Optimization with MATLAB

CME 192 Lecture 6

02/11/2026

Stanford University

Announcement

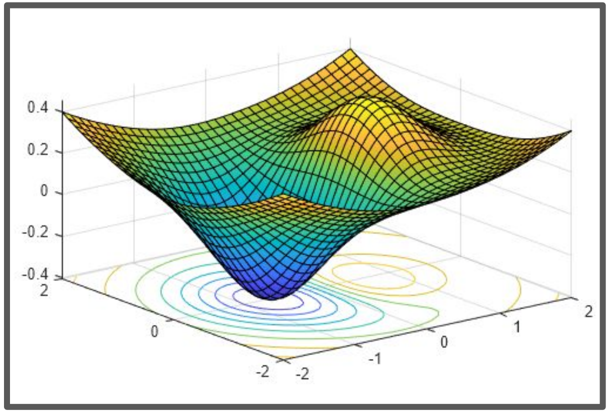
Looking to level up your biomedical data science workflow? Learn how you can import, visualize, analyze, and model biomedical data using interactive tools in MATLAB.

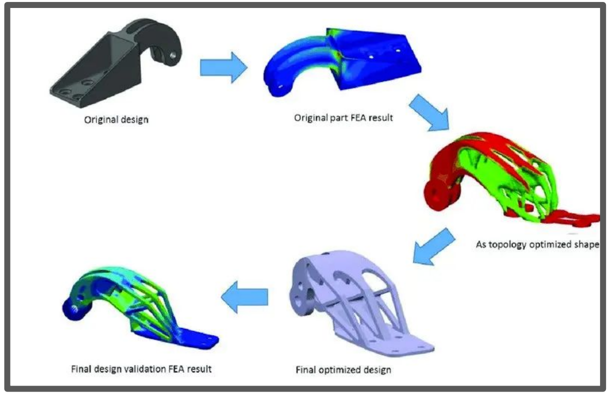
- **Where:** Shriram Center, Room 104
- **When:** 2/17 (Tues), 5:30 PM – 6:20 PM
- **Food/Merch:** Plenty of MATLAB merch and food
- **Presenter: Dr. Reza Fazel-Rezai** is a Senior Scientist at MathWorks and expert in Machine Learning for biomedical signal processing, with over 20 years of experience in academia and industry.

Luma: **https://luma.com/jzwgcarg**

Numerical Optimization

Turning Mathematics into Decisions





Function Derivatives

	Jacobian	Gradient	Hessian
Scalar-valued function	$\frac{\partial}{\partial x} f(x)$	$\nabla f(x) = \left(\frac{\partial}{\partial x} f(x)\right)^T$	$\left[\nabla^2 f(x)\right]_{ij} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}(x)$
Vector-valued function	$\left[\frac{\partial}{\partial x} F(x)\right]_{ij} = \frac{\partial F_i(x)}{\partial x_j}$	$\nabla F = \left(\frac{\partial}{\partial x} F(x)\right)^T$	$\left[\nabla^2 F(x)\right]_{ijk} = \frac{\partial^2 F_i}{\partial x_j \, \partial x_k}(x)$

General Optimization Problem

Objective	$\boxed{\begin{array}{c} \text{minimize } f(x) \\ x \in R^{n_v} \end{array}}$		
Constraints	subject to $Ax \leq b$	$\rightarrow$	Linear inequality constraints
	$A_{eq}x = b_{eq}$	$\rightarrow$	Linear equality constraints
	$c(x) \leq 0$	$\rightarrow$	Nonlinear inequality constraints
	$c_{eq}(x) = 0$	$\rightarrow$	Nonlinear equality constraints
	$l \leq x \leq u$	$\rightarrow$	box constraints

Lagrangian and Karush-Kuhn-Tucker (KKT) optimality conditions

$$\begin{aligned} \underset{x \in R^{n_v}}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g(x) \leq 0 \\ & h(x) = 0 \end{aligned}$$

$$\text{Lagrangian: } L(x, \lambda, \mu) = f(x) + \lambda^T g(x) + \mu^T h(x)$$

$$\begin{aligned} \nabla_x L(x^*, \lambda^*, \mu^*) = 0 \quad & \rightarrow \text{ Stationarity} \\ g(x^*) \leq 0 \quad & \rightarrow \text{ Primal feasibility} \\ h(x^*) = 0 \quad & \\ \lambda^* \geq 0 \quad & \rightarrow \text{ Dual feasibility} \\ \lambda^{*T} g(x^*) = 0 \quad & \rightarrow \text{ Complementary slackness} \end{aligned}$$

Lagrangian and KKT Conditions: What and Why

- Lagrangian: $L(x,\lambda,\mu) = f(x) + \lambda^T g(x) + \mu^T h(x)$
  - Idea: combine the objective and constraints using multipliers (λ for inequalities, μ for equalities).
- KKT conditions (at a candidate optimum x^*):
  - Stationarity: $\nabla_x L(x^*,\lambda^*,\mu^*) = 0$
  - Primal feasibility: $g(x^*) \leq 0$, $h(x^*) = 0$
  - Dual feasibility: $\lambda^* \geq 0$
  - Complementary slackness: for each i, $\lambda^*_i g_i(x^*) = 0$ (active $\Leftrightarrow$ $g_i(x^*) = 0$ can have $\lambda^*_i > 0$)
- Why important:
  - They generalize “set derivative to zero” to constrained optimization.
  - Most constrained solvers (interior-point, SQP, active-set) are designed to satisfy the KKT system.
  - Multipliers provide sensitivity information (“shadow prices”) and help identify active constraints.

Nonlinear System of Equations

Find $x \in R^n$ such that

$$F(x) = 0$$

where $F: R^n \rightarrow R^m$ is continuously differentiable, nonlinear function.

- Solution methods are iterative, in general, which require initial guess and convergence criteria.
- Solution is not guaranteed to exist.
- If the solution, it is not necessarily unique. The solution found depends heavily on the initial guess.

Derivative-Free Methods (n=m=1)

- Derivative-free methods: use only evaluations of f(x) (no f′(x)).
- Bisection (root finding): start with an interval [a,b] where f(a)·f(b) < 0; repeatedly halve the interval to keep bracketing a sign change.
- Fixed point iteration: rewrite the equation as x = g(x); iterate x_{k+1} = g(x_k) until convergence (depends on the choice of g).
- MATLAB: **`fzero`** (for scalar functions)
- **`[x,fval,exitflag,output] = fzero(fun,x0,options)`**
- **`fzero`** can take **`x0`** (a guess) or **`[a,b]`** (a bracket). It searches for a sign change and refines the root (bisection + secant + interpolation).

```
% Example: solve x^2 - 2 = 0
fun = @(x) x.^2 - 2;

% Use an initial guess
[x,fval,exitflag,output] =
fzero(fun,1);

% Or provide a bracket [a,b] with a
sign change
[x,fval] = fzero(fun,[0,2]);
```

Gradient-Based Methods (n=m=1)

- Gradient-based methods: use evaluations of f(x) and its derivative f′(x).
- Newton's method (tangent-line root): $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$
- Near a simple root, Newton's method can converge very quickly, but it can fail if the initial guess is poor or f′(x_k) is near zero.
- Secant method (finite-difference derivative):
- Approximate

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

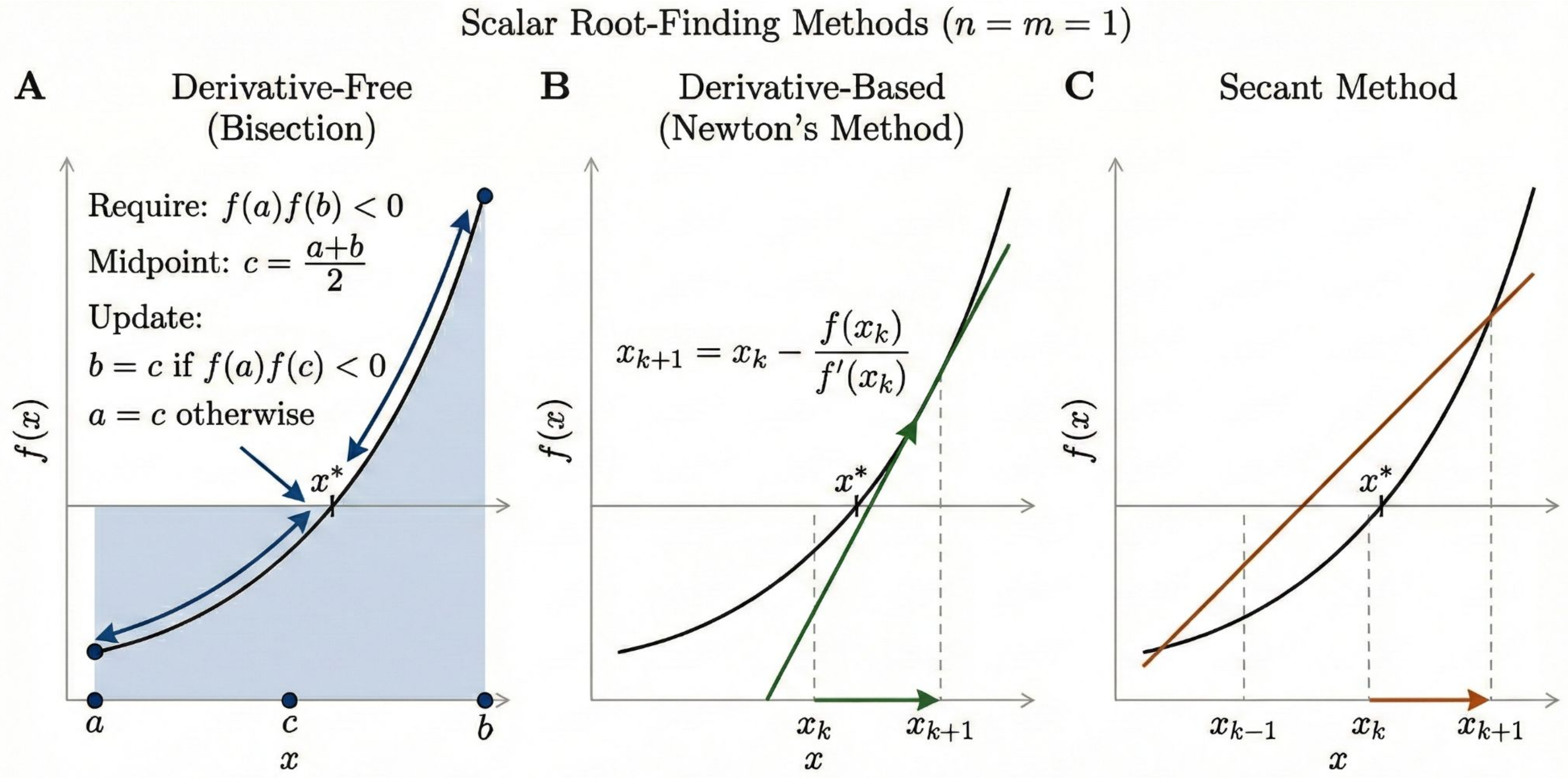
- Update:

$$x_{k+1} = x_k - \frac{f(x_k)\,(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

- Secant uses only f(x) evaluations (no analytic derivative) but needs two starting points.

```
% Newton's method
f  = @(x) x.^2 - 2;
df = @(x) 2*x;
x = 1;
for k = 1:10
  x = x - f(x)/df(x);
end

% Secant method
x0 = 0; x1 = 2;
for k = 1:10
  x2 = x1 -
f(x1)*(x1-x0)/(f(x1)-f(x0));
  x0 = x1; x1 = x2;
end
x = x1;
```



Requires only evaluations of f(x)

Requires functions and derivative evaluations

Approximates derivatives using a finite difference between two iterations

Quiz

```
[x,fval,exitflag,output] = fzero(fun,x0,options)
```

Try solving $x^2 = 0$ and $x^2 + 1 = 0$ using **`fzero`**. What do you get? How would you explain the results?

Quiz

```
[x,fval,exitflag,output] = fzero(fun,x0,options)
```

Try solving $x^2 = 0$ and $x^2 + 1 = 0$ using **`fzero`**. What do you get? How would you explain the results?

Solution

```
[x2,fval2,exitflag2,output2] = fzero(@(x) x^2,4);
[x3,fval3,exitflag3,output3] = fzero(@(x) x^2+1,4);
```

`fzero` first finds an interval containing **`X0`** where the function values of the interval endpoints differ in sign, then searches that interval for a zero.

General case

Derivative-free methods

- Requires function, F(x), evaluations
- Fixed point iteration, Secant method, etc

Gradient-based methods

- Requires function and Jacobian evaluations
- Newton-Raphson method
- Gauss-Newton and Levenberg-Marquardt (nonlinear least squares)
- Can use finite difference approximations to gradients instead of analytic gradients (only requires function evaluations)

`fsolve` **(for vector-valued functions)**

- Gradient-based
- `[x,fval,exitflag,output,jac] = fsolve(fun,x0,options)`
- Algorithms: standard trust region (default), trust region reflexive, Gauss-Newton, Levenberg-Marquardt

General case (F(x) = 0)

- **Goal**: find $x \in R^n$ such that $F(x) = 0$, where $F: R^n \longrightarrow R^m$.
- **Derivative-free methods** (only F(x) evaluations):
  - Fixed point iteration.
  - Multidimensional secant / Broyden-type updates (build an approximate Jacobian without analytic derivatives).
- **Gradient-based methods** (use Jacobian J(x)):
  - Newton-Raphson: solve $J(x_k)\,\Delta x = -F(x_k)$, then $x_{k+1} = x_k + \Delta x$. Intuition: use the best local linear approximation of FFF to jump directly to where that linear model predicts the root is.
  - For nonlinear least squares min $\|F(x)\|^2$: Gauss-Newton and Levenberg-Marquardt. Intuition: instead of solving $F(x) = 0$ F(x)=0 directly, reduce the squared residual step-by-step using curvature information to guide efficient descent.

```
function [F,J] = myfun(x)
F = [x(1)^2 + x(2) - 37;
     x(1) - x(2)^2 - 5];
J = [2*x(1), 1;
     1,      -2*x(2)];
end

x0 = [1; 1];
opts =
optimoptions('fsolve','SpecifyObjectiv
eGradient',true,'Display','iter');
[x,fval,exitflag,output,jac] =
fsolve(@myfun,x0,opts);
```

General case (F(x) = 0)

- **MATLAB**: `fsolve` (vector-valued functions)
- `[x,fval,exitflag,output,jac] = fsolve(fun,x0,options)`
- Providing the Jacobian (second output of fun) can reduce function evaluations and improve robustness.
- Roughly speaking:
  - If you provide the Jacobian, fsolve uses a trust-region Newton method (dogleg algorithm).
    - A trust-region method builds a local linear/quadratic model of F and restricts each step to a region where that model is considered reliable.
    - The dogleg strategy blends two directions: the steepest-descent direction (safe but slow) and the full Newton step (fast but potentially unstable), choosing a step that stays inside the trust region.
  - If you do not provide the Jacobian, fsolve approximates it using finite differences (and can optionally update it quasi-Newton style)
    - A quasi-Newton method approximates the Jacobian numerically and updates that approximation at each step using changes in x and F(x), avoiding analytic derivatives.
- Intuition overall: `fsolve` builds a local linear model of your nonlinear system and carefully chooses steps that are large enough to converge quickly but small enough to remain stable.

Types of Optimization Solvers

Derivative-free (only function evaluations)

- Brute force
- Genetic algorithms
- Finite difference computations of gradients/Hessians
- Usually require very large number of function evaluations

Gradient-based (most popular, function and gradient evaluations)

- Finite difference computations of Hessians
- SPD updates to build Hessian approximations (BFGS)

Hessian-based (function, gradient, and Hessian evaluations)

- Hessians is usually difficult/expensive to compute, although often very sparse.
- Second-order optimality conditions

Types of Optimization Solvers (continued)

Interior Point Methods

- Iterates always strictly feasible
- Use barrier functions to keep iterates away from boundaries
- Sequence of optimization problems

Active set methods

- Active set refers to the inequality constraints active at the solution.
- There are possibly infeasible iterates when constraints are nonlinear.
- Minimize problem for given active set, then update active set based on Lagrange multipliers

Globalization:

- Techniques to make optimization solver globally convergent (convergent to some local minima from any starting point)
- Trust region methods, line search methods

MATLAB Solvers

- Linear Program (LP):
  `linprog`
- Binary Integer Linear Program:
  `bintprog`
- Integer Linear Program:
  `intlinprog`
- Quadratic Program (QP):
  `quadprog`

$$\begin{array}{lll} \underset{x \in R^{n_v}}{\text{minimize}} & f^T x & \text{LP} \\ \text{subject to} & Ax \leq b & \\ & A_{eq}x = b_{eq} & \\ & l \leq x \leq u & \end{array}$$

$$\begin{array}{lll} \underset{x \in R^{n_v}}{\text{minimize}} & \frac{1}{2}x^T Hx + f^T x & \text{QP} \\ \text{subject to} & Ax \leq b & \\ & A_{eq}x = b_{eq} & \\ & l \leq x \leq u & \end{array}$$

- Unconstrained optimization:

`fminsearch, fminunc`

$$\underset{x \in R^{n_v}}{\text{minimize}} \quad f(x)$$

- Linearly constrained optimization:

`fminbnd, fmincon`

$$\begin{aligned} \underset{x \in R^{n_v}}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & Ax \leq b \\ & A_{eq}x = b_{eq} \\ & l \leq x \leq u \end{aligned}$$

- Nonlinear constrained optimization:

`fmincon, fseminf, fgoalattai`
`fminimax`

fminunc (Rosenbrock)

Unconstrained optimization with analytic gradient:

```
function [f,g] = rosenbrock(x)
f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
g = [ -400*x(1)*(x(2)-x(1)^2) - 2*(1-x(1));
       200*(x(2)-x(1)^2) ];
end

x0 = [-1.2; 1];
opts = optimoptions('fminunc','Algorithm','quasi-newton', ...
    'SpecifyObjectiveGradient',true,'Display','iter');
[x,fval,exitflag,output,grad,hess] = fminunc(@rosenbrock,x0,opts);
```

Notes:

- SpecifyObjectiveGradient = true uses the returned g instead of finite differences.
- output provides iteration counts and termination details; exitflag reports the exit condition.

`fmincon` (bounds + nonlinear constraint)

Nonlinear constrained optimization:

```
fun   = @(x)  (x(1)-1)^2 + (x(2)-2)^2;
% Nonlinear constraint: x(1)^2 + x(2)^2 <= 1
nonlcon = @(x) deal( x(1)^2 + x(2)^2 - 1, [] );

x0 = [0.5; 0.5];
lb = [-2; -2];
ub = [ 2;  2];

opts = optimoptions('fmincon','Algorithm','interior-point','Display','iter');
[x,fval,exitflag,output,lambda,grad,hess] = ...
    fmincon(fun,x0,[],[],[],[],lb,ub,nonlcon,opts);
```

Notes:

- nonlcon returns [c, ceq] via deal(c,ceq).
- lambda contains multipliers for bounds and nonlinear constraints.

`fsolve` (provide Jacobian)

Solve F(x) = 0 with analytic Jacobian:

```
function [F,J] = mySystem(x)
% Example: F(x) = [ x(1)^2 + x(2) - 37;  x(1) - x(2)^2 - 5 ]
F = [ x(1)^2 + x(2) - 37;
      x(1) - x(2)^2 - 5 ];
J = [ 2*x(1),  1;
      1,      -2*x(2) ];
end

x0 = [6; 6];
opts = optimoptions('fsolve','SpecifyObjectiveGradient',true,'Display','iter');
[x,fval,exitflag,output,jac] = fsolve(@mySystem,x0,opts);
```

Note: fval is F(x) at the solution x (vector-valued).

How to choose a MATLAB solver (LP / QP / ILP)

- Step 1 — Identify the objective form:

$$\text{Linear (LP):} \quad \min_x \; f^\top x$$
$$\text{Quadratic (QP):} \quad \min_x \; \frac{1}{2}x^\top H x + f^\top x$$
$$\text{Nonlinear:} \quad \min_x \; f(x)$$

- Step 2 — Identify constraint types:

$$\text{Linear inequalities:} \quad Ax \le b$$
$$\text{Linear equalities:} \quad A_{\text{eq}}x = b_{\text{eq}}$$
$$\text{Bounds (box constraints):} \quad \ell \le x \le u$$
$$\text{Nonlinear constraints:} \quad c(x) \le 0, \qquad c_{\text{eq}}(x) = 0$$

- Step 3 — Match to a solver:
  - LP (linear objective + linear constraints): **`linprog`**
  - QP (quadratic objective + linear constraints/bounds): **`quadprog`** $H = H^\top, \qquad H \succeq 0 \quad \text{(convex QP condition)}$
  - Integer/binary variables (some x_i must be integers, often 0/1): **`intlinprog`** (binary is a special case)
  - Nonlinear objective and/or nonlinear constraints: **`fmincon`** (or **`fminunc`** if unconstrained)
  - Systems of equations: **`fzero`** (scalar) or **`fsolve`** (vector-valued)

Choosing a solver

Questions to ask before selecting a method:

- Do you have constraints? (bounds / linear / nonlinear)
- Can you provide derivatives? (gradient / Jacobian / Hessian)
- Is the objective smooth and well-scaled?
- Is this a least-squares structure? (Gauss-Newton / Levenberg-Marquardt)
- Do you need global search vs a local minimum?

Typical mapping (examples):

- Unconstrained: fminunc (or fminsearch for derivative-free)
- Bounds/linear/nonlinear constraints: fmincon
- Nonlinear least squares: lsqnonlin / lsqcurvefit
- Root finding (vector): fsolve
- Global search (examples): MultiStart / GlobalSearch / ga / patternsearch

Call to Optimization Solver

```
[x,fval,exitflag,out,lam,grad,hess] =
solver(f,x0,A,b,Aeq,beq,lb,ub,nlcon,opt)
[x,fval,exitflag,out,lam,grad,hess] = solver(problem)
```

Instead input a problem **structure** with fields

Inputs

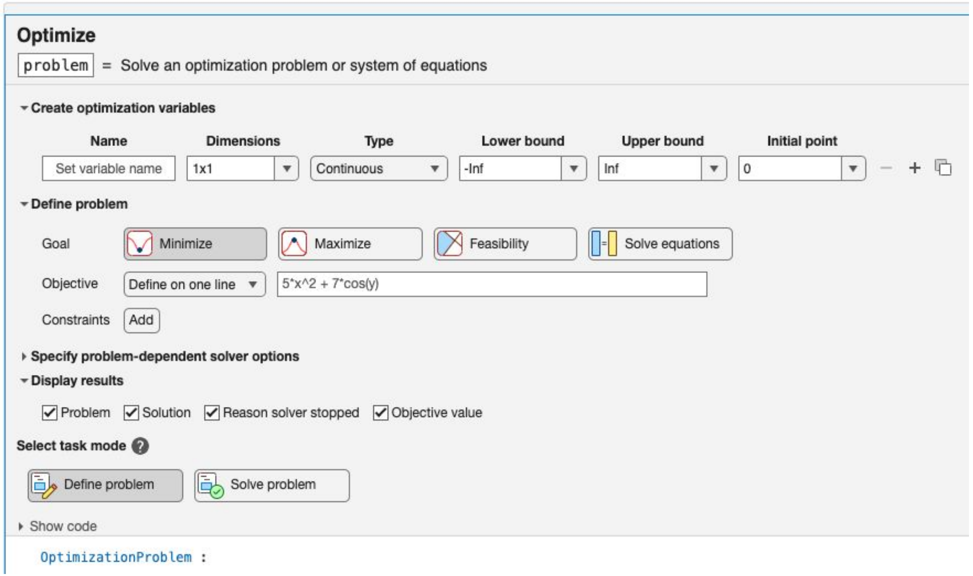
- `f` – function handle (or vector for LP) defining objective function (and gradient)
- `x0` – vector defining initial guess
- `A,b` – matrix, RHS defining linear inequality constraints
- `Aeq`, `beq` – matrix, RHS defining linear equality constraints
- `lb,ub` – vectors defining lower, upper bounds
- `nlcon` – function handle defining nonlinear contraints (and Jacobians)
- `opt` – optimization options structure
- `problem` – structure containing above information

```
[x,fval,exitflag,out,lam,grad,hess] =
solver(f,x0,A,b,Aeq,beq,lb,ub,nlcon,opt)
```

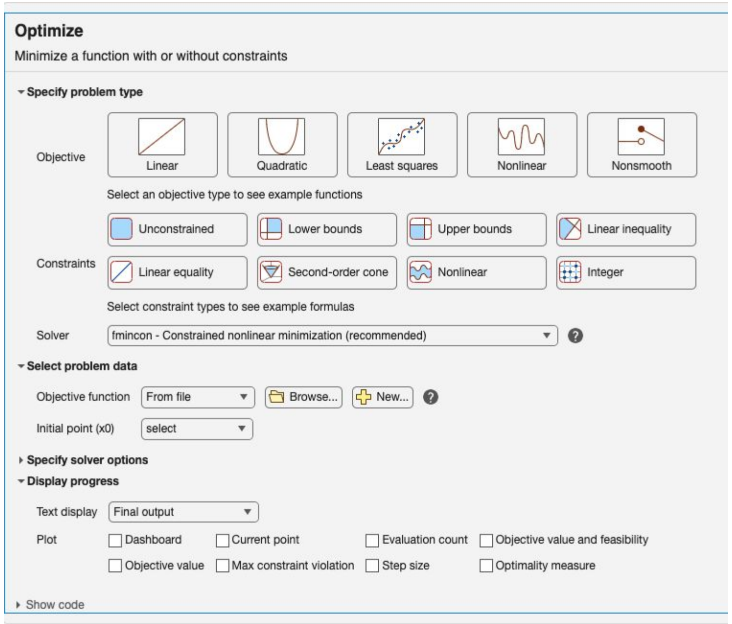
Outputs

- `x` – minimum found
- `fval` – value of objective function at x
- `exitflag` – describes exit condition of solver
- `out` – structure containing output information
- `lam` – structure containing Lagrange multipliers at x
- `grad` – gradient of objective function at x
- `hess` – Hessian of objective function at x

Optimize Live Editor



Problem-Based



Solver-Based

Diagnostics and Robustness

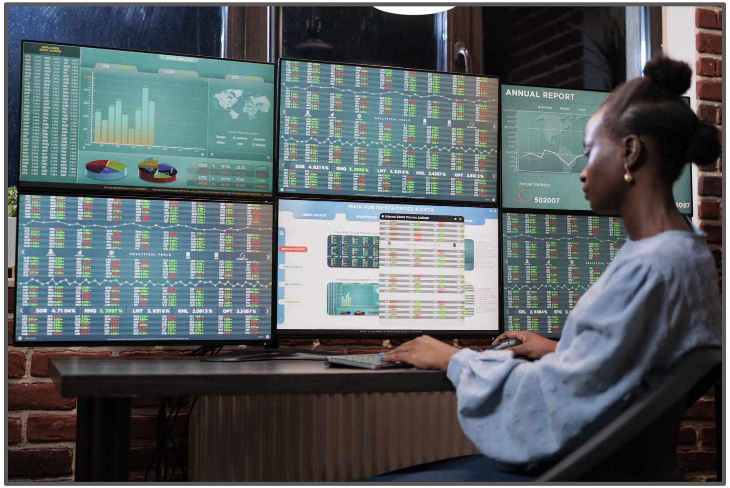
- Use exitflag and output to confirm termination reason (not just fval).
- Try multiple initial guesses for nonconvex problems.
- Scale variables so typical magnitudes are similar.
- Use finite-difference checks for supplied derivatives (and fix mismatches).
- Turn on iteration display / plots during development; turn off for production runs.

Example option patterns:

```
opts = optimoptions('fmincon', ...
    'Display','iter', ...
    'OptimalityTolerance',1e-8, ...
    'StepTolerance',1e-10);
```

Livescript

We are all **PASSIONATE** about making the markets more efficient



Constrained Portfolio Optimization (QP)

- Universe: 9 liquid ETFs (SPY, QQQ, IWM, EFA, EEM, AGG, TLT, GLD, VNQ)
- Data: download daily closes from Stooq API -> align dates -> work with simple daily returns
- Monthly rebalance after a 252-day burn-in (rolling window)
- At each rebalance: estimate rolling μ and Σ from the past 252 returns; add ridge $\Sigma = \Sigma + 1e{-}6 \cdot I$; solve the quadratic program
- To optimize our portfolio holdings, we will solve a convex quadratic program with long-only and a 30% cap on each ETF using `quadprog`
- Backtest includes transaction costs and compares vs equal-weight and buy-and-hold SPY

Optimization Problem (Solved with **`quadprog`**)

Objective (mean–variance tradeoff):

$$\min_{w} \quad \underbrace{\frac{1}{2} w^{\top} \Sigma w}_{\text{Portfolio variance (risk)}} \quad - \quad \underbrace{\gamma \mu^{\top} w}_{\text{Expected portfolio return}}$$

Constraints:

- Fully invested: $\sum w_i = 1$
- Long-only: $w_i \geq 0$
- Diversification cap: $w_i \leq 0.30$

What the equation means?

- First term: “How risky is this portfolio?”
- Second term: “How much return do I expect?”
- The optimizer finds the weights that give the best tradeoff between the two.
- We are not maximizing return alone: we are optimizing risk-adjusted return.

w: portfolio weight vector (fraction of capital in each ETF): *what we invest in*
μ : estimated mean return vector (from past 252 days): *what we expect to earn*
Σ: estimated covariance matrix of returns (risk + correlations): *how risky and correlated the assets are*
γ: risk–return tradeoff parameter (higher gamma means more return-seeking, less risk-averse): *how aggressive the strategy is*

Why quadprog is the right solver and some notes

- Quadratic objective + linear constraints: we should use convex QP
- Fast and reliable vs generic nonlinear solvers

How are μ and Σ computed?

- Use the previous 252 trading days (rolling window) of daily returns μ = sample mean of daily returns for each ETF
- Σ = sample covariance matrix of daily returns
  - Add small ridge term: Σ = Σ + 1e-6I for numerical stability
  - Note: Σ is PSD (the added ridge makes it numerically well-conditioned)
- Re-estimated every month (walk-forward, no look-ahead bias)

Walk-Forward Backtest Mechanics

- Rebalance schedule: first trading day of each day
- No look-ahead: estimates use only returns $t-252 \ldots t-1$
- Hold weights constant between rebalance dates
- Transaction cost on rebalance day: $\text{cost} = c \cdot \sum|\Delta w|$ with $c = 0.0005$
- Track turnover: $\sum|\Delta w|$ (connects "optimal" weights to implementability)
- Baselines: (1) equal-weight $w_i=1/n$ with same rebalance+cost model, (2) buy-and-hold SPY

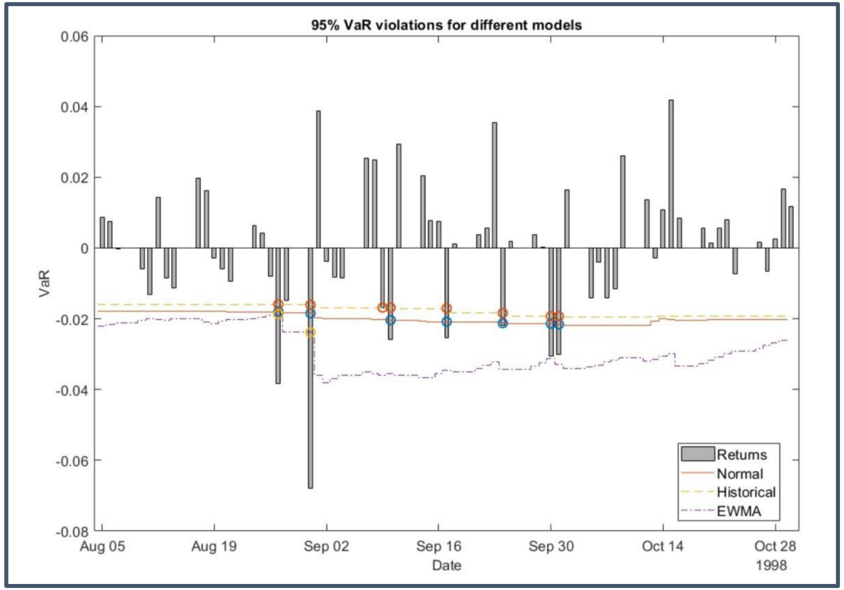


Figure depicting backtesting to compare multiple VaR models. This is NOT related to the livescript code.

Interpreting the Results

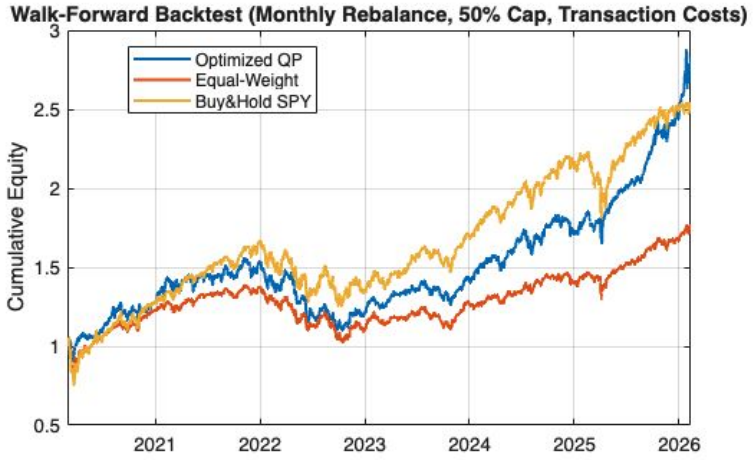
- Optimized QP has lower volatility than SPY (diversification + cap)
- Sharpe improves: risk-adjusted performance is stronger than SPY in this sample
- Raw return can lag SPY in equity-led bull markets because the 30% cap forces diversification
- Turnover (~31%/rebalance here) is modest; transaction-cost drag stays small
- Equal-weight is a useful sanity check: diversified but not risk/return-aware
- Exercise: Scale the axes by the volatility in order to visualize the performance gains of our Optimized QP over SPY

Strategy	AnnReturnPct	AnnVolPct	Sharpe0	MaxDrawdownPct
{'Optimized QP'}	18.566	16.639	1.1073	29.156
{'Equal-Weight'}	9.8519	14.605	0.71675	25.957
{'Buy&Hold SPY'}	16.574	20.641	0.84659	28.32

```
Average turnover per rebalance:

Optimized QP:   43.22%

Equal-Weight:   1.39%
```



Takeaway: this is a “good” result for a constrained mean–variance optimizer. It improves risk-adjusted performance while respecting implementable constraints.