ASTR8150/PHYS8150
Optimization

Fabien Baron

Georgia State University

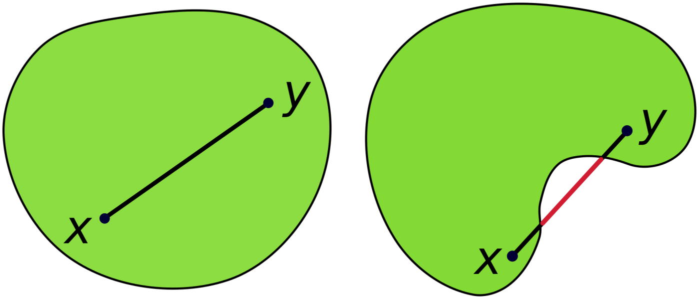
fbaron@gsu.edu

Fall 2025

Optimization Difficulty and Function Structure

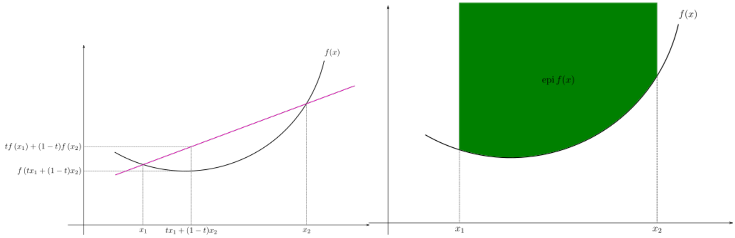
- Inference in a Bayesian framework will most often require the optimization of an objective function (e.g. minimization of the negative log-likelihood)
- The optimization difficulty depends on two properties:
  1. **Convexity of the function:** will straight move within parameter space lead to complex behavior, such as local minima? Does every local minimum equal the global minimum?
  2. **Smoothness of the function:** is there a gradient? Is it continuous and bounded?

Convexity of a set



- In a convex set, for every pair of points within the region, every point on the straight line segment that joins the pair of points is also within the region.
- A set which is hollow or has an indent, for example, a crescent shape, is not convex.

Convexity of a function



- A real-valued function is called convex if the set of points on or above the graph of the function (epigraph) is a convex set.
- For a twice differentiable function of a single variable, if the second derivative is always greater than or equal to zero for its entire domain then the function is convex. Examples: $f(x) = x^2$ or $f(x) = e^x$
- Jensen's inequality: if X is a convex set and $f : X \to \mathbb{R}$, f is convex if:

$$\forall x_1, x_2 \in X, \forall t \in [0,1] : \quad f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2).$$

Smoothness of a function

- The smoothness of a function is a property measured by the number of derivatives it has which are continuous. A smooth function is a function that has derivatives of all orders everywhere in its domain.
- The function $f(x) = |x|^k$ is continuous and k times differentiable at all x. But at $x = 0$ it is not $(k+1)$ times differentiable.
- The norms ℓ_2, ℓ_1 and pseudo-norm ℓ_0 are used in regularization. ℓ_2 is convex, differentiable and smooth. ℓ_1 is convex, differentiable but nonsmooth. ℓ_0 is non-convex and nonsmooth.

Optimization of Convex and Smooth Functions

- Typical form: quadratic or log-sum-exp functions.
- Gradient is Lipschitz continuous:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

- Guarantees:
  - Any local minimum is global.
  - Gradient descent converges linearly with suitable step size.
- Example: least-squares regression.

Optimization of Non-Convex but Smooth Functions

- Many local minima and saddle points.
- Gradient is well-behaved (differentiable, continuous), but:
  - Optimization can get trapped in local minima.
  - Second-order methods (Hessian) can help identify saddles.
- Example: neural network loss surfaces.
- Techniques:
  - Random initialization, momentum, stochasticity.
  - Trust-region or adaptive methods (Adam, LBFGS).

Optimization of Convex but Non-Smooth Functions

- Example: ℓ_1-regularized objectives, such as classic total variation.
- Gradient may not exist everywhere.
- Use of *subgradients* or proximal operators (later in this chapter).
- Guarantees:
  - Still globally convex.
  - Convergence is typically sublinear ($<$ linear $<$ quadratic)
- Methods:
  - Subgradient descent, bundle methods, proximal gradient (ADMM).

Optimization of Non-Convex and Non-Smooth Functions

- Hardest class of problems ("NP-hard")
- Examples:
  - Compressed sensing with ℓ_0, sparse deep learning, robust estimators, combinatorial losses.
- No general guarantees of convergence or global optimality.
- Strategies:
  - Initialization heuristics and regularization.
  - Relaxation ($\ell_0 \to \ell_1$) or smoothing of objectives.
  - Stochastic (e.g., simulated annealing) or global (some NLOpt methods) searches.

Derivative-free optimization methods

- **Derivative-free** optimization methods do not require information about the gradient to work
- These include most MCMC optimization methods (except for Hamiltonian Monte Carlo).
- Non-MCMC derivative free methods are ill-suited to optimize more than 3 parameters at once due to their slowness.
- Among the most popular local optimizer is **Nelder–Mead** method (aka downhill simplex method or amoeba method), which moves points of a polytope of $n+1$ vertices in n-parameter dimensions via reflection, contraction, and expansion steps.
- The NLopt library provides mostly derivative-free algorithms, some of them for global optimization.

Calculating gradients - Overview

- The gradient of the objective function with respect to the parameters to optimize gives the local slope of the function.
- Its usage will speed up optimization & machine learning codes significantly when compared to derivative-free methods.
- But first we need to calculate it
- Three main approaches (also used for Jacobian calculation):
  1. Analytic differentiation
  2. Numerical differentiation
  3. Automatic differentiation (AD)

Analytic Differentiation

- Compute derivatives symbolically from known formulas.
- Produces exact expressions (e.g. $\frac{d}{dx} \sin x = \cos x$).
- Pros:
  - Exact derivatives.
- Cons:
  - Tedious or intractable for complex programs.
  - Requires algebraic manipulation and simplification.

Numerical Differentiation

- Approximates derivative using finite differences:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Pros:
  - Simple to implement.
- Cons:
  - Sensitive to step size h (round-off and truncation errors).
  - Computationally expensive for many parameters (requires multiple function calls).

Automatic Differentiation (AD)

- Computes exact derivatives using the chain rule at the level of elementary operations.
- Express program as a sequence of primitive operations:

$$x_1 = x, \quad x_2 = \sin(x_1), \quad x_3 = x_1 \cdot x_2, \text{ etc.}$$

- AD propagates derivatives through this computational graph.
- Two main modes:
    - Forward mode (tangent propagation)
    - Backward mode (adjoint or reverse accumulation)

Forward Mode AD

- For each intermediate variable v_i, compute both its value and its derivative $\dot{v}_i$.
- Using the chain rule:

$$\dot{v}_i = \sum_j \frac{\partial v_i}{\partial v_j} \dot{v}_j$$

- Efficient when the number of inputs is small (many outputs).
- Example:

$$f(x) = \sin(x^2) \implies \begin{cases} v_1 = x^2, \;\; \dot{v}_1 = 2x \\ v_2 = \sin(v_1), \;\; \dot{v}_2 = \cos(v_1)\dot{v}_1 \end{cases}$$

Backward Mode AD

- Computes derivatives from outputs back to inputs.
- Store the computation graph during the forward evaluation.
- In reverse pass, accumulate sensitivities:

$$\bar{v}_i = \sum_j \frac{\partial v_j}{\partial v_i} \bar{v}_j$$

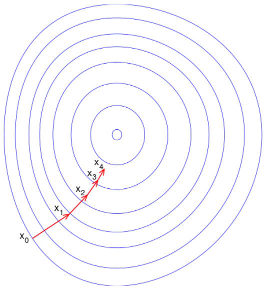
- Efficient when the number of outputs is small (many inputs), e.g. neural networks.
- Example: backpropagation for neural network.

Comparison of gradient obtention methods

Method	**Exactness**	**Performance Cost**	**Scaling**
Analytic	Exact	Lowest	Good but tedious
Numerical	Approximate	High (many evals)	Poor for many vars
AD (Forward)	Exact	Moderate	$\propto$ #inputs
AD (Backward)	Exact	Moderate	$\propto$ #outputs

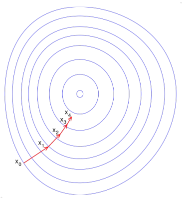
- Modern computer languages or libraries will provide forward and/or backward automatic differentiation with easy setup:
  - Python: Autograd, JAX, Pytorch/Tensorflow
  - Julia: ForwardDiff, ReverseDiff, Zygote, Enzyme, Mooncake
  - Matlab: ADiMat, MAD
  - Fortran and C/C++: Enzyme
  - IDL: nothing

Gradient descent (1)



- Gradient descent is based on the observation that if the multi-variable function $f(\mathbf{x})$ is defined differentiable in a neighborhood of a point $\mathbf{x_0}$, then $f(\mathbf{x})$ decreases "fastest" if one goes from $\mathbf{x_0}$ in the direction of the negative gradient of f at $\mathbf{x_0}$, $-\nabla f(\mathbf{x_0})$.

Gradient descent (2)



- It follows that, if

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \nabla f(\mathbf{x}_n) \tag{1}$$

  for α small enough, then $f(\mathbf{x_n}) \geq f(\mathbf{x_{n+1}})$. In other words, the term $\alpha \nabla f(\mathbf{x})$ is subtracted from $\mathbf{x}$ because we want to move against the gradient, namely down toward the minimum.
- How can we choose α ?
- The Rosenbrock function $f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - {x_1}^2)^2$. has a narrow curved valley which contains the minimum. The bottom of the valley is very flat. Because of the curved flat valley the optimization is zig-zagging slowly with small stepsizes towards the minimum.

Steepest descent using line search

- **Inexact line search** consists in finding $\alpha_k \simeq \underset{\alpha \in \mathbb{R}_+}{\operatorname{argmin}} \; f(\mathbf{x}_k + \alpha \mathbf{d}_k)$
- The **line search** method is one of two basic iterative approaches to find a local minimum $\mathbf{x}^*$ of an objective function $f : \mathbb{R}^n \to \mathbb{R}$ using gradients. The other approach is **trust region**.

Algorithm 1 Steepest descent with line search

procedure $\mathrm{STEEPEST\ DESCENT}(f, \mathbf{x})$
    $k = 0,\ \mathbf{x}_0$ ▷ Iteration counter + initial parameter guess
    while $\|\nabla f(\mathbf{x}_k)\| > \epsilon$ **do** ▷ ϵ = tolerance
        $\mathbf{d_k} = -\nabla f(x_k)$ ▷ Descent direction = Steepest descent
        $\alpha_k \simeq \underset{\alpha \in \mathbb{R}_+}{\operatorname{argmin}} \; f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ ▷ Line search
        $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
        $k = k + 1$
    end while
end procedure

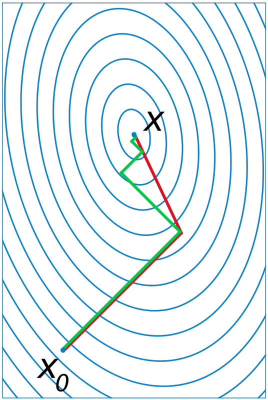
Nonlinear conjugate gradient methods

- Let's pose $\mathbf{g_k} = \nabla f(x_k)$
- The steepest descent direction was $\mathbf{d_k} = -\mathbf{g_k}$
- Conjugate directions differs from the steepest descent by attempting moves based on the history of the previous moves
- The descent direction for nonlinear conjugate gradient methods is

$$\mathbf{d_{k+1}} = -\mathbf{g_{k+1}} + \beta_k \mathbf{d_k}, \quad \mathbf{d_0} = -\mathbf{g_0} \tag{2}$$

- The variation of the gradient is measured by $\mathbf{y_k} = \mathbf{g_{k+1}} - \mathbf{g_k}$
- The Conjugate Gradient update parameter β_k can be updated with different formulas

Conjugate gradient: convergence



- A comparison of the linear convergence of simple gradient descent with optimal step size (in green) and the superlinear convergence of conjugate gradient (in red) for minimizing a quadratic function.

Nonlinear conjugate gradient methods

$\beta_k^{HS} = \dfrac{\mathbf{g}_{k+1}^\top \mathbf{y}_k}{\mathbf{d}_k^\top \mathbf{y}_k}$	(1952)	in the original (linear) CG paper of Hestenes and Stiefel [59]
$\beta_k^{FR} = \dfrac{\|\mathbf{g}_{k+1}\|^2}{\|\mathbf{g}_k\|^2}$	(1964)	first nonlinear CG method, proposed by Fletcher and Reeves [45]
$\beta_k^{D} = \dfrac{\mathbf{g}_{k+1}^\top \nabla^2 f(\mathbf{x}_k)\mathbf{d}_k}{\mathbf{d}_k^\top \nabla^2 f(\mathbf{x}_k)\mathbf{d}_k}$	(1967)	proposed by Daniel [39], requires evaluation of the Hessian $\nabla^2 f(\mathbf{x})$
$\beta_k^{PRP} = \dfrac{\mathbf{g}_{k+1}^\top \mathbf{y}_k}{\|\mathbf{g}_k\|^2}$	(1969)	proposed by Polak and Ribière [84] and by Polyak [85]
$\beta_k^{CD} = \dfrac{\|\mathbf{g}_{k+1}\|^2}{-\mathbf{d}_k^\top \mathbf{g}_k}$	(1987)	proposed by Fletcher [44], CD stands for "Conjugate Descent"
$\beta_k^{LS} = \dfrac{\mathbf{g}_{k+1}^\top \mathbf{y}_k}{-\mathbf{d}_k^\top \mathbf{g}_k}$	(1991)	proposed by Liu and Storey [67]
$\beta_k^{DY} = \dfrac{\|\mathbf{g}_{k+1}\|^2}{\mathbf{d}_k^\top \mathbf{y}_k}$	(1999)	proposed by Dai and Yuan [27]
$\beta_k^{N} = \left(\mathbf{y}_k - 2\mathbf{d}_k \dfrac{\|\mathbf{y}_k\|^2}{\mathbf{d}_k^\top \mathbf{y}_k}\right)^\top \dfrac{\mathbf{g}_{k+1}}{\mathbf{d}_k^\top \mathbf{y}_k}$	(2005)	proposed by Hager and Zhang [53]

Newton optimization method

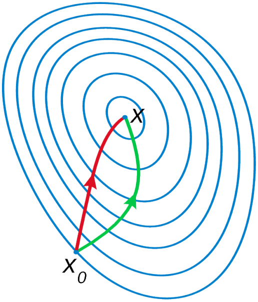


Figure: A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes). Newton's method uses curvature information to take a more direct route.

- Hessian is used to exploit the curvature information

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha[\mathbf{H}f(\mathbf{x}_n)]^{-1}\nabla f(\mathbf{x}_n) \tag{3}$$

- $\alpha \in (0, 1)$, with $\alpha = 1$ the exact form.

Newton-Raphson root-finding and eccentric anomaly

- Newton optimization method and Newton-Raphson's root finding methods are based on similar principles
- Newton-Raphson: $x_{n+1} = x_n - f(x_n)/f'(x_n)$
- Example: the mean anomaly is proportional to time it is an easily measured quantity for an orbiting body. Given the mean anomaly M, find the eccentric anomaly E and the orbital eccentricity e with Kepler's Equation:

$$M = E - e \sin E \tag{4}$$

Better than Newton: quasi-Newton methods

- Also known as variable metric methods, they avoid computing the Hessian then its inverse.
- The Broyden-Fletcher-Goldfarb-Shanno (BFGS) or Davidon-Fletcher-Powell (DFP) algorithms build iteratively approximations of $[\mathbf{H}f(\mathbf{x}_n)]^{-1}$.
- The most successful and well-known quasi-Newton method is the **Limited-memory BFGS (L-BFGS)** that approximates $[\mathbf{H}f(\mathbf{x}_n)]^{-1}\nabla f(\mathbf{x}_n)$ directly and thus can work on large scale problems (millions of variables).
- New gradient descent variants attempt to deal with non-smooth functions (subgradient and bundle method).
- There are variants that deal with constrained miminization (i.e. bounds on variables or linearly tied variables) such as L-BFGS-B. Further refinements led to the VMLM algorithm in OptimPack.

Trust-region method and Levenberg-Marquardt

- Consider the quadratic approximation of function f around x_0:

$$q(\epsilon) \simeq f(x_0) + \nabla f(x_0)\epsilon + \frac{1}{2}\epsilon^T \nabla^2 f(x_0)\epsilon \tag{5}$$

- $q(\epsilon)$ has a close-form mimimum.
- $q(\epsilon)$ remains a good approximation within a given radius, $||e||_2 < r^2$ defines the **trust region** radius r.
- The quadratic approximation predicts a certain reduction in the cost function, Δf_{pred}, which is compared to the true reduction $\Delta f_{\mathrm{actual}} = f(x) - f(x+\epsilon)$. By looking at the ratio $\Delta f_{\mathrm{pred}}/\Delta f_{\mathrm{actual}}$ we can estimate the trust-region size at each iteration, jump to the closed-form minimum within the trust region, and iterate.
- The **Levenberg-Marquardt** algorithm (first published in 1944 by Kenneth Levenberg, rediscovered in 1963 by Donald Marquardt) uses the trust-region approach with conjugate-gradients and Gauss-Newton (Newton optimized for non-linear χ^2). Like conjugate gradient and Newton, these are local optimization codes.

Constrained minimization: the Lagrangian method

- **Constrained** minimization is minimization under equality or inequality constrains. **Bounded** optimization is a special case of constrained optimization where bounds are imposed on parameters (e.g. positivity, or variable within a range). In Bayesian terms, we're imposing a prior.
- A classic example is:

$$(\tilde{x}, \tilde{y}) = \underset{(x,y)\in\mathbb{R}^2}{\operatorname{argmin}} \ (x+y) \quad \text{s. t. } \ x^2 + y^2 = 1$$

- We pose $g(x, y) = x^2 + y^2 - 1$ and the Lagrangian is:

$$\begin{aligned} \mathcal{L}(x, y, \lambda) &= f(x, y) + \lambda \cdot g(x, y) \\ &= x + y + \lambda(x^2 + y^2 - 1). \end{aligned}$$

where λ is a Lagrange multiplier.

Constrained minimization: the Lagrangian method

- The gradient with respect to variables x,y and λ

$$\begin{aligned}\nabla_{x,y,\lambda}\mathcal{L}(x,y,\lambda) &= \left(\frac{\partial\mathcal{L}}{\partial x}, \frac{\partial\mathcal{L}}{\partial y}, \frac{\partial\mathcal{L}}{\partial\lambda}\right) \\ &= \left(1+2\lambda x, 1+2\lambda y, x^2+y^2-1\right)\end{aligned}$$

and therefore:

$$\nabla_{x,y,\lambda}\mathcal{L}(x,y,\lambda) = 0 \quad \Leftrightarrow \quad \begin{cases} 1+2\lambda x = 0 \\ 1+2\lambda y = 0 \\ x^2+y^2-1 = 0 \end{cases}$$

- Solution $x = y = -\frac{1}{2\lambda}$, $\lambda \neq 0$.. Substituting into the last equation we get $\lambda = \pm\frac{1}{\sqrt{2}}$ which implies that the stationary points of $\mathcal{L}$ are $\left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, -\frac{1}{\sqrt{2}}\right), \left(-\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}, \frac{1}{\sqrt{2}}\right)$. And since $f\left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right) = \sqrt{2}$ and $f\left(-\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right) = -\sqrt{2}$. the solution is found.

Constrained minimization: Half-quadratic splitting (1)

- Let's say we want to minimize:

$$\tilde{\boldsymbol{x}} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{H}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \mu\Phi(\boldsymbol{x})$$

- If $\Phi(\boldsymbol{x}) = \|\boldsymbol{W}\boldsymbol{x} - \boldsymbol{w}\|_2^2$, Tikhonov gives us the solution $\boldsymbol{x} = (\boldsymbol{H}^\top \boldsymbol{H} + \boldsymbol{W}^\top \boldsymbol{W})^{-1}(\boldsymbol{H}^\top \boldsymbol{y} + \boldsymbol{W}^\top \boldsymbol{w})$
- Other cases do not have closed-form solution, typical example is $\Phi(\boldsymbol{x}) = \ell_1(\boldsymbol{x}) = \|\boldsymbol{x}\|_1$.
- Splitting methods are methods that split the unconstrained problem into a constrained problem, using two different variables to represent the same one in different functions:

$$\begin{aligned}
\tilde{\boldsymbol{x}} &= \underset{\boldsymbol{x}}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{H}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \mu\Phi(\boldsymbol{z}) \quad s.t. \quad \boldsymbol{z} = \boldsymbol{x} \\
&= \underset{\boldsymbol{x},\boldsymbol{z}}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{H}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \mu\Phi(\boldsymbol{z}) + \frac{\rho}{2} \|z - x\|_2^2
\end{aligned}$$

Constrained minimization: the two subproblems

- So we now want to minimize:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{2} \|\boldsymbol{H}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \mu\Phi(\boldsymbol{z}) + \frac{\rho}{2} \|\boldsymbol{z} - \boldsymbol{x}\|_2^2$$

- $\frac{\rho}{2} \|\boldsymbol{z} - \boldsymbol{x}\|_2^2$ is called an augmented term, and ρ is the augmented penalty hyperparameter.
- The **half-quadratic splitting method** solves iteratively (iteration variable $= k$) the problem with respect to $\boldsymbol{x}$, then $\boldsymbol{z}$:

$$\tilde{\boldsymbol{x}}^{k+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{H}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \frac{\rho}{2} \left\|\tilde{\boldsymbol{z}}^k - \boldsymbol{x}\right\|_2^2 \qquad \boldsymbol{x} \text{ sub-problem}$$

$$\tilde{\boldsymbol{z}}^{k+1} = \underset{\boldsymbol{z}}{\operatorname{argmin}} \frac{\rho}{2} \left\|\boldsymbol{z} - \tilde{\boldsymbol{x}}^{k+1}\right\|_2^2 + \mu\Phi(\boldsymbol{z}) \qquad \boldsymbol{z} \text{ sub-problem}$$

and then increases ρ from initially low values to higher and higher ones.

Constrained minimization: analytical solutions exist

- Why is this easier than the original problem ?

$$\tilde{\boldsymbol{x}}^{k+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \frac{1}{2} \left\| \boldsymbol{Hx} - \boldsymbol{y} \right\|_2^2 + \frac{\rho}{2} \left\| \tilde{\boldsymbol{z}}^k - \boldsymbol{x} \right\|_2^2 \qquad \boldsymbol{x} \text{ sub-problem}$$

$$\tilde{\boldsymbol{z}}^{k+1} = \underset{\boldsymbol{z}}{\operatorname{argmin}} \frac{\rho}{2} \left\| \boldsymbol{z} - \tilde{\boldsymbol{x}}^{k+1} \right\|_2^2 + \mu \Phi(\boldsymbol{z}) \qquad \boldsymbol{z} \text{ sub-problem}$$

- The $\boldsymbol{x}$ sub-problem can be solved by classic Tikhonov.
- The $\boldsymbol{z}$ sub-problem can be solved analytically for some functions, for which we know the solution of the problem:

$$\operatorname{prox}_f(\boldsymbol{x}) = \underset{\boldsymbol{z}}{\operatorname{argmin}} \left(\frac{1}{2} \left\| \boldsymbol{z} - \boldsymbol{x} \right\|_2^2 + f(\boldsymbol{z}) \right)$$

- This solution $\operatorname{prox}_f(\boldsymbol{x})$ is the **proximal operator** for the function f. At each point $\boldsymbol{x}$ it finds a close-by local minimum of f.

Proximal operator for the ℓ_1 norm and positivity

- One can demonstrate that the proximal operator for ℓ_1 norm is:

$$\text{prox}_{\alpha\ell_1}(\boldsymbol{x}) = \underset{\boldsymbol{z}}{\text{argmin}} \left(\frac{1}{2} \|\boldsymbol{z} - \boldsymbol{x}\|_2^2 + \alpha\ell_1(\boldsymbol{z}) \right) = \text{sign}(\boldsymbol{x}) \cdot \max(|\boldsymbol{x}| - \alpha, 0)$$

  where $\cdot$ is the Hadamard product.

- The proximal operator for positivity is the projection onto the positive set:

$$\text{prox}_{I_{\mathbb{R}+}}(\boldsymbol{x}) = \underset{\boldsymbol{z} \in \mathbb{R}^{+n}}{\text{argmin}} \left(\frac{1}{2} \|\boldsymbol{x} - \boldsymbol{z}\|_2^2 \right) = \max(\boldsymbol{x}, 0)$$

Half-quadratic splitting: beyond the 1:1 change of variable

- Half-quadratic splitting only involves analytical steps: the $\boldsymbol{x}$ sub-problem is solved via Tikhonov and the $\boldsymbol{z}$ sub-problem via proximal operators (provided it is known). Generalizing beyond $\boldsymbol{z} = \boldsymbol{x}$, we can also have more complex linear constrains under the form $\boldsymbol{Ax} + \boldsymbol{Bz} + \boldsymbol{c} = 0$.
- How should we solve the total variation problem, i.e. minimize $\frac{1}{2}\|\boldsymbol{Hx} - \boldsymbol{y}\|_2^2 + \mu\ell_1(\boldsymbol{\nabla x})$?
- We only know the proximal operator for $\ell_1(\boldsymbol{z})$ and not for $\ell_1(\boldsymbol{\nabla z})$. So while we could pose $\boldsymbol{z} = \boldsymbol{x}$, we wouldn't know how to solve the $\boldsymbol{z}$ sub-problem in closed form. However we can pose $\boldsymbol{z} = \boldsymbol{\nabla x}$, leading to:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{2}\|\boldsymbol{Hx} - \boldsymbol{y}\|_2^2 + \mu\ell_1(\boldsymbol{z}) + \frac{\rho}{2}\|\boldsymbol{z} - \boldsymbol{\nabla x}\|_2^2$$

Total variation solved via Half-quadratic splitting

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{z}) = \frac{1}{2} \left\| \boldsymbol{H}\boldsymbol{x} - \boldsymbol{y} \right\|_2^2 + \mu \ell_1(\boldsymbol{z}) + \frac{\rho}{2} \left\| \boldsymbol{z} - \boldsymbol{\nabla} \boldsymbol{x} \right\|_2^2$$

- The $\boldsymbol{x}$ sub-problem has the closed form Tikhonov solution:

$$\tilde{\boldsymbol{x}}^{k+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \frac{1}{2} \left\| \boldsymbol{H}\boldsymbol{x} - \boldsymbol{y} \right\|_2^2 + \frac{\rho}{2} \left\| \tilde{\boldsymbol{z}}^k - \boldsymbol{\nabla} \boldsymbol{x} \right\|_2^2 \qquad \boldsymbol{x} \text{ sub-problem}$$

$$\Longrightarrow \boldsymbol{H}^\top (\boldsymbol{H}\boldsymbol{x} - \boldsymbol{y}) - \rho \boldsymbol{\nabla}^\top (\tilde{\boldsymbol{z}}^k - \boldsymbol{\nabla} \boldsymbol{x}) = 0$$

$$\Longrightarrow \tilde{\boldsymbol{x}}^{k+1} = (\boldsymbol{H}^\top \boldsymbol{H} + \rho \boldsymbol{\nabla}^\top \boldsymbol{\nabla})^{-1} (\boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k)$$

- The $\boldsymbol{z}$ sub-problem has a closed form proximal solution:

$$\tilde{\boldsymbol{z}}^{k+1} = \underset{\boldsymbol{z}}{\operatorname{argmin}} \, \mu \ell_1(\boldsymbol{z}) + \frac{\rho}{2} \left\| \boldsymbol{z} - \boldsymbol{\nabla} \tilde{\boldsymbol{x}}^{k+1} \right\|_2^2 \qquad \boldsymbol{z} \text{ sub-problem}$$

$$= \operatorname{prox}_{\frac{\mu}{\rho} \ell_1} (\boldsymbol{\nabla} \tilde{\boldsymbol{x}}^{k+1}) = \operatorname{sign}(\boldsymbol{\nabla} \tilde{\boldsymbol{x}}^{k+1}) \cdot \max(|\boldsymbol{\nabla} \tilde{\boldsymbol{x}}^{k+1}| - \frac{\mu}{\rho}, 0)$$

Making the $\boldsymbol{x}$ step faster: some tricks

- The x sub-problem is the slowest one since it involves:

$$\tilde{\boldsymbol{x}}^{k+1} = (\boldsymbol{H}^\top \boldsymbol{H} + \rho \boldsymbol{\nabla}^\top \boldsymbol{\nabla})^{-1} (\boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k)$$

  but matrix inversion is $\sim \mathcal{O}(N^3)$ process for a $N \times N$ matrix, and thus costly both in memory space and computing time. To work with larger images, tricks to speed up the inversion are used in practice.
- One is to use **sparse arrays** that only store the non-zero elements of $\boldsymbol{H}$ and $\boldsymbol{\nabla}$. The inversion and subsequent multiplication are then faster.
- Using an **orthogonal wavelet basis**, $\boldsymbol{W}$, as a sparsity basis instead of the spatial gradient $\boldsymbol{\nabla}$ since in this case $\boldsymbol{W}^\top \boldsymbol{W} = \alpha \boldsymbol{I}$.
- Another is to employ the backslash operator in Julia (or Matlab). We have $\boldsymbol{X}^{-1} \boldsymbol{Y} = \boldsymbol{X} \backslash \boldsymbol{Y}$, but the latter operation doesn't store the inverted matrix $\boldsymbol{X}^{-1}$; instead it just temporarily stores the parts useful for its multiplication by $\boldsymbol{Y}$.

$$\tilde{\boldsymbol{x}}^{k+1} = (\boldsymbol{H}^\top \boldsymbol{H} + \rho \boldsymbol{\nabla}^\top \boldsymbol{\nabla}) \backslash (\boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k)$$

Making the x step faster: circulant matrices (1)

- It turns out $\nabla^\top \nabla$ and $\boldsymbol{H}^\top \boldsymbol{H}$ can be simplified if ∇ and $\boldsymbol{H}$ are circulant matrices, or concatenation of circulant matrices. This is the case when ∇ is the spatial gradient, and when $\boldsymbol{H}$ is a convolution (= if modeling the imaging done by an optical system).
- A circulant matrix $\boldsymbol{C}$ takes the form:

$$\boldsymbol{C} = \begin{bmatrix} c_1 & c_n & \dots & c_3 & c_2 \\ c_2 & c_1 & c_n & & c_3 \\ \vdots & c_2 & c_1 & \ddots & \vdots \\ c_{n-1} & \vdots & \ddots & \ddots & c_n \\ c_n & c_{n-1} & \dots & c_2 & c_1 \end{bmatrix}.$$

- All circulant matrixes can be written as $\boldsymbol{C} = \boldsymbol{F}\boldsymbol{D}\boldsymbol{F}^\top$ where $\boldsymbol{F}$ is the Fourier transform ($n^2 \times n^2$ if implemented via matrix operations) and $\boldsymbol{D} = \text{diag}(\boldsymbol{F}([c_1 \dots c_n]))$ is diagonal.

Making the x step faster: circulant matrices (2)

- We remind that $\boldsymbol{F}^\top \boldsymbol{F} = \boldsymbol{I}$ since $\boldsymbol{F}^\top = \boldsymbol{F}^{-1}$ and $(\boldsymbol{AB})^\top = \boldsymbol{B}^\top \boldsymbol{A}^\top$
- Since $\boldsymbol{C} = \boldsymbol{FDF}^\top$, $\boldsymbol{C}^\top = ((\boldsymbol{FD})\boldsymbol{F}^\top)^\top = \boldsymbol{F}(\boldsymbol{FD})^\top = \boldsymbol{FD}^\top \boldsymbol{F}^\top$
- Consequently $\boldsymbol{C}^\top \boldsymbol{C} = \boldsymbol{FD}\cancel{\boldsymbol{F}^\top}\cancel{\boldsymbol{F}}\boldsymbol{D}^\top \boldsymbol{F}^\top = \boldsymbol{FD}^2\boldsymbol{F}^\top$ since $\boldsymbol{D}$ is diagonal.
- The application relies on the fact that applying the Fourier transforms $\boldsymbol{F}$ or $\boldsymbol{F}^{-1}$ can be done with very fast FFT algorithms:

$$\begin{aligned}
(\boldsymbol{H}^\top \boldsymbol{H} + \rho \boldsymbol{\nabla}^\top \boldsymbol{\nabla})\tilde{\boldsymbol{x}}^{k+1} &= \boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k \\
(\boldsymbol{F}\boldsymbol{D_H}^2 \boldsymbol{F}^\top + \rho \boldsymbol{F}\boldsymbol{D_\nabla}^2 \boldsymbol{F}^\top)\tilde{\boldsymbol{x}}^{k+1} &= \boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k \\
\boldsymbol{F}(\boldsymbol{D_H}^2 + \rho \boldsymbol{D_\nabla}^2)\boldsymbol{F}^\top \tilde{\boldsymbol{x}}^{k+1} &= \boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k \\
(\boldsymbol{D_H}^2 + \rho \boldsymbol{D_\nabla}^2)\boldsymbol{F}^\top \tilde{\boldsymbol{x}}^{k+1} &= \boldsymbol{F}^{-1}(\boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k) \\
\boldsymbol{F}^\top \tilde{\boldsymbol{x}}^{k+1} &= \frac{\boldsymbol{F}^{-1}(\boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k)}{\boldsymbol{D_H}^2 + \rho \boldsymbol{D_\nabla}^2} \\
\tilde{\boldsymbol{x}}^{k+1} &= \boldsymbol{F}\left(\frac{\boldsymbol{F}^{-1}(\boldsymbol{H}^\top \boldsymbol{y} + \rho \boldsymbol{\nabla}^\top \tilde{\boldsymbol{z}}^k)}{\boldsymbol{D_H}^2 + \rho \boldsymbol{D_\nabla}^2}\right)
\end{aligned}$$

Augmented Lagrangian methods

- Let's say we want to minimize $f(\boldsymbol{x})$ under a sets of constraints on $\boldsymbol{x}$. We saw we could express each of these constrains as $g_i(\boldsymbol{x}) = 0$. Then we just need to minimize the Lagrangian:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \sum_i \lambda_i \cdot g_i(\boldsymbol{x})$$

  where $\boldsymbol{\lambda}$ is a vector this time.
- The idea is to use an **augmented Lagrangian**:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \sum_i \lambda_i \cdot g_i(\boldsymbol{x}) + \frac{\rho}{2} \sum_i |g_i(\boldsymbol{x})|^2$$

  where the augmentation term $\sum_i |g_i(\boldsymbol{x})|^2$ is multiplied by penalty terms ρ.
- The **method of multipliers** solves this by iterating:

$$\boldsymbol{x}^{k+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{x}, \lambda_i^k)$$

$$\lambda_i^{k+1} = \lambda_i^k + \rho^k g_i(\boldsymbol{x}^k) \quad \forall i$$

Example: Method of multipliers for linear constraints

- Let's say we want to minimize $f(\boldsymbol{x})$ under a sets of linear constraints on $\boldsymbol{x}$, we can express them as $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.
- The augmented Lagrangian is:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^\top (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) + \frac{\rho}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$$

- The method of multipliers solves this by iterating:

$$\begin{aligned} \boldsymbol{x}^{k+1} &= \operatorname*{argmin}_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}^k) \\ \boldsymbol{\lambda}^{k+1} &= \boldsymbol{\lambda}^k + \rho(\boldsymbol{A}\boldsymbol{x}^{k+1} - \boldsymbol{b}) \end{aligned}$$

Alternating Direction Method of Multipliers (ADMM)

- Putting all our previous knowledge together, using variable splitting, Lagrangian multipliers, and augmentation terms, we come with ADMM, a method to solve:

$$\underset{\boldsymbol{x},\boldsymbol{z}}{\operatorname{argmin}}\ f(\boldsymbol{x}) + g(\boldsymbol{z}) \quad \text{s.t. } \boldsymbol{Ax} + \boldsymbol{Bz} + \boldsymbol{c} = 0$$

- ADMM solves this by posing:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + g(\boldsymbol{z}) + \boldsymbol{\lambda}^\top(\boldsymbol{Ax} + \boldsymbol{Bz} + \boldsymbol{c}) + \frac{\rho}{2}\left\|\boldsymbol{Ax} + \boldsymbol{Bz} + \boldsymbol{c}\right\|_2^2$$

- And then we iterate:

$$\boldsymbol{x}^{k+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{x}, \boldsymbol{z}^k, \boldsymbol{\lambda}^k) = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f(\boldsymbol{x}) + \frac{\rho}{2}\left\|\boldsymbol{Ax} + \boldsymbol{Bz^k} + \boldsymbol{c} + \frac{\boldsymbol{\lambda^k}}{\rho}\right\|_2^2$$

$$\boldsymbol{z}^{k+1} = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{x}^{k+1}, \boldsymbol{z}, \boldsymbol{\lambda}^k) = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, g(\boldsymbol{z}) + \frac{\rho}{2}\left\|\boldsymbol{Ax^{k+1}} + \boldsymbol{Bz} + \boldsymbol{c} + \frac{\boldsymbol{\lambda^k}}{\rho}\right\|_2^2$$

$$\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \rho(\boldsymbol{Ax}^{k+1} + \boldsymbol{Bz}^{k+1} + \boldsymbol{c})$$

Scaled ADMM

- For convenience we often find in paper the scaled form of ADMM, using the lagragian multiplier $\eta = \lambda/\rho$

$$
\begin{aligned}
\boldsymbol{x}^{k+1} &= \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f(\boldsymbol{x}) + \frac{\rho}{2} \left\| \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{z}^{\boldsymbol{k}} + \boldsymbol{c} + \boldsymbol{\eta}^{\boldsymbol{k}} \right\|_2^2 \\
\boldsymbol{z}^{k+1} &= \underset{\boldsymbol{z}}{\operatorname{argmin}}\, g(\boldsymbol{z}) + \frac{\rho}{2} \left\| \boldsymbol{A}\boldsymbol{x}^{\boldsymbol{k+1}} + \boldsymbol{B}\boldsymbol{z} + \boldsymbol{c} + \boldsymbol{\eta}^{\boldsymbol{k}} \right\|_2^2 \\
\boldsymbol{\eta}^{k+1} &= \boldsymbol{\eta}^{k} + (\boldsymbol{A}\boldsymbol{x}^{k+1} + \boldsymbol{B}\boldsymbol{z}^{k+1} + \boldsymbol{c})
\end{aligned}
$$

- η is just the running sum of residuals.

ADMM residuals and convergence

- Primal residual: $\boldsymbol{r}^{k+1} = \boldsymbol{A}\boldsymbol{x}^{k+1} + \boldsymbol{B}\boldsymbol{z}^{k+1} + \boldsymbol{c}$, i.e. checks if the relation between x and z works.
- Dual residual: $\boldsymbol{s} = \rho \boldsymbol{A}^\top \boldsymbol{B}(\boldsymbol{z}^{\boldsymbol{k+1}} - \boldsymbol{z}^{\boldsymbol{k}})$, i.e. if z has converged
- Convergence properties: global convergence of the iterative procedure to a local optimum is guaranteed. Slow (linear) rate of convergence, slower than gradient-based methods in theory. But the steps can be made very fast.
- Adaptive ADMM modifies ρ based on heuristics such as residual norm balancing or spectral methods.

ADMM: application to TV regularization with Gaussian likelihood

- Like with half quadratic, we pose $f(x) = \frac{1}{2} \|\boldsymbol{Hx} - \boldsymbol{y}\|_2^2$, $g(\boldsymbol{z}) = \ell_1(\boldsymbol{z})$ and $\boldsymbol{z} = \boldsymbol{\nabla x}$, i.e. $\boldsymbol{A} = \boldsymbol{\nabla}$, $\boldsymbol{B} = -\boldsymbol{I}$ and $\boldsymbol{c} = \boldsymbol{0}$.

$$\boldsymbol{x}^{k+1} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f(\boldsymbol{x}) + \frac{\rho}{2} \left\| \boldsymbol{\nabla x} - \boldsymbol{z}^k + \boldsymbol{\eta}^k \right\|_2^2 \rightarrow \text{Tikhonov}$$

$$\boldsymbol{z}^{k+1} = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, g(\boldsymbol{z}) + \frac{\rho}{2} \left\| \boldsymbol{\nabla x}^{\boldsymbol{k+1}} - \boldsymbol{z} + \boldsymbol{\eta}^k \right\|_2^2 \rightarrow \text{Proximal operator for g}$$

$$\boldsymbol{\eta}^{k+1} = \boldsymbol{\eta}^k + \boldsymbol{\nabla x}^{k+1} - \boldsymbol{z}^{k+1}$$

These steps are very similar to Half Quadratic, with just the addition of η in the squared norms.

Consensus ADMM

- Consensus ADMM is solving $\operatorname*{argmin}_{\boldsymbol{x}} \sum_{i=1}^{N} f_i(\boldsymbol{x})$ by recasting it as:

$$\operatorname*{argmin}_{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N,\boldsymbol{x}} \sum_{i=1}^{N} f_i(\boldsymbol{x}_i) \quad \text{s.t. } \boldsymbol{x}_i = \boldsymbol{x}, \quad \forall i.$$

- Used to add more regularization terms. But more importantly if the log-likelihood can be split (into different wavelengths, epochs, etc.), such as is often the case with χ^2 then ADMM allows for **efficient parallelization** of large-scale problems.
- Reminder: k = iteration index, i = term index.

$$\boldsymbol{x}_i^{k+1} = \operatorname*{argmin}_{\boldsymbol{x}_i} f_i(\boldsymbol{x_i}) + \frac{\rho}{2}\left\|\boldsymbol{x}_i - \boldsymbol{x^k} + \boldsymbol{\eta^k}\right\|_2^2 \rightarrow \text{computed in parallel}$$

$$\boldsymbol{x}^{k+1} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i^{k+1} + \boldsymbol{\eta^k}) \rightarrow \textbf{consensus step}\text{, fast}$$

$$\boldsymbol{\eta}^{k+1} = \boldsymbol{\eta}^k + \boldsymbol{x}_i^{k+1} - \boldsymbol{x}^{k+1} \rightarrow \text{fast}$$

Interesting ADMM papers

- Deblurring with Poisson noise: `https://ieeexplore.ieee.org/document/5492199`
- Deblurring with unknown boundaries: `https://ieeexplore.ieee.org/document/6738120`