TFOCS v1.0c user guide (BETA)

Stephen Becker, Emmanuel J. Candès and Michael Grant

Applied and Computational Mathematics, Caltech, Pasadena, CA 91125

March 20, 2011

Contents

1 Introduction

TFOCS (pronounced *tee-fox*) is a library designed to facilitate the construction of first-order methods for a variety of convex optimization problems. Its development was motivated by its authors' interest in compressed sensing, sparse recovery, and low rank matrix completion, see the companion paper [1], but the software is applicable to a wider variety of models than those discussed in the paper. Before we begin, we would advise the reader to check [1] as many of the underlying mathematical concepts are introduced therein.

The core TFOCS routine `tfocs.m` supports a particular standard form: the problem

$$\text{minimize} \quad \phi(x) \triangleq f(\mathcal{A}(x) + b) + h(x) \tag{1.1}$$

where f and h are convex, $\mathcal{A}$ is a linear operator, and b is a vector. The input variable x is a real or complex vector, matrix, or element from a composite vector space. The function f must be *smooth*: its gradient $\nabla f(x)$ must be inexpensive to compute at any point in its domain. The function h, on the other hand, must be what we shall henceforth call *prox-capable*: it must be inexpensive to compute its *proximity operator*

$$\Phi_h(x,t) = \operatorname*{argmin}_z h(z) + \tfrac{1}{2}t^{-1}\langle z - x, z - x\rangle \tag{1.2}$$

for any fixed x and $t > 0$. In [1], we refer to this calculation as a *generalized projection*, because it reduces to a projection when h is an indicator function. A variety of useful convex functions are prox-capable, including norms and indicator functions for many common convex sets. Convex constraints are handled by including in h an appropriate indicator function; unconstrained smooth problems by choosing $h(x) \equiv 0$; and concave maximizations by minimizing the negative of the objective.

Let us briefly discuss the explicit inclusion of an affine form $\mathcal{A}(x)+b$ into (1.1). Numerically speaking, it is redundant: the linear operator can instead be incorporated into the smooth function. However, it turns out that with careful accounting, one can reduce the number of times that $\mathcal{A}$ or its adjoint $\mathcal{A}^*$ are called during the evolution of a typical first-order algorithm. These savings can be significant when the linear operator is the most expensive part of the objective function, as with many compressed sensing models. Therefore, we encourage users to employ a separate affine form whenever possible, though it is indeed optional.

As a simple example, consider the LASSO problem as specified by Tibshirani:

$$\begin{array}{ll} \text{minimize} & \tfrac{1}{2}\|Ax - b\|_2^2 \\ \text{subject to} & \|x\|_1 \leq \tau, \end{array} \tag{1.3}$$

where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and $\tau > 0$ are given; A can be supplied as a matrix or a function handle implementing the linear operator (see §3.2). One can rewrite this as

$$\text{minimize} \quad \tfrac{1}{2}\|Ax - b\|_2^2 + h(x),$$

where $h(x) = 0$ if $\|x\|_1 \leq \tau$ and $+\infty$ otherwise. Because the TFOCS library includes implementations of simple quadratics and ℓ_1 norm balls, this model can be translated to a single line of code:

```
x = tfocs( smooth_quad, { A, -b }, proj_l1( tau )  );
```

Of course, there other ways to solve this problem, and some further customization is necessary to obtain the best performance. The library provides a file `solver_LASSO.m` that implements this model, and includes some of these improvements.

A second TFOCS routine `tfocs_SCD.m` includes support for a different standard form, motivated by the *smoothed conic dual* (SCD) model studied in [1]:

$$\text{minimize} \quad \bar{f}(x) + \tfrac{1}{2}\mu\|x - x_0\|_2^2 + h(\mathcal{A}(x) + b). \tag{1.4}$$

In this case, neither $\bar{f}$ nor h must be smooth, but both must be prox-capable. When h is the indicator function for a convex cone $\mathcal{K}$, (1.4) is equivalent to

$$\begin{array}{ll} \text{minimize} & \bar{f}(x) + \tfrac{1}{2}\mu\|x - x_0\|_2^2 \\ \text{subject to} & \mathcal{A}(x) + b \in \mathcal{K}, \end{array} \tag{1.5}$$

which is the SCD model discussed in [1]. For convenience, then, we refer to (1.4) as the SCD model, even though it is actually a bit more general. The SCD model is equivalent to

$$\begin{array}{ll} \text{minimize} & \bar{f}(x) + \tfrac{1}{2}\mu\|x - x_0\|_2^2 + h(y) \\ \text{subject to} & \mathcal{A}(x) + b = y, \end{array}$$

which TFOCS expresses in saddle-point form

$$\text{maximize} \quad \inf_{x,y} \bar{f}(x) + \tfrac{1}{2}\mu\|x - x_0\|_2^2 + h(y) + \langle \mathcal{A}(x) + b - y, z \rangle.$$

This simplifies to something useful, namely,

$$\text{maximize} \quad \inf_{x} \bar{f}(x) + \tfrac{1}{2}\mu\|x - x_0\|_2^2 + \langle \mathcal{A}(x) + b, z \rangle - h^*(z), \tag{1.6}$$

where h^* is the convex conjugate of h; that is,

$$h^*(z) \triangleq \sup_y \langle z, y \rangle - h(y). \tag{1.7}$$

This model can be expressed as a maximization over z in our primary standard form (1.1). Therefore, given specifications for $\bar{f}$ and h^*, TFOCS can use the standard first-order machinery to solve it. For instance, consider the smoothed Basis Pursuit Denoising (BPDN) model

$$\begin{array}{ll} \text{minimize} & \|x\|_1 + \tfrac{1}{2}\mu\|x\|_2^2 \\ \text{subject to} & \|Ax - b\|_2 \leq \delta \end{array} \tag{1.8}$$

with A, b, $\mu > 0$, and $\delta > 0$ given. The function h here is the indicator function for the norm ball of size δ; its conjugate is $h^*(z) = \delta\|z\|_2$ (see the Appendix). The resulting TFOCS code is

```
x = tfocs_SCD( prox_l1, { A, -b }, prox_l2( delta ), mu  );
```

This model is considered in more detail in the file `solver_sBPDN.m`. We have provided code for other common sparse recovery models as well.

TFOCS includes a library of common functions and linear operators, so many useful models can be constructed without writing code. Users are free to implement their own functions as well. For a function f, TFOCS requires the ability to compute its value, as well as its gradient, proximity minimization (1.2), or both, depending upon how it is to be used. For a linear operator $\mathcal{A}$, TFOCS requires the ability to query the size of its input and output spaces, and to apply the forward or adjoint operation. The precise conventions for each of these constructs is provided in §3 below. If you wish to construct a prox-capable function, we also refer you to the appendix of [3] for a list of proximity operators and their calculus.

The design of TFOCS attempts to strike a balance between two competing interests. On one hand, we seek to present the algorithms themselves in a clean, readable style, so that it is easy to understand the mathematical steps that are taken and the differences between the variants. On the other, we wish to provide a flexible system with configurability, full progress tracking, data collection, and so forth—all of which introduce considerable implementation complexity. To achieve this balance, we have moved as much of the complexity to scripts, objects, and functions that are not intended for consumption by the end user. Of course, in the spirit of open source, you are free to view and modify the internals yourself; but the documentation described here focuses on the interface presented to the user.

1.1 Example library

This document does not currently provide complete examples of TFOCS-based applications. However, we are accumulating a number of examples within the software distribution itself. For instance, a variety of drivers have been created to solve specific models; these have been given the prefix `solver_` and are found in the main TFOCS directory.

In addition, we invite the reader to peruse the `examples/` directory. Feel free to use one of the examples there as a template for your project. The subdirectory `paper/` provides code that you can use to reproduce the results printed in [1]. We will be adding to and updating the examples as we can.

2 Software details

2.1 Installation

The TFOCS package is organized in a relatively flat directory structure. In order to use TFOCS, simply unpack the compressed archive wherever you would prefer. Then add the base directory to your MATLAB path; for example,

```
addpath /home/mcg/matlab/TFOCS/
```

Do *not* add the `private/` directory or any directories beginning with `@` to your path; MATLAB will find those directories automatically under appropriate circumstances. You can also add the directory via `pathtool`, which will give you the option to save the path so that you never have to do this again.

2.2 File overview

The different types of files found in the `TFOCS/` directory are distinguished by their prefix. A more complex description of each function is provided in their on-line help; a later version of this user guide will provide detailed descriptions of each in an appendix.

- `tfocs_`: The core solvers implementing optimal first-order methods for the primary standard form (1.1) (`tfocs.m` and others), and the SCD model (1.4) (`tfocs_SCD.m`).
- `solver_`: Solvers for specific standard forms such as the smoothed Dantzig selector and the LASSO. Besides providing ready-to-use solvers for specific models, these provide good templates to copy for constructing new solvers.
- `smooth_`, `prox_`, `proj_`, `tfunc_`: functions to construct and manipulate various smooth and nonsmooth functions.
- `linop_`: functions to construct and manipulate linear operators.

2.3 Calling sequences

The primary solver `tfocs.m` accepts the following input sequence:

```
[ x, out ] = tfocs( smoothF, affineF, nonsmoothF, x0, opts );
```

The inputs are as follows:

- `smoothF`: a smooth function (§3.1).
- `affineF`: an affine form specification. To represent an affine form $\mathcal{A}(x) + b$, this should be a cell array `{ linearF, b }`, where `linearF` is the implementation of $\mathcal{A}$ (§3.2). However, if $b = 0$, then supplying `linearF` alone will suffice.

- `nonsmoothF`: a nonsmooth function (§3.1).
- `x0`: the starting point for the algorithm.
- `opts`: a structure of configuration options.

The smooth function is required, but all other inputs are optional, and may be omitted or replaced with an empty array `[]` or cell array `{}`.

2.3.1 The initial point

If `x0` is not supplied, TFOCS will attempt to deduce its proper size from the other inputs (in particular the linear operator). If successful, it will initialize `x0` with the zero vector of that size. But whether or not `x0` is supplied, TFOCS must verify its feasibility as follows:

1. If $h(x_0) = +\infty$, the point must be projected into $\operatorname{dom} h$. So a single projection with step size 1 is performed, and x_0 is replaced with this value.
2. The value and gradient of $f(\mathcal{A}(x_0))$ are computed. Its value must be finite or, the algorithm cannot proceed; TFOCS has no way to query for a point in $\operatorname{dom} f(\mathcal{A}(\cdot))$.

Therefore, for best results, it is best to supply an explicit value of x_0 that is known to lie within the domain of the objective function.

2.3.2 The options structure

The `opts` structure provides several options for customizing the behavior of TFOCS. To obtain a copy of the default option structure for a particular solver, call that solver with no arguments:

```
opts = tfocs;
opts = tfocs_SCD;
```

We will discuss the various entries of the `opts` structure throughout the remainder of §2. For now, we highlight one: `opts.maxmin`. By default, `maxmin = 1` and TFOCS performs a minimization; setting it to `maxmin = -1` causes TFOCS to perform a concave maximization. In that case, the smooth function `smoothF` must be concave; the nonsmooth function `nonsmoothF` remains convex. Thus the objective function being maximized is $f(\mathcal{A}(x) + b) - h(x)$.

2.3.3 The SCD solver

The calling sequence for the SCD solver is as follows:

```
[ x, out ] = tfocs_SCD( objectiveF, affineF, conjugateF, mu, x0, z0, opts );
```

The inputs are as follows:

- `objectiveF`: a function g; or, more precisely, any function that supports the proximity minimization (1.2).
- `affineF`: an affine form specification.
- `conjugateF`: the conjugate h^* of the second nonsmooth function h.
- `mu`: the scaling for the quadratic term $\frac{1}{2}\mu\|x - x_0\|_2$. Must be positive.
- `x0`: the center-point for the quadratic term; defaults to 0.
- `z0`: the initial dual point.

- `opts`: a structure of configuration options.

In this case, `affineF`, `conjugateF`, and `mu` are required. If `objectiveF` is empty, it is assumed that $g(x) \equiv 0$.

Because TFOCS solves the dual of the SCD model, it is in fact the dual point `z0` that the underlying algorithm uses to initialize itself. Therefore, `z0` must be verified in the manner that `x0` is above. However, the all-zero value of `z0` is always acceptable: in the worst case, TFOCS will have to project away from zero to begin, but that result will always be feasible.

2.4 Customizing the solver

2.4.1 Selecting the algorithm

TFOCS implements six different first-order methods, each represented by a 2/3-letter acronym:

- `AT`: Auslender and Teboulle's single-projection method.
- `GRA`: A standard, un-accelerated proximal gradient method.
- `LLM`: Lan, Lu, and Monteiro's dual-projection method.
- `N07`: Nesterov's dual-projection 2007 method.
- `N83`: Nesterov's single-projection 1983 method.
- `TS`: Tseng's single-projection modification of Nesterov's 2007 method.

To select one of these algorithms explicitly, provide the corresponding acronym in the `opts.alg` parameter. For instance, to select the Lan, Lu, & Monteiro method, use

```
opts.alg = 'LLM';
```

when calling `tfocs.m` or `tfocs_SCD.m`. The current default algorithm is `AT`, although this is subject to change as we do further research. Therefore, once you are satisfied with the performance of your model, you may wish to explicitly specify `opts.alg = 'AT'` to protect yourself against unexpected changes.

A full discussion of these variants, and their practical differences, is given in §5.2 of [1]. Here are some of the highlights:

- For most problems, the standard proximal gradient method `GRA` will perform significantly worse than a properly tuned optimal method. We provide it primarily for comparison.
- One apparent exception to this rule is when a model is strongly convex. In that case, `GRA` will achieve linear performance, and the others will not. However, this disadvantage can be eliminated with judicious use of the `opts.restart` parameter; see §2.4.2 for information.
- The iterates generated by Nesterov's 1983 method `N83` sometimes fall outside of the domain of the objective function. If the smooth function is finite everywhere, this is not an issue. But if it is not, one of the other methods should be considered.
- In most cases, the extra projections made by `LLM` and `N07` do not significantly improve performance as measured by the number of linear operations or projections required to achieve a certain tolerance. Therefore, when the projection cost is significant (for example, for matrix completion problems), single-projection methods are preferred.

Outside of the specific cases discussed above, all of the optimal methods (that is, except `GRA`) achieve similar performance on average. However, we have observed that in some cases, one specific method will stand out over others. Therefore, for a new application, it is worthwhile to experiment with the different variants and/or solver parameters to find the best possible combination.

You may notice that the TFOCS distribution includes a number of files of the form `tfocs_AT.m`, `tfocs_GRA.m`, and so forth. These are the actual implementations of the specific algorithms. The `tfocs.m` driver calls one of these functions according to the value of the `opts.alg` option, and they have the same calling sequence as `tfocs.m` itself. Feel free to examine these files; we have endeavored to make them clean and readable.

2.4.2 Improving strong convexity performance

As mentioned above, so-called optimal first-order methods tend to suffer in performance compared to a standard gradient method when the objective function is strongly convex. This is an inevitable consequence of the way optimal first-order methods are constructed.

Using the `restart` option, it is possible to overcome this limitation. This option has a simple effect: it resets the optimal first-order method every `restart` iterations. It turns out that by doing this, the acceleration parameter θ_k remains within a range that preserves linear convergence for strongly convex problems.[1] Supplying a negative value of `restart` imposes a "no regress" condition: it resets θ_k *either* after `abs(restart)` iterations, *or* if the objective function fails to decrease, whichever comes first.

The disadvantage of restart is that the optimal choice for `opts.restart` can almost never be determined in advance. A bit of trial and error testing is required to determine the best value. However, if you are willing to invest this effort, many models can achieve significant speedups. In fact, experimenting with restart is beneficial for many models that are *not* strongly convex.

Examples of the effect of restart on algorithm performance are given in §5.6 and §6.1 of [1]. You can examine and reproduce those experiments using the code found in the subdirectories

```
TFOCS/examples/strong_convexity
TFOCS/examples/compare_solvers
```

of the TFOCS distribution. Some of the model-specific scripts, such as `solver_LASSO.m`, already include a default value of the `restart` parameter; but even when using those codes, further experimentation may be worthwhile.

In a future version of TFOCS, we hope to provide a more automatic way to adaptively detect and exploit local strong convexity.

2.4.3 Line search control

TFOCS implements a slight variation of the backtracking line search methods presented in [1]. The following parameters in the `opts` structure can be used to control it:

`L0`: The initial Lipschitz estimate. The default is `1`, or `Lexact` (see below) if it is provided. `L=1` is typically a severe underestimate, but the backtracking line search generally corrects for this after the first backtracking step.

`beta`: The step size reduction that should occur if the Lipschitz bound is violated. If `beta>=1`, TFOCS employs a fixed step size `t=1/L`. The default is `beta=0.5`; that is, the step size is halved when a violation occurs.

`alpha`: The step size will be *increased* by `1/alpha` at each iteration. This allows the step size to adapt to changes in local curvature. The default value is `alpha=0.9`.

`Lexact`: The exact Lipschitz estimate. If supplied, it will do two things: first, it will prevent the step size from growing beyond `t=1/Lexact`. Second, if the backtracking search *tries* to grow it beyond this level, it will issue a warning. This is useful if you believe you know what the global Lipschitz constant is, and would like to verify either your calculations or your code.

2.4.4 Stopping criteria

There are a variety of ways to decide when the algorithm should terminate:

`tol`: TFOCS terminates when the iterates satisfy $\|x_{k+1} - x_k\| / \max\{1, \|x_{k+1}\|\} \leq$ `tol`. The default value is 10^{-8}; if set to zero or a negative value, this criterion will never be engaged.

`maxIts`: The maximum number of iterations the algorithm should take; defaults to `Inf`.

[1] See Section 5 in [1] for a proper introduction to the role played by the parameter sequence $\{\theta_k\}$.

`maxCounts`: This option causes termination after a certain number of function calls or linear operations are made; see §2.4.6 for details. It defaults to `Inf`.

`stopCrit`: Choose from one of several stopping criteria. By default, `stopCrit` is 1, which is our recommended stopping criteria. Setting this to 3 will use a stopping criteria applied to the dual value (so this is only available in SCD models, where the dual is really the primal), and setting this to 4 is similar but uses a relative error tolerance. For details, see the code in `private/tfocs_iterate.m`.

`stopFcn`: This option allows you to supply one or more stopping criteria of your own design. To use it, set `stopFcn` must be a function handle or a cell array of function handles. For `tfocs.m`, these function handles will be called as follows:

```
stop = stopFcn( f, x );
```

where `f` is the function value and `x` is the current point.

```
stop = stopFcn( f, z, x );
```

where `f` is the current *dual* function value, `z` is the current dual point, and `x` is the current primal point. The output should either be `true` or `false`; if `true`, the algorithm will stop.

Note that the standard stopping criteria still apply, so the algorithm will halt when any of the stopping criteria are reached. To ignore the standard stopping criteria, set `stopCrit` to ∞.

2.4.5 Data collection and printing

The `printEvery` option tells TFOCS to provide a printed update of its progress once every `printEvery` iterations. Its default value is 100. To suppress all output, set `printEvery` to zero. By default, the printing occurs on the standard output; to redirect it to another file, set the `fid` option to the FID of the file (the FID is the output of MATLAB's `fopen` command).

The second output `out` of `tfocs.m` and `tfocs_SCD.m` (as well as the algorithm-specific functions `tfocs_AT.m`, etc.) is a structure containing additional information about the execution of the algorithm. The fields contained in this structure include:

`alg`: the 2-3 letter acronym of the algorithm used.

`algorithm`: the long name of the algorithm.

`status`: a string describing the reason the algorithm terminated.

`dual`: the value of the dual variable, for saddle-point problems.

Furthermore, if `opts.saveHist = true`, several additional fields will be included containing a per-iteration history of the following values:

`f`: the objective value.

`theta`: the acceleration parameter θ.

`stepsize`: the step size; i.e., the reciprocal of the local Lipschitz estimate.

`norm_x`: the Euclidean norm of the current iterate $\|x_k\|$.

`norm_dx`: the Euclidean norm of the difference $\|x_k - x_{k-1}\|$.

`counts`: operation counts; see §2.4.6.

`err`: custom measures; see below for a description.

Note that for saddle point problems (like those constructed for `tfocs_SCD`), TFOCS is actually solving the dual, so `norm_x` and `norm_dx` are computed using the dual variable.

Using the `errFcn` option, you can construct your own error measurements for printing and/or logging. The convention is very similar to `stopFcn`, in that `errFcn` should be a function handle or an array of function handles, and the calling convention is identical; that is,

```
val = errFcn( f, x );
val = errFcn( f, z, x );
```

for `tfocs.m` and `tfocs_SCD.m`, respectively. However, unlike the `stopFcn` functions, error functions can return any scalar numeric value they wish. The results will be stored in the matrix `out.err`, with each error function given its own column.

2.4.6 Operation counts

Upon request, TFOCS can count the number of times that the algorithm requests each of the following five computations:

- smooth function value,
- smooth function gradient,
- forward or adjoint linear operation,
- nonsmooth function value, and
- nonsmooth proximity minimization.

To do this, TFOCS wraps the functions with code that increments counter variables; the results are stored in `out.counts`. Unfortunately, we have found that this wrapper causes a noticeable slowdown of the algorithm, particularly for smaller models, so it is turned off by default. To activate it, set the `countOpts` option to `true`.

Operation counts may also be used to construct a stopping criterion, using the `maxCounts` option to set an upper bound on the number of each operation the algorithm is permitted to make. For instance, to terminate the algorithm after 5000 applications of the linear operator, set

```
opts.maxCounts = [ Inf, Inf, 5000, Inf, Inf ].
```

If you set `opts.maxCounts` but not `opts.countOps`, TFOCS will only count those operations involved in the stopping criteria. Of course, the number of operations is strongly correlated with the number of iterations, so the best choice is likely to use `opts.maxIts` instead.

3 Constructing models

The key tasks in the construction of a TFOCS model is the specification of the smooth function, the linear operator, and the nonsmooth function. The simplest way to do so is to use the suite of *generators* provided by TFOCS. A generator is a MATLAB function that accepts a variety of parameters as input, and returns as output a function handle suitable for use in TFOCS. The generators that TFOCS provides for smooth functions, linear operators, and nonsmooth functions are listed in the subsections below.

If the generator library does not suit your application, then you will have to build your own functions. To do so, you will need to be reasonably comfortable with MATLAB programming, including the concepts of function handles and anonymous functions. The following MATLAB help pages are good references:

```
doc function_handle
MATLAB > User Guide > Mathematics > Function Functions
MATLAB > User Guide > Programming Fundamentals > Types of Functions
       > Anonymous Functions
Optimization Toolbox > User Guide > Setting Up an Optimization Problem
       > Passing Extra Parameters
```

The use of function handles and structures is similar to functions like `fminunc` from MATLAB's `Optimization Toolbox`.

Remember, TFOCS expects minimization objectives to be convex and maximization objectives to be concave. TFOCS makes no attempt to check if your function complies with these conditions, or if the quantities are computed correctly. The behavior of TFOCS when given incorrect function definitions is undefined; it *may* terminate gracefully, but it may also exhibit strange behavior.

If you do implement your own functions—even better, if you implement your own function *generators*—then we hope you will consider submitting them to us so that we may include them in a future version of TFOCS.

3.1 Functions: smooth and nonsmooth

When TFOCS is given a smooth function f, it must be able to compute its gradient $\nabla f(x)$ at any point $x \in \operatorname{dom} f$. (Note that this implies that $\operatorname{dom} f$ is open.) On the other hand, when given a nonsmooth function h, it must be able to compute the proximity operation

$$x = \Phi_h(z,t) = \operatorname*{argmin}_x h(x) + \tfrac{1}{2}t^{-1}\langle x - z, x - z\rangle. \tag{3.1}$$

Put another way, we are to find the unique value of z that satisfies

$$0 \in \partial h(z) + t^{-1}(z - x), \tag{3.2}$$

where $\partial h(z)$ represents the subgradient of h at z. But in fact, for some differentiable functions, this proximity operation can be computed efficiently: for instance,

$$f(x) = \tfrac{1}{2}x^T x \quad \Longrightarrow \quad \nabla f(x) = x,\ \Phi_f(x,t) = (1-t)x. \tag{3.3}$$

While there is no reason to use a nonsmooth function in this manner with `tfocs.m`, it does allow certain smooth objectives to be specified for `tfocs_SCD`, or perhaps for other standard forms we might consider in the future.

For that reason, TFOCS defines a single, unified convention for implementing smooth and nonsmooth functions. The precise computation that TFOCS is requesting at any given time is determined by the number of inputs and arguments employed:

Computing the value. With a single input and single output,

```
v = func( x )
```

the code must return the value of the function at the current point.

Computing the gradient. With a single input and two outputs,

```
[ v, grad ] = func( x )
```

the code must return the value and gradient of the function at the current point.

Performing proximity minimization. With two input arguments,

```
[ vz, z ] = func( x, t )
```

the code is to determine the minimizer `z` of the proximity minimization (3.1) above, and return the value of the function $f(z)$ evaluated at that point.

3.1.1 Generators

Smooth functions:

`smooth_constant( d )`: $f(x) \equiv d$. d must be real.

`smooth_linear( c, d )`: $f(x) = \langle c, x \rangle + d$. If `d` is omitted, then `d=0` is assumed. `c` may be real or complex, but `d` must be real.

`smooth_quad( P, q, r )`: $f(x) = \frac{1}{2}\langle x, Px \rangle + \langle q, x \rangle + r$. `P` must either be a matrix or a square linear operator. It must be positive or negative semidefinite, as appropriate, but this is not checked. All arguments are optional; the defaults are `P=I`, `q=0`, and `r=0`, thus calling `smooth_quad` with no arguments yields $f(x) = \frac{1}{2}\langle x, x \rangle$. `r` must be real, but `P` and `q` may be complex.

`smooth_logsumexp`: $f(x) = \log \sum_{i=1}^{n} e^{x_i}$. This generator takes no arguments.

`smooth_entropy`: $f(x) = -\log \sum_{i=1}^{n} x_i \log x_i$. This generator also takes no arguments. This function is concave. *Important note:* the entropy function fails the Lipschitz continuity test used to guarantee the global convergence and performance of the first-order methods.

`smooth_logdet(q,C)`: $f(X) = \langle C, X \rangle - q \log \det(X)$, for C symmetric/Hermitian and $q > 0$. By default, $q = 1$ and $C = 0$. The function is convex, and the domain is the set of positive definite matrices. *Important note:* like the entropy function, the gradient of logdet is not Lipschitz continuous.

`smooth_logLLogistic(y)`: $f(\mu) = \sum_i y_i \mu_i - \log(1+e^{\mu_i})$ is the log-likelihood function for a logistic regression model with two classes ($y_i \in \{0, 1\}$) where $\mathbb{P}(Y_i = y_i | \mu_i) = e^{\mu_i y_i}/(1+e^{\mu_i})$, and μ is the (unknown) parameter to be estimated given that the data y have been observed.

`smooth_logLPoisson(y)`: $f(\lambda) = \sum_i -\lambda_i - y_i \log(\lambda_i)$ is the log-likelihood function when the y_i are observations of the independent Poisson random variables Y_i with parameters λ_i.

`smooth_huber(tau)`: is defined component-wise $f(x) = \sum_i h(x_i)$ where $h(x) = \begin{cases} x^2/(2\tau) & |x| \le \tau \\ |x| - \tau/2 & |x| > \tau \end{cases}$. This function is convex. By default, $\tau = 1$; τ must be real and positive. Though it may be possible to also use the Huber function in a nonsmooth context, it is currently not yet implemented.

`smooth_handles(f,g)`: this allows the user to easily build their own function in the TFOCS format. f is a function handle to the user's smooth function, and g is a function handle to the gradient of this function. Often the function and gradient can share some computation to save computational cost, so if this is the case, you should write your own function and not use `smooth_handles`.

The functions `smooth_constant`, `smooth_linear`, some versions of `smooth_quad` (specifically, when P an explicit matrix so that we can form its resolvent; this is efficient when P is a scalar or diagonal matrix), and `smooth_logdet` can be used in both smooth and nonsmooth contexts since they support proximity operations.

Indicator functions: See also the Appendix.

`proj_Rplus`: the nonnegative orthant $\mathbb{R}^n_+ \triangleq \{x \in \mathbb{R}^n \mid \min_i x_i \ge 0\}$.

`proj_simplex( s )`: the s-simplex $S_t \triangleq \{x \in \mathbb{R}^n \mid \min_i x_i \ge 0, \sum_i x_i = s\}$.

`proj_l1( s )`: the ℓ_1 ball $\{x \mid \|x\|_1 \le s\}$.

`proj_l2( s )`: the ℓ_2 ball $\{x \mid \|x\|_2 \le s\}$.

`proj_linfty( s )`: the ℓ_∞ ball $\{x \mid \|x\|_\infty \le s\}$.

`proj_psd`: the space of positive definite matrices: $\{X \in \mathbb{R}^{n \times n} \mid \lambda_{\min}(X + X^H) \ge 0\}$.

`proj_psdUTrace( s )`: the space of positive definite matrices with trace s: $\{X \in \mathbb{R}^{n \times n} \mid \lambda_{\min}(X + X^H) \ge 0,\ \mathrm{Tr}(X) = s\}$.

`proj_box( l, u )`: the box $\{x \in \mathbb{R}^n \mid \ell \preceq x \preceq u\}$.

`proj_Rn`: the entire space $\mathbb{R}^n$ (i.e., the unconstrained case).

For all of the cases that accept a single parameter `s`, it is optional; if omitted, `s=1` is assumed. So, for instance, `proj_l2` returns the indicator of the ℓ_2 ball of unit radius.

Other nonsmooth functions:

`prox_l1( s )`:.`prox_l2( s )`, `prox_linf( s )`. $h(x) = s\|x\|_1$, $s\|x\|_2$, and $s\|x\|_\infty$, respectively.

`prox_l1l2( s )` is the sum (i.e. ℓ_1 norm) of the ℓ_2 norm of the rows of a matrix. s may be a scalar or a vector, in which case it scales the rows of the matrix.

`prox_l1linf( s )` is the sum (i.e. ℓ_1 norm) of the ℓ_∞ norm of the rows of a matrix. s may be a scalar or a vector, in which case it scales the rows of the matrix.

`prox_nuclear( s )`. The nuclear norm scaled by $s > 0$: $h(X) = s \cdot \sum_{i=1}^{n} \sigma_i(X)$ where $\sigma_i(X)$ are the singular values of X.

`prox_spectral( q )`. The spectral norm scaled by $q > 0$: $h(X) = q\|X\| = q \max_{i=1}^{n} \sigma_i(X)$.

`prox_trace(q, largescale_flag)`. The trace of a matrix, scaled by $q > 0$: $h(X) = q\mathrm{tr}(X)$. For proximity function, this imposes the constraint that $X \succeq 0$. If *largescale_flag* is provided, and is true, then uses a Lanczos-based solver to compute the eigenvalue decomposition. This is beneficial when the size of X exceeds roughly 200×200.

`prox_hinge(q,r)`. The hinge loss function, $hl(x) = q \sum_i [r - x_i]_+$, where $[x]_+ = \max(0, x)$, and $q > 0$. By default, $q = r = 1$.

`prox_hingeDual(q,r)`. The dual to the (q, r) hinge loss function. Explicitly, $h(y) = \begin{cases} ry & y \in [-q, 0] \\ +\infty & \text{else} \end{cases}$.

`prox_0`. A synonym for `proj_Rn`; $h(x) \equiv 0$.

As with the indicator functions, `s` is optional; `s=1` is assumed if it is omitted.

Function combining and scaling:

`tfunc_sum( f1, f2, ..., fn )`. $f(x) = \sum_i f_i(x)$. The inputs are handles to other functions. They must all have the same curvature; do not mix convex and concave functions together. Sums are only useful for smooth functions; it is generally not possible to efficiently solve the proximity minimization for sums.

`tfunc_scale( f1, s, A, b )`. $f(x) = s \cdot f(A \cdot x + b)$. `s` must be a real scalar, and `f1` must be a handle to a smooth function. `A` must be a scalar, a matrix, or a linear operator; and `b` must be a vector. `A` and `b` are optional; if not supplied, they default to `A=1`, `b=0`.

This function can be used to scale both smooth and nonsmooth functions as long as `A` is a nonzero scalar (or if it is omitted). If `A` is a matrix or linear operator, it can only be applied to smooth functions. Furthermore, in this latter case it is more efficient to move `A` into the linear operator specification.

3.1.2 Building your own

In order to properly determine which computation TFOCS is requesting, it is necessary to test both `nargin` (the number of input arguments) and `nargout` (the number of output arguments). The examples in this section provide useful templates for performing these tests. That said, TFOCS will not attempt to compute the gradient of any function it expects to be smooth; likewise, it will not attempt a proximity minimization for any function it expects to be nonsmooth. Furthermore, when supplied, the step size `t` is guaranteed to be positive.

With `x` and `t` being the only input arguments, it would seem impossible to specify functions to TFOCS that depend on one or more known (but fixed) parameters. That problem is resolved using MATLAB's *anonymous function* facility. For example, consider how we would implement a quadratic function $f(x) \triangleq \frac{1}{2}x^T P x + q^T x + r$. (Of course, TFOCS already includes a `smooth_quad` generator.) We can easily create a function that accepts P, q, r, and x, and returns the value and gradient of the function:

```
function [ f, g ] = quad_func( P, q, r, x, t )
if nargout == 5,
    error( 'This function does not support proximity minimization.' );
else
    g = P * x + q;
    f = 0.5 * ( x' * ( g + q ) ) + r;
end
```

TFOCS cannot use this function in this form. But using an anonymous function, we can "hide" the first three arguments as follows:

```
my_quad = @(varargin)quad_func( P, q, r, varargin{:} );
```

Now, calls to `my_quad( x )` will automatically call `quad_func` with the given values of `P`, `q`, and `r`. The way we have designed it `my_quad( x, t )` will result in an error message.

There is one important caveat here: once `my_quad` has been created, the values of `P`, `q`, and `r` that it uses are fixed. This is due to the way MATLAB constructs anonymous functions. So don't change `P` *after the fact* expecting your function to change with it! Instead, to you must actually *re-create* the anonymous function again.

For an example of an indicator function, let us show how to implement the function generated by `proj_box`. A four-argument version of the function is

```
function [ hx, x ] = proj_box_lu( l, u, x, t )
hx = 0;
if nargin == 4,
    x = max( min( x, u ), l );
elseif nargout == 2,
    error( 'This function is not differentiable.' );
elseif any( x < l ) || any( x > u ),
    hx = Inf;
end
```

To convert this to a form usable by TFOCS, we utilize an anonymous function to hide the first two arguments:

```
my_box = @(varargin)proj_box_lu( l, u, varargin{:} );
```

Note the use of the value `+Inf` to indicate that the input `x` falls outside of the box.

Finally, for an example of a nonsmooth function that is not an indicator, here is an implementation of the ℓ_1 norm $h(z) = \|z\|_1$:

```
function [ hx, x ] = l1_norm( x, t )
if nargin == 2,
    x  = sign(x) .* max( abs(x) - t, 0 );
elseif nargout == 2,
    error( 'This function is not differentiable.' );
end
hx = sum( abs( x ) );
```

This is the well known shrinkage operator from sparse recovery. TFOCS includes a more advanced version of this function in its library with support for scaling and complex vectors.

To assist with building nonsmooth functions, see `private/tfocs_prox.m` which is analogous to `linop_handles.m` and `smooth_handles.m`. For smooth and nonsmooth functions, we have some test functions `test_smooth.m` and `test_nonsmooth.m` which can help find bugs (but unfortunately cannot guarantee bug-free code).

3.2 Linear operators

The calling sequence for the implementation `linearF` of a linear operator $\mathcal{A}$ is as follows:

```
y = linearF( x, mode )
```

The first input `x` is the input to the operation. The second input `mode` describes what the operator should do, and can take one of three values:

- `mode=0`: the function should return the size of the linear operator; more on this below. The first argument `x` is ignored.
- `mode=1`: the function should apply the forward operation $y = \mathcal{A}(x)$.
- `mode=2`: the function should apply the adjoint operation $y = \mathcal{A}^*(x)$.

In addition to the generators listed below, TFOCS provides two additional functions, `linop_normest` and `linop_test`, that provide useful information about linear operators. The function `linop_normest` estimates the induced operator norm

$$\|\mathcal{A}\| \triangleq \max_{\|x\|=1} \|\mathcal{A}(x)\| = \max_{\langle x,x\rangle=1} \langle \mathcal{A}(x), \mathcal{A}(x)\rangle^{1/2} \tag{3.4}$$

which is useful when rescaling matrices for more efficient computation (see §4.5). The function `linop_test` performs some useful tests to verify the correctness of a linear operator; see §3.2.2 below for more information.

3.2.1 Generators

`linop_matrix( A, cmode )`. $\mathcal{A}(x) = A \cdot x$. If `A` is complex, then the second input `cmode` is required; it is described below.

`linop_dot( c, adj )`. $\mathcal{A}(x) = \langle c, x\rangle$ if `adj=false` or `adj` is omitted; $\mathcal{A}(x) = c \cdot x$ if `adj` is true. In other words, `linop_dot( c, true )` is the adjoint of `linop_dot( c )`.

`linop_scale( s )`. $\mathcal{A}(x) = s \cdot x$. `s` must be a scalar.

`linop_handles( sz, Af, At, cmode )`. Constructs a linear operator from two function handles `Af` and `At` that implement the forward and adjoint operations, respectively. The `sz` parameter describes the size of the linear operator, according to the rules described in §3.2.2 below. The `cmode` string is described below.

`linop_compose( A1, A2, ..., An )`. Constructs the operator formed from the composition of n supplied operators or matrices: $\mathcal{A}(x) = \mathcal{A}_1(\mathcal{A}_2(...\mathcal{A}_N(x)...))$. Any matrices must be real; complex matrices must first be converted to operators first using `linop_matrix`.

`linop_spot( opSpot, cmode )`. Constructs a TFOCS-compatible linear operator from a linear operator object from the SPOT library [2]. If the operator is complex, then the `cmode` string must also be supplied. In a later version of TFOCS, you will be able to pass SPOT operators directly into TFOCS.

`linop_adjoint( A1 )`. $\mathcal{A}(x) = \mathcal{A}_1^*(x)$. That is, `linop_adjoint` returns a linear operator that is the adjoint of the one supplied.

`linop_TV( sz )`. Implements a real-to-complex total variation operator for a matrix of size `sz`. Given an instance `tv_op` of this operator, the total variation of a matrix `X` is `norm(tv_op(X,1),1)`.

`linop_subsample`. Used for subsampling the entries of a vector, the rows of a matrix (e.g. for a partial Fourier Transform), or the entries of a matrix (e.g. for matrix completion).

For `linop_matrix`, `linop_handles`, and `linop_spot`, a string parameter `cmode` is used to specify how the operator is to interact with complex inputs. The string can take one of four values:

- `'C2C'`: The input and output spaces are both complex.

- `'R2C'`: The input space is real, the output space is complex.
- `'C2R'`: The input space is complex, the output space is real.
- `'R2R'`: The input and output spaces are both real. This is provided primarily for completeness, and effectively causes `imag(A)` to be ignored.

So for instance, given the operator

```
linearF = linop_matrix( A, 'R2C' ),
```

The forward operation `linearF(x,1)` will compute `A*x`, and the adjoint operation `linearF(x,2)` will compute `real(A'*x)`. If one of these operators is fed a complex input when it is not expected—for instance, if `linearF` is fed a complex input with `mode=2`—then an error will result.

3.2.2 Building your own

When building your own linear operator, one of the trickier aspects is correctly reporting the size of the linear operator when `mode=0`. There are actually two ways to do this. For linear operators that operate on column vectors, we can use a standard MATLAB convention `[m,n]`, where `m` is the number of *output* elements and `n` is the number of `n` *input* elements (in the forward operation). Note that this is exactly the result that would be returned by `size(A)` if `A` were a matrix representation of the same operator.

However, TFOCS also supports operators that can operate on matrices and arrays; and a future version will support custom vector space objects as well. Therefore, the standard MATLAB convention is insufficient. To handle the more general case, a linear operator object can return a 2-element *cell array* `{ i_size, o_size }`, where `i_size` is the size of the input, and `o_size` is the size of the output (in the forward operation). Note that the input size comes first.

For example, consider the linear operator described by the Fourier transform:

```
function y = fft_linop( N, x, mode )
switch mode,
case 0, y = [N,N];
case 1, y = (1/sqrt(N)) * fft( x );
case 2, y = sqrt(N) * ifft( x );
end
```

To use the alternate size convention, replace the `case 0` line above with this:

```
case 0, y = { [N,1], [N,1] };
```

For use with TFOCS, we construct an anonymous function to hide the first input:

```
fft_1024 = @(x,mode)fft_linop( N, x, mode );
```

It is a common error when constructing linear operator objects to compute the adjoint operation incorrectly. For instance, note the scaling factors used in `fft_linop` above, which yield a unitary linear operator; other scaling factors are possible, but to omit them altogether would destroy the adjoint relationship. The key mathematical identity that defines the adjoint of $\mathcal{A}^*$ is its satisfaction of the inner product test,

$$\langle y, \mathcal{A}(x) \rangle = \langle \mathcal{A}^*(y), x \rangle \quad \forall x, y. \tag{3.5}$$

We encourage you to fully test your linear operators by verifying compliance with this condition before attempting to use it in TFOCS. The function `linop_test` will do this for you: it accepts a linear operator as input and performs a number of inner product tests using randomly generated data. Upon completion, it prints out measures of deviation from compliance with this test, as well as estimates of the operator norm.

4 Advanced usage

4.1 Matrix variables

It is not necessary to limit oneself to simple vectors in TFOCS; the system will happily accept variables that are matrices or even multidimensional arrays. Image processing models, for instance, may keep the image data in its natural two-dimensional matrix form.

The functions `tfocs_dot.m` and `tfocs_normsq.m` provide an implementation of the inner product $\langle x, y \rangle$ and the implied squared norm $\|x\|^2 = \langle x, x \rangle$ that work properly with matrices and arrays. Using these operators instead of your own will help to minimize errors.

Linear operators must be implemented with care; in particular, you must define the size behavior properly; that is, the behavior when the linear operator is called with the `mode=0` argument. For instance, to define an operator `linearF` that accepts arrays of size $m \times n$ as input and returns vectors of size p as output, a call to `linearF([],0)` must return the cell array `{[m,n],[p,1]}`. The reader is encouraged to study §3.2.2 closely, and to consider the matrix-based example models provided in the library itself.

Smooth and nonsmooth functions may be implemented to accept matrix or array-valued inputs as well. Standard definitions of convexity or concavity must hold. For instance, if f is concave, then it must be the case that

$$f(Y) \leq f(X) + \langle \nabla f(X), Y - X \rangle \quad \forall X \in \operatorname{dom} f,\, Y \tag{4.1}$$

Note that $\nabla f(X)$ is a member of the same vector space as X itself. Particular care must be exercised to implement the proximity minimization properly; for matrix variables, for instance, the corresponding minimization involves the Frobenius norm:

$$\Phi_h(X, t) = \operatorname*{argmin}_Z h(Z) + \tfrac{1}{2} t^{-1} \|Z - X\|_F^2 \tag{4.2}$$

4.2 Complex variables and operators

As we have already stated, TFOCS supports complex variables, linear operators on complex spaces, and functions accepting a complex input. Nevertheless, we feel it worthwhile to collect the various caveats that one must follow when dealing with complex variables under a single heading.

First of all, note that TFOCS works exclusively with Hilbert spaces. Thus they must have a *real* inner product; e.g., for $\mathbb{C}^n$, $\langle x, y \rangle = \Re x^H y$. In other contexts, complex vector spaces are given complex inner products satisfying $\langle x, y \rangle = \overline{\langle y, x \rangle}$; but only the real inner product allows us to define the metric $\|x\| = \langle x, x \rangle^{1/2}$. This distinction is particularly important when verifying the correctness of linear operators applied to complex vector spaces, as discussed in §3.2.2. TFOCS provides a function `tfocs_dot.m` that computes the correct inner product for all real and complex vectors and spaces. The function `tfocs_normsq.m` is defined as well, and implements $\|x\|^2$ in a manner consistent with that inner product.

Secondly, note that care must be taken when constructing linear operators that map from real to complex vector spaces, or vice versa. The complex-to-real direction must be implemented properly—specifically, the operator *itself* ensures that the output is real. This is precisely why the `linop_matrix` and `linop_handles` functions require an explicit statement of the intended real/complex behavior. In the case of `linop_matrix`, the function it generates will take the real part for you, when appropriate. But in the case of `linop_handles`, it expects the functions you provide to do this work, and will throw an error if it detects otherwise.

Finally, note that convex and concave functions by their very definition are real-valued, even if they accept complex input. The same caveats given for matrix variables in §4.1 also apply here. For instance, note that the gradient of a function accepting complex input is itself complex.

4.3 Block structure

Suppose for a moment you wish to construct a model whose smooth component is the sum of $M > 1$ simpler smooth functions, like so:

$$\text{minimize} \quad \phi(x) \triangleq \textstyle\sum_{i=1}^{M} f_i(\mathcal{A}_i(x) + b_i) + h(x) \tag{4.3}$$

This can be accomplished using a combination of calls to `tfocs_sum` and `tfocs_scale`:

```
f = tfocs_sum( tfocs_scale( f1, 1, A1, b1 ), ...
               tfocs_scale( f2, 1, A2, b2 ), ...
```

But this approach circumvents more efficient use of linear operators that TFOCS provides; and it is quite cumbersome to boot.

As an alternative, TFOCS allows you to specify a cell array of smooth functions, and a corresponding cell *matrix* of affine operations, like so:

```
smoothF = { f1, f2, f3, f4 };
affineF = { A1, b1 ; A2, b2 ; A3, b3 ; A4, b4 };
[ x, out ] = tfocs( smoothF, affineF, nonsmoothF );
```

Note the use of both commas and semicolons in `affineF` to construct a 4×2 cell array: the number of rows equals the number of smooth functions provided.

Now consider the following case, in which the optimization variable has Cartesian structure, and the nonsmooth function is separable:

$$\text{minimize} \quad \phi(x) \triangleq f(\textstyle\sum_{j=1}^{N} \mathcal{A}_j(x^{(j)}) + b) + \sum_{j=1}^{N} h_j(x^{(j)}) \tag{4.4}$$

To accommodate this case, TFOCS allows the affine operator matrix to be extended *horizontally*:

```
affineF = { A1, A2, A3, A4, b };
nonsmoothF = { h1, h2, h3, h4 };
[ x, out ] = tfocs( smoothF, affineF, nonsmoothF );
```

The number of columns in the cell array is *one greater* than the number of nonsmooth functions, due to the presence of the constant offset `b`. The return value `x` will be a four-element cell array; likewise, if we were to specify an initial point `x0`, we must provide a cell array of four elements.

The logical combination of these cases yields a model with multiple smooth functions, linear operators, and nonsmooth functions:

$$\text{minimize} \quad \phi(x) \triangleq \textstyle\sum_{i=1}^{M} f_i(\sum_{j=1}^{N} \mathcal{A}_{ij}(x^{(j)}) + b) + \sum_{j=1}^{N} h_j(x^{(j)}) \tag{4.5}$$

A corresponding TFOCS model might look like this:

```
smoothF = { f1, f2 };
affineF = { A11, A12, A13, A14, b1 ; A21, A22, A23, A24, b2 };
nonsmoothF = { h1, h2, h3, h4 };
[ x, out ] = tfocs( smoothF, affineF, nonsmoothF );
```

Again, the number of rows of `affineF` equals the number of smooth functions, while the number of columns equals the number of nonsmooth functions *plus one*.

The above are the basics. To that, we have added some conventions that, we hope, will further simplify the use of block structure:

- The scalar value `0` can be used in place of any entry in the affine operator matrix; TFOCS will determine its proper dimension if the problem is otherwise well-posed.
- Similarly, the scalar value `1` can be used in place of any linear operator to represent the identity operation $\mathcal{A}_{ij}(x) \equiv x$.
- Real matrices can be used in place of linear operators; they will be converted to linear operators automatically. (You must convert complex matrices yourself, so you can properly specify the real/complex behavior.)

- If all of the constant offsets are zero, the last column may be omitted entirely.
- For a smooth-plus-affine objective $f(\mathcal{A}(x)+b)+\langle c,x\rangle+d$, the TFOCS model is

```
smoothF = { f, smooth_linear( 1 ) };
affineF = { A, b ; linop_dot( c ), d };
[ x, out ] = tfocs( smoothF, affineF, nonsmoothF );
```

  In this case, we have provided a simplification: you can omit the `smooth_linear` term and the `linop_dot` conversion, and let TFOCS add them for you:

```
smoothF = f;
affineF = { A, b ; c, d };
[ x, out ] = tfocs( smoothF, affineF, nonsmoothF );
```

  This convention generalizes to the case when you have multiple smooth or nonsmooth functions as well. The rule is this: if the number of rows in the affine matrix is one greater than the number of smooth functions, TFOCS assumes that the final row represents a linear functional.

Many of the `solver_` drivers utilize this block composite structure. You are encouraged to examine those as further examples of how this works. It may seem complicated at first—but we argue that this is because the models *themselves* are complicated. We hope that our cell matrix approach has at least made it as simple as possible to specify the models once they are formulated.

4.4 Block structure and SCD models

For `tfocs_SCD.m`, the composite standard form looks like this:

$$\text{minimize} \quad \sum_{j=1}^{N}\left(\bar{f}_j(x^{(j)})+\tfrac{1}{2}\mu\|x^{(j)}-x_0^{(j)}\|^2\right)+\sum_{i=1}^{M}h_i\Big(\sum_{j=1}^{N}\mathcal{A}_{i,j}(x^{(j)})+b_i\Big) \tag{4.6}$$

In this case, the composite convention is precisely reversed:

- The number of rows of the affine matrix must equal the number of nonsmooth functions h_i, or be one greater. In the latter case, the last row is assumed to represent a linear functional.
- The number of columns must equal the number of objective functions f_j, or be one greater. In the latter case, the last column represents the constant offsets b_i.

It turns out that the composite form comes up quite often when constructing compressed sensing problems in analysis form. Consider the model

$$\text{minimize} \quad \alpha\|Wx\|_1+\tfrac{1}{2}\mu\|x-x_0\|_2^2+h(\mathcal{A}(x)+b). \tag{4.7}$$

where W is any linear operator, $\alpha>0$, and h is prox-capable. At first glance, this problem resembles the SCD standard form (1.4) with $\bar{f}=\alpha\|Wx\|_1$, but $\bar{f}$ is not prox-capable. By rewriting it as follows,

$$\text{minimize} \quad 0+\tfrac{1}{2}\mu\|x-x_0\|_2^2+h(\mathcal{A}(x)+b)+\alpha\|Wx\|_1 \tag{4.8}$$

it is now in *composite* SCD form (4.6) with $(M,N)=(2,1)$; specifically,

$$\bar{f}_1(x)\triangleq 0, \quad h_1(y_1)\triangleq h(y_2), \quad h_2(y_2)\triangleq \alpha\|y\|_1, \quad (\mathcal{A}_1,b_1)\triangleq(\mathcal{A},b), \quad (\mathcal{A}_2,b_2)\triangleq(W,0) \tag{4.9}$$

So this problem may indeed be solved by `tfocs_SCD.m`. In particular, the conjugate $h_2^*(z)$ is the indicator function of the norm ball $\{z\,|\,\|z\|_\infty\le\alpha\}$. The code might look like this:

```
affineF = { A, b ; W, 0 };
dualproxF = { hstar, proj_linf( alpha ) };
[ x, out ] = tfocs_SCD( 0, affineF, dualproxF );
```

where, as its name implies, `hstar` implements the conjugate h^*. This technique is used in solvers such as `solver_sBPDN_W.m` and `solver_sBPDN_TV.m`.

This technique generalizes to $\bar{f}=\sum_{i=1}\alpha_i\|W_i\|$ in a natural fashion.

4.5 Scaling issues

With the SCD model, every constraint corresponds to a dual variable. Consider the model in (4.7) where h is the indicator function of the zero set; this is equivalent to imposing the constraint that $\mathcal{A}(x) + b = 0$. The SCD model will create two dual variables, λ_1 corresponding to the constraint $\mathcal{A}(x) + b = y_1$ and λ_2 corresponding to $Wx = y_2$.

The negative Hessian of the smooth part of the dual function is bounded (in the PSD sense) by the block matrix $\frac{2}{\mu}\begin{pmatrix} \mathcal{A}\mathcal{A}^T & 0 \\ 0 & WW^T \end{pmatrix}$. Thus the Lipschitz constant is given by $L = \frac{2}{\mu}\max(\|\mathcal{A}\mathcal{A}^T\|, \|WW^T\|)$. Intuitively, λ_1 has scale $\|\mathcal{A}\mathcal{A}^T\|$ and λ_2 has scale $\|WW^T\|$. If these scales differ, then because the Lipschitz constant is limited by the small scale variable, the step sizes will be very small for the variable with the large scale. This is similar to the phenomenon of a "stiff" problem in differential equations.

Luckily, the fix is quite easy. Recall the α parameter from (4.7), and note that it does not affect the Lipschitz constant. This suggests that we solve the problem using $\hat{\alpha}\|\hat{W}x\|_1 = \alpha\|Wx\|_1$ where $\hat{W} = W\|\mathcal{A}\|/\|W\|$ and $\hat{\alpha} = \alpha\|W\|/\|\mathcal{A}\|$. This ensures that $\hat{W}$ and $\mathcal{A}$ have the same scale.

In general, the user must be aware of this scaling issue and implement the fix as suggested above. For some common solvers, such as `solver_sBPDN_W` and `solver_sBPDN_TV`, it is possible to provide $\|\mathcal{A}\|^2$ via the `opts` structure and the solver will perform the scalings automatically.

4.6 Continuation

Continuation is a technique described in [1] to systematically reduce the effect of the nonzero μ parameter used in the TFOCS SCD model. The software package includes the file `continuation.m` which implements continuation. For convenience, `tfocs_SCD.m` automatically uses continuation when specified in the options.

To turn on continuation, set `opts.continuation = true`. To specify further options to control how continuation is performed, call `tfocs_SCD` with one extra parameter `continuationOptions`, which is a structure of options used in the same way as `opts`. As in § 2.3.2, you may call the continuation solver with no options (`continuation()`) to see a list of available options for `continuationoOptions`.

The continuation technique requires solving several SCD problems, but it is often beneficial since it allows one to use a larger value of μ and thus the subproblems are solved more efficiently.

4.7 Custom vector spaces

We are currently experimenting with giving TFOCS the capability of handling custom vector spaces defined by user-defined MATLAB objects. This is useful when the iterates contain a kind of structure that is not easily represented by MATLAB's existing dense and sparse matrix objects. For example, in many sparse matrix completion problems, it is advantageous to store the iterates in the form $S + \sum_{i=1}^{r} s_i v_i w_i^T$, where S is sparse (even zero) and the summation represents a low-rank matrix stored in dyadic form.

The basic idea is this: we define a custom MATLAB object that can act like a vector space, giving it support for addition, subtraction, multiplication by scalars, and real inner products. If done correctly, TFOCS can manipulate these objects in the same manner that it currently manipulates vectors and matrices.

Our first attempts will focus on the symmetric and non-symmetric versions of this sparse-plus-low-rank structure. Once these are complete, we will document the general interface so that users can construct their own custom vector spaces. Of course, this is a particularly advanced application so we expect only a handful of experts will join us. But if you are already comfortable with using MATLAB's object system, feel free to contact us in advance with your thoughts.

4.8 Standard form linear and semidefinite programming

The power of the SCD method is apparent when you consider the standard linear program (LP)

$$\underset{x}{\text{minimize}} \quad c^T x \quad \text{such that} \quad Ax = b, x \geq 0.$$

By putting this in the SCD framework, it is possible to solve the LP without ever needing to solve a (possibly very large) system of equations. The package includes the `solver_sLP` solver to cover this standard form. When the LP has more structure, it is likely more efficient to write a special purpose TFOCS wrapper, but the generic LP solver can be very useful for prototyping.

It is similarly possible to solve the standard form semi-definite program (SDP):

$$\underset{X}{\text{minimize}} \quad \langle C, X \rangle \quad \text{such that} \quad \mathcal{A}(X) = b, X \succeq 0$$

and its dual, the linear matrix inequality (LMI) problem:

$$\underset{y}{\text{minimize}} \quad b^T y \quad \text{such that} \quad A_0 + \sum_i y_i A_i \succeq 0$$

where $C, A_0, A_1, \ldots, A_m$ are symmetric (if real) or Hermitian (if complex), and b is real. The solvers `solver_sSDP` and `solver_sLMI` handle these forms.

5 Feedback and support

The current version of TFOCS was written by Michael Grant (`mcg@cvxr.com`) and Stephen Becker (`sbecker@acm.caltech.edu`), with considerable design input and initial prototypes provided by Emmanuel Candès (`candes@stanford.edu`). We welcome your feedback, suggestions for improvement, and bug reports.

If you encounter problems with TFOCS, we encourage you to contact Michael or Stephen. In order for us to effectively evaluate a bug report, we will need the following information:

- The output of the `tfocs_version` command, which provides information about your operating system, your MATLAB version, and your TFOCS version. Just copy and paste this information from your MATLAB command window into your email.
- A description of the error itself. If TFOCS itself provided an error message, please copy the full text of the error output into the bug report.
- If it is at all possible, please provide us with a brief code sample and supporting data that reproduces the error. If that cannot be accomplished, please provide a detailed description of the circumstances under which the error occurred.

We have a strong interest in making sure that TFOCS works well for its users. After all, we use it ourselves! But please note that as with any free software, support is likely to be limited to bug fixes, accomplished as we have time to spare. You are welcome to contact us about paid support contracts if you would like more rapid response or custom enhancements.

If you use TFOCS in published research, we require that you acknowledge this fact in your publication, citing both [1] and the software itself in your bibliography. We also kindly request that you drop us a note and let us know that you have found it useful!

6 Acknowledgments

We are very grateful to many users who have submitted bug reports or simply told us what they do or do not like about the software. In particular, much thanks to Graham Coleman and Ewout van den Berg.

7 Appendix: dual functions

When solving the Smooth Conic Dual formulation, as in Equation (1.4), the user must convert to either the convex dual function (for (1.4)) or to the dual cone (for (1.5)). Both the dual function and dual

Table 1: Common functions and their conjugates

$h(y)$	TFOCS atom	conjugate $h^*(z)$	TFOCS atom of the conjugate
$h(y) = 0 = \iota_{\mathbb{R}^n}$	`prox_0`, `proj_Rn`	$h^*(z) = \iota_{z=0}$	`proj_0`
$h(y) = c$	`smooth_constant`	$h^*(z) = \iota_{z=0}$	`proj_0`
$h(y) = \iota_{\mathbb{R}^n_+}$	`proj_Rplus`	$h^*(z) = h(z)$	`proj_Rplus`
$h(Y) = \iota_{Y \succeq 0}$	`proj_psd`	$h^*(Z) = h(Z)$	`proj_psd`
$h(y) = \|y\|_1$	`prox_l1`	$h^*(z) = \iota_{\|z\|_\infty \leq 1}$	`proj_linf`
$h(y) = \|y\|_\infty$	`prox_linf`	$h^*(z) = \iota_{\|z\|_1 \leq 1}$	`proj_l1`
$h(y) = \|y\|_2$	`prox_l2`	$h^*(z) = \iota_{\|z\|_2 \leq 1}$	`proj_l2`
$h(Y) = \|Y\|_{1,2}$	`prox_l1l2`	$h^*(Z) = \|Z\|_{\infty,2} = \|Z\|_{2\to\infty}$	NA
$h(Y) = \|Y\|_{1,\infty}$	`prox_l1linf`	$h^*(Z) = \|Z\|_{\infty,1} = \|Z\|_{\infty\to\infty}$	NA
$h(Y) = \|Y\|_{tr}$	`prox_nuclear`	$h^*(Z) = \iota_{\|Z\| \leq 1}$	NA
$h(Y) = \|Y\|$	`prox_spectral`	$h^*(Z) = \iota_{\|Z\|_{tr} \leq 1}$	NA
$h(Y) = \mathrm{tr}Y + \iota_{Y \succeq 0}$	`prox_trace`	$h^*(Z) = \iota_{\lambda_{max}(Z) \leq 1}$	NA
$h(y) = \iota_{l \leq y \leq u}$	`proj_box`	$h^*(z) = \sum_i \max(z_i l_i, z_i u_i)$	NA
$h(y) = hl(y)$ see §3.1.1	`prox_hinge`	see §3.1.1	`prox_hingeDual`
$h(Y) = -\log\det X$	`smooth_logdet`	see §3.1.1	NA
$h(y) = c^T x$	`smooth_linear`	$h^*(z) = \iota_{z=c}$	`proj_0(c)`
$h(y) = c^T x + x^T P x/2$	`smooth_quad`	$h^*(z) = \frac{1}{2}\|z - c\|^2_{P^{-1}}$	NA

cone interpretations are equivalent; in this appendix, we briefly review some facts for the dual function interpretation.

The convex dual function (also know as the Fenchel or Fenchel-Legendre dual) of a proper convex function h is given by Equation (1.7): $h^*(z) \triangleq \sup_y \langle z, y\rangle - h(y)$. Let ι_A denote the indicator function of the set A:

$$\iota_I(x) = \begin{cases} 0 & x \in A \\ +\infty & x \notin A \end{cases}.$$

Define the dual norm of any norm $\|\cdot\|$ to be $\|\cdot\|_*$ where

$$\|y\|_* \triangleq \sup_{\|x\| \leq 1, x \neq 0} \langle y, x\rangle.$$

For the ℓ_p norm $\|x\|_p \triangleq (\sum |x_i|^p)^{1/p}$, the dual norm is the ℓ_q norm where $1/p + 1/q = 1$ for $p \geq 1$ and with the convention that $1/\infty = 0$.

With respect to using the software, the most important relation is

$$h(y) = s\|y\| = h^{**}(y) \quad \Longleftrightarrow \quad h^*(z) = \iota_{\{z : \|z\|_* \leq s\}}. \tag{7.1}$$

When h is an indicator function, the proximity operator (1.2) is just a projection, and in the TFOCS package the corresponding atom is prefixed with `proj_` as opposed to `prox_`.

Using (7.1), Table 1 lists below a table of common functions and their convex conjugates, as well as the names of their TFOCS atoms.

We write $\|A\|_{1,p}$ to denote the sum of the p-norms of the rows of a matrix. This is in contrast to the norm$\|A\|_{q\to p} \triangleq \sum_{z \neq 0} \|Az\|_p / \|z\|_q$. The $\|\cdot\|_{1,2}$ norm is also know as the *row-norm* of a matrix. The spectral norm $\|A\|$ is the maximum singular value; the trace norm $\|A\|_{tr}$ (also known as the nuclear norm to the spectral norm) is the dual of the spectral norm (see § 3.1.1). When an atom has not been implemented, it is marked as "NA." These atoms may be added in the future if there is demand for them.

References

[1] S. Becker, E. J. Candès, and M. Grant. Templates for convex cone problems with applications to sparse signal recovery. Technical report, Department of Statistics, Stanford University, 2010. Preprint available at `http://tfocs.stanford.edu/tfocs/paper.shtml`.

[2] E. van den Berg and M. Friedlander. Spot—a linear-operator toolbox. Software and web site, Department of Computer Science, University of British Columbia, 2009. `http://www.cs.ubc.ca/labs/scl/spot/`.

[3] P. L. Combettes and J.-C. Pesquet, Proximal splitting methods in signal processing, in *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, H. H. Bauschke, R. Burachik, P. L. Combettes, V. Elser, D. R. Luke, H. Wolkowicz, Editors. New York: Springer-Verlag, 2010. `http://arxiv.org/abs/0912.3522`