consist of groups of flows which are linked together in linear sequences called *chains*, a functional-style organization pattern. Special additional flows called *loops* allow iteration; each loop has a *body flow*, executed once per iteration, and *condition flow*, executed before the body flow and

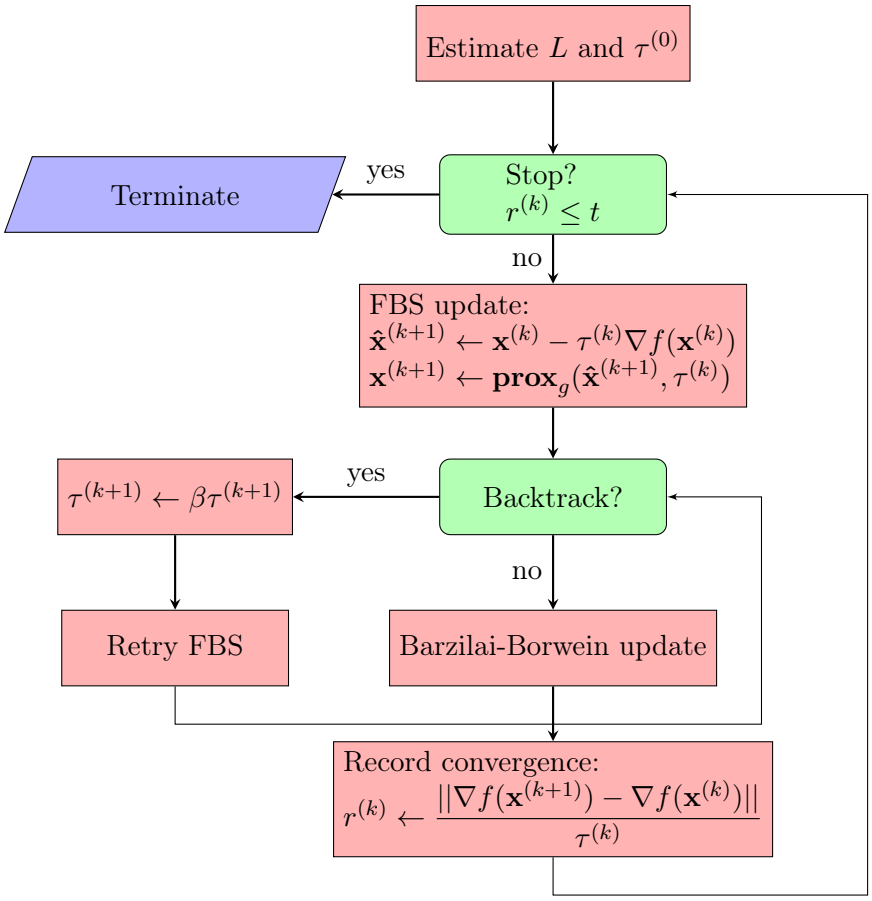

Figure 4: A flowchart that represents a FLOW-based implementation of adaptive FASTA with backtracking. FLOW allows the programmer to develop individual imperative flows and then explicitly chain, loop, and switch them together, explicitly assembling "flowchart"-like programs such as the one above.

assigning a special variable in the state called the *flow condition*. Each loop has a collection of associated *loop variables* which are assigned each iteration; for each loop, the state also tracks the histories of each loop variable in sequences called *tapes*. *Switch flows* are used to switch control flow depending on a given condition. Finally, convenience flows are included for a variety of tasks, including recording the current time or inspecting the current state for debugging.

What follows is an excerpt from the actual Python implementation of FASTA that I created using FLOW.

```
@flow.flow
def fbs(inp, state):
    """A flow to perform a forwards/backwards splitting (FBS) step."""
    # Forward step (gradient step)
    state[xhat] = state[fasta_loop, x, -1] - state[gradfx] * state[tau]
    # Backward step (proximal step)
```

```
    state[x] = inp['proxg'](state[xhat], state[tau])

    # Update loop variables
    state[Dx] = state[x] - state[fasta_loop, x, -1]
    state[z] = inp['A'](state[x])
    state[fx] = inp['f'](state[z])
    state[gradfx] = inp['A'].H(inp['gradf'](state[z]))

...

# Chain the body of the FASTA loop
body = fbs >>\
       (backtracker if backtrack else None) >>\
       (adaptive_stepsize_selector if adaptive else None) >>\
       (accelerator if accelerate else None)\
       >> convergence

# Check which variables to record every iteration
loop_vars = [x, fx, gradfx, tau, flow.TIME, residual, norm_residual, objective]
initial_vars = [x, z, fx, gradfx, tau]
if accelerate:
    loop_vars += accel_vars
    initial_vars += accel_vars

# Loop and chain FASTA
fasta_loop = flow.Loop(flow.time(body), residual_stop, loop_vars, save=True,
↪  check_first=False, initial_vars=initial_vars)
fasta_flow = stepsize_estimator >> initializer >> (acceleration_initializer if
↪  accelerate else None) >> fasta_loop
```

5 Numerical Results

As we saw earlier in Section [5], how well-conditioned a problem is can heavily affect the time it takes for FASTA to solve the problem. The relationship between these quantities depends itself on which mode FASTA is using: plain, adaptive, or accelerated.

I produced a series of 1000 measurement matrices A_i for the (noiseless) linear least squares (LLS)

problem

$$\arg\min_{\mathbf{x}} ||A_i\mathbf{x} - \mathbf{b}||^2,$$

that were progressively more ill-conditioned and compared the convergence rates of the adaptive, accelerated, and plain algorithms. Each measurement matrix A_i was selected using the following algorithm.

Algorithm 3 Measurement matrix selection

Require: σ

1: **for** $k \in \{0, 1, \ldots, N\}$ **do**
2: $B \leftarrow \mathcal{N}(0, 1)$
3: $U, V \leftarrow \textbf{svd}(B)$ (discarding the vector of singular values)
4: $S \leftarrow \textbf{diag}((e^{k\sigma\mathcal{N}(0,1)})^2)$
5: $A_k = USV^\mathsf{T}$
6: **end for**

Using these matrices, I generated 1000 least squares problems, each with $M = 10$ measurements, signals of length $N = 50$ drawn from the standard normal distribution, and a σ value of $1/15000$. I solved each problem using all three modes of FASTA. At most 100 iterations were permitted per algorithm execution, and the stopping rule was the standard normalized residual rule with a tolerance of 10^{-5}.

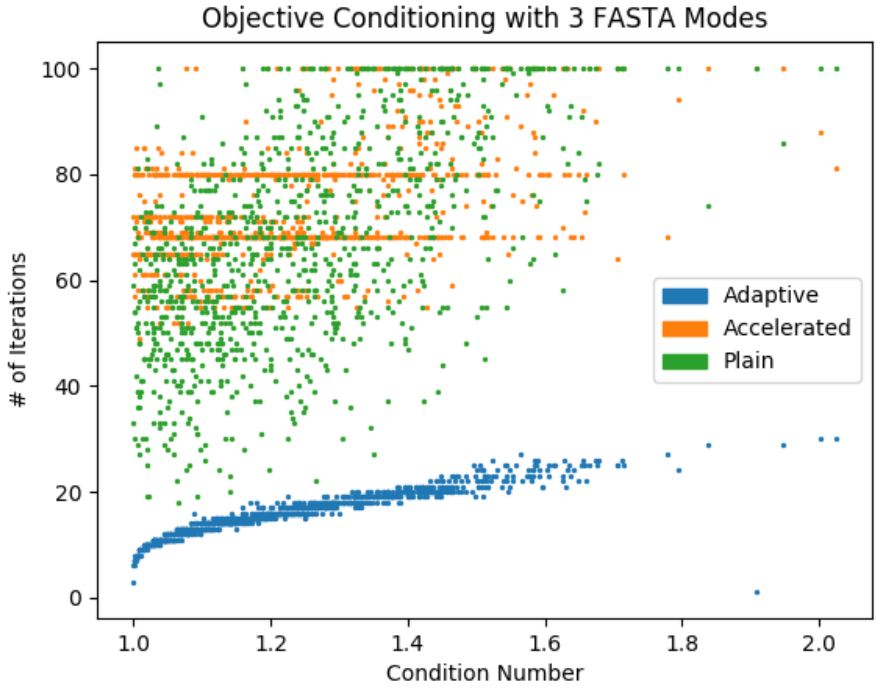

Figure 5: The relationship between the condition number of the measurement matrix and the number of iterations required for each algorithm to converge.

As expected, the adaptive heuristic converged more slowly the more ill-conditioned the measurement matrix. However, especially for very well-conditioned measurement matrices, the adaptive heuristic still converges an order of magnitude more quickly than accelerated variant. Although the graph of the convergence of the adaptive algorithm appears totally linear, when the condition number becomes very close to 1, it curves heavily downwards. Interestingly, certain specific numbers of iterations seem to be very common for the accelerated algorithm, regardless of the condition number (specifically 68 and 80); further verification of the correct convergence of the accelerated mode and investigation into these particular numbers and what factors affect them 80is certainly warranted in future research.

6 Conclusions

Optimization is one of the most important subfields of computer science due to its vast theoretical implications and practical applications. Convex optimization, which encompasses a wide group of important optimization problems, is one of the most active and productive areas of optimization research today. I have implemented a tool, FASTA, that enables comparison and analysis of different algorithms for convex optimization, as well as a framework, FLOW, which facilitates the development of iterative numerical algorithms like FASTA. Planned areas of further development for FLOW and FASTA include implementing phase retrieval algorithms (an important class of non-convex optimization algorithms) in Python with FLOW and implementing facilities for automated testing and development of FLOW algorithms with different parameters and configurations. FASTA and FLOW will also be further documented, tested, and developed in anticipation of a public release with a documentation website.

FASTA and FLOW are both completely open-source and available on GitHub; FASTA is located at `phasepack/fasta-python` and FLOW at `phasepack/flow`.

7 Acknowledgements

My deepest thanks to Prof. Thomas Goldstein at the University of Maryland, College Park, for designing the original FASTA algorithm and mentoring me throughout the research process; to Rohan Chandra for mathematical background and for technical advice; to my undergraduate collaborators, Val McCulloch at Smith, Justin Hontz at UMCP, and Ziyuan Zhong at Reed; to Prof. William Gasarch and the entire summer research program at UMCP; and to Ms. Angelique Bosse and the Senior Research Project program at Montgomery Blair High School.

References

[1] M. Teboulle A. Breck. "A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems". In: *SIAM Journal of Imaging Sciences* 2.1 (2009), pp. 183–202. URL: https://people.rennes.inria.fr/Cedric.Herzet/Cedric.Herzet/Sparse_Seminar/Entrees/2012/11/12_A_Fast_Iterative_Shrinkage-Thresholding_Algorithmfor_Linear_Inverse_Problems_(A._Beck,_M._Teboulle)_files/Breck_2009.pdf.

[2] L. Gao B. Zhou and Y.-H. Dai. "Gradient methods with adaptive step-sizes". In: *Computational Optimization and Applications* 35 (2006), pp. 69–86.

[3] R. Nowak S. Wright and M. Figueiredo. "Sparse reconstruction by separable approximation". In: *IEEE Transactions on Signal Processing* 57 (2009), pp. 2479–2493.

[4] L. Vanderberghe S.Boyd. *Convex Optimization*. 2009. URL: https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf.

[5] R. Baraniuk T. Goldstein C. Studer. "A Field Guide to Forward-Backward Splitting with a FASTA Implementation". In: (2016). arXiv: 1411.3406 [cs.NA].

[6] W. Yin. *The Barzilai-Borwein method*. 2015.