Contents

Convex + Mixed-Integer Optimization Solver (Rust) — Engineering Design Doc (Final) 2
0. Key technical choices (what makes this plausibly "MOSEK-ish") 2
1. Product requirements 3
2. Canonical formulation and sign conventions (make this unambiguous) 3
3. Architecture (Rust crates and module responsibilities) 4
4. Public API (Rust) — proposed 5
5. Continuous solver: HSDE + predictor–corrector (equation-complete) 6
6. Scaling matrices H 8
7. Predictor–corrector RHS definitions (affine + combined) 8
8. Step size selection and neighborhoods 9
9. Linear algebra and KKT solve engineering 10
10. Presolve and scaling 11
11. Cone kernel implementation contract 11
12. Nonsymmetric cones: EXP and POW (full details) 14
13. Scaling objects and KKT H-block representation 16
14. Nonsymmetric scaling (BFGS) — equation-complete spec (EXP/POW) 16
15. 3rd-order correction for nonsymmetric cones (EXP/POW) — implementable 17
16. Termination criteria and statuses 19
17. Mixed-integer design (Branch-and-Bound) 19
18. C ABI design (stable) 20
19. Python bindings and CVXPY integration 21
20. Benchmarking and test harness 21
21. Testing strategy 22
22. Performance engineering checklist 22
23. GPU backend roadmap (architecture hooks) 23
24. Implementation milestones 23
25. References / reading list (practical) 24
26. "Hello world" sketches 24

Convex + Mixed-Integer Optimization Solver (Rust) — Engineering Design Doc (Final)

Version: 0.3
Date: 2026-01-01

> Goal: a **production-grade** convex optimization solver library (LP/QP/SOCP/EXP/POW/PSD) with a **basic discrete layer** (MILP/MIQP/MISOCP initially), written in **Rust**, with **C ABI + Python bindings**, and a **CVXPY interface**.
> The core continuous solver targets **high accuracy, robustness, and speed** in the same algorithm family as modern conic IPM solvers (e.g., Clarabel / MOSEK-class approaches) while being **significantly faster than ECOS** on common workloads.

---

0. Key technical choices (what makes this plausibly "MOSEK-ish")

If you want to be anywhere near MOSEK on convex conic problems, the "secret sauce" is not a single trick; it's a coherent stack:

1. **Homogeneous embedding + infeasibility certificates**
   - Avoids "mysterious" failures on infeasible/ill-posed instances.
   - Gives reliable `PrimalInfeasible` / `DualInfeasible` statuses.
2. **Sparse quasi-definite KKT solves (LDL^T) with robust regularization**
   - Static diagonal regularization to ensure quasi-definiteness even when `P` is PSD / singular.
   - Dynamic pivot regularization to avoid tiny pivots.
   - Symbolic factorization reused across iterations (allocation-free hot loop).
3. **High-quality scaling for cones**
   - **Symmetric cones** (LP/SOC/PSD): **Nesterov–Todd (NT)** scaling.
   - **Nonsymmetric cones** (EXP/POW): **primal-dual secant scaling via BFGS + 3rd-order correction** (Clarabel-style) to keep iterations low and stable.
4. **Aggressive but safe step selection**
   - Fraction-to-boundary for symmetric cones.
   - Backtracking + neighborhood control for nonsymmetric cones.
5. **Engineering discipline**
   - No allocations in the IPM iteration loop.
   - Careful overflow/underflow control (`exp`, `log`, PSD eigens).
   - Comprehensive test harness with published benchmark suites and cross-solver comparisons.

The design below is "equation-complete" for the parts that typically go wrong (Newton system, nonsymmetric cone scaling, and 3rd-order correction), so a strong systems developer can implement it without being an optimization theorist.

---

1. Product requirements

1.1 Must-have capabilities (v1)

Continuous (convex)
Solve problems in either of these equivalent forms:

- Conic form: minimize `c^Tx + d` subject to `Ax + b ∈ K`
- QP+conic form: minimize `(1/2)x^TPx + q^Tx` subject to `Ax + s = b, s ∈ K` with $P \succeq 0$

Cones `K` to support: - Zero / equality cone - Nonnegative cone - Second-order cone (SOC) + rotated SOC via reduction - Exponential cone (EXP) (3D blocks) - 3D power cone (POW, parameter `α ∈ (0,1)`) - PSD cone (SDP): start with dense blocks; add chordal decomposition later

Discrete (mixed-integer)
- Integrality constraints on a subset of variables. - v1 discrete scope: - MILP, MIQP, MISOCP via Branch-and-Bound (B&B) over continuous relaxations - Basic heuristics (rounding+repair, diving)

Integration / packaging - Rust crates: `solver-core`, `solver-mip`, `solver-ffi`, `solver-py` - CVXPY integration package (`cvxpy_solvername`)

Performance / quality - Must beat ECOS on representative QP and SOCP workloads. - Deterministic results given fixed seeds / ordering. - Strong numerical robustness: certificates, stable termination status, informative diagnostics.

1.2 Non-goals (v1)

- Full "Gurobi-class" MILP cut engine in v1.
- Full MISDP in v1.
- Distributed multi-node solving.

1.3 Target users

- Strong developer with some convex optimization background, not an IPM expert.
- Wants predictable behavior and clear failure modes.

---

2. Canonical formulation and sign conventions (make this unambiguous)

2.1 Canonical internal form (continuous)

We standardize internally on:

$$\begin{aligned} \min_{x,s} \quad & \frac{1}{2}x^\top P x + q^\top x \\ \text{s.t.} \quad & Ax + s = b, \\ & s \in K, \end{aligned}$$

with: - $x \in \mathbb{R}^n$ - $s \in \mathbb{R}^m$ - $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $q \in \mathbb{R}^n$ - $P \succeq 0$ (PSD, possibly singular)
Dual variables: - $z \in K^*$ associated with $Ax + s = b$

2.2 Mapping from other common forms

If a user provides: `min c^Tx` s.t. `A x + b ∈ K`, rewrite as: - Let $s = b + Ax$, so $Ax + s = -b$ depending on sign. - **Pick one public convention** and consistently transform into the internal form.

For CVXPY: - CVXPY's canonicalization already produces `A x + s = b`, `s ∈ K`. - Match CVXPY's cone block ordering and dimensions exactly (or implement a strict adapter).

2.3 Cone bookkeeping

Cone K is a Cartesian product of blocks:

$$K = K_1 \times K_2 \times \cdots \times K_B,$$

Each block has: - type (Zero, NonNeg, SOC, PSD, EXP, POW) - dimension (or PSD size) - parameters (α for POW) - contiguous offsets into the global slack/dual vectors `s` and `z`

2.4 Barrier degrees (needed for μ)

Define the barrier parameter (degree) $\nu(K)$ as the sum of block degrees:

- Zero cone: $\nu = 0$
- Nonnegative cone $\mathbb{R}^k_+$: $\nu = k$
- SOC (Lorentz) of dimension d: $\nu = 2$
- PSD cone S^n_+: $\nu = n$
- Exponential cone (3D): $\nu = 3$
- 3D power cone: $\nu = 3$

For the product cone: $\nu = \sum_b \nu(K_b)$.

---

3. Architecture (Rust crates and module responsibilities)

3.1 Crates

- `solver-core`
  - `problem`: data model, validation, IO/parsers
  - `cones`: cone kernels (barrier, scaling, interior tests, step-to-boundary)
  - `ipm`: HSDE + primal-dual IPM loop
  - `linalg`: sparse/dense ops, factorization abstraction, ordering, refinement
  - `presolve`: Ruiz equilibration, reductions, (future) chordal decomposition
  - `util`: logging, timers, deterministic RNG, numeric helpers
- `solver-mip`
  - `bnb`: branch-and-bound tree, branching, node selection, heuristics
  - `cuts`: (future) OA / cuts
- `solver-ffi`: stable C ABI
- `solver-py`: Python bindings (pyo3 or cffi)
- `cvxpy-solvername`: CVXPY plugin adapter

3.2 Data flow

1) Parse / ingest problem

2) Presolve + scaling

3) Continuous solve (HSDE-IPM)

4) If integrality exists: B&B calls continuous solver at nodes

5) Unscale + return result + diagnostics

3.3 GPU readiness (design constraint)

All heavy ops must go through thin traits: - sparse factorization + triangular solves - SpMV / SpMM - BLAS-like kernels

Cone kernels are written to support: - block batching (SOC blocks independent; EXP/POW independent) - optional vectorization / threading - future GPU SoA layouts for 3D cones

---

4. Public API (Rust) — proposed

4.1 Core types

```
pub struct ProblemData {
    pub P: Option<SparseSymmetricCsc<f64>>, // n×n (upper triangle), PSD
    pub q: Vec<f64>,                        // n
    pub A: SparseCsc<f64>,                  // m×n
    pub b: Vec<f64>,                        // m
    pub cones: Vec<ConeSpec>,               // partitions m
    pub var_bounds: Option<Vec<VarBound>>,  // optional bounds on x
    pub integrality: Option<Vec<VarType>>,  // continuous/int/binary
}

pub enum ConeSpec {
    Zero { dim: usize },
    NonNeg { dim: usize },
    Soc { dim: usize },          // (t, x) with dim >= 2
    Psd { n: usize },            // symmetric n×n in svec form
    Exp { count: usize },        // dim = 3*count
    Pow { cones: Vec<Pow3D> },   // each dim=3
}

pub struct Pow3D { pub alpha: f64 }

pub struct SolverSettings {
    pub max_iter: usize,
    pub time_limit_ms: Option<u64>,
    pub verbose: bool,

    pub tol_feas: f64,
    pub tol_gap: f64,
    pub tol_infeas: f64,

    pub ruiz_iters: usize,

    pub static_reg: f64,
    pub dynamic_reg_min_pivot: f64,

    pub threads: usize,

    // determinism / heuristics
    pub seed: u64,

    // GPU (future)
    pub enable_gpu: bool,
}
```

4.2 Solve output

```
pub struct SolveResult {
    pub status: SolveStatus,
    pub x: Vec<f64>,   // primal x (unscaled, recovered x¯)
    pub s: Vec<f64>,   // slack s¯
    pub z: Vec<f64>,   // dual z¯
    pub obj_val: f64,
    pub info: SolveInfo,
}

pub enum SolveStatus {
    Optimal,
    PrimalInfeasible,
    DualInfeasible,
    Unbounded,        // optional; often dual infeasible implies primal unbounded
    MaxIters,
    TimeLimit,
    NumericalError,
}

pub struct SolveInfo {
    pub iters: usize,
    pub solve_time_ms: u64,
    pub kkt_factor_time_ms: u64,
    pub kkt_solve_time_ms: u64,
    pub cone_time_ms: u64,

    pub primal_res: f64,
    pub dual_res: f64,
    pub gap: f64,
    pub mu: f64,

    pub reg_static: f64,
    pub reg_dynamic_bumps: u64,
}
```

---

5. Continuous solver: HSDE + predictor–corrector (equation-complete)

5.1 Homogeneous embedding (QP-native HSDE)

We work with variables (x, s, z, τ, κ) satisfying:

$$\begin{aligned} Px + A^\top z + q\tau &= 0 \\ Ax + s - b\tau &= 0 \\ \frac{1}{\tau}x^\top Px + q^\top x + b^\top z &= -\kappa \\ (s, z, \tau, \kappa) &\in K \times K^* \times \mathbb{R}_+ \times \mathbb{R}_+. \end{aligned}$$

When $\tau > 0$, recover the original variables:

$$\bar{x} = x/\tau, \quad \bar{s} = s/\tau, \quad \bar{z} = z/\tau.$$

5.2 Central path equations (what the IPM follows)

Maintain strict interior: - $s \in \text{int}(K)$, $z \in \text{int}(K^*)$, $\tau > 0$, $\kappa > 0$

Define: - ν = barrier degree (Section 2.4) - $\mu = \frac{\langle s,z\rangle + \tau\kappa}{\nu+1}$ (common HSDE choice)

Centrality conditions: - For **symmetric cone blocks**: scaled complementarity $s \circ z = \mu e$ (Jordan product) - For **nonsymmetric blocks**: barrier centrality $z = -\mu \nabla f(s)$ (equivalently $s = -\mu \nabla f^*(z)$) - Scalar complementarity: $\tau\kappa = \mu$

5.3 Newton system (step directions)

Let $\xi := x/\tau$ (used only in the Jacobian).
Given a right-hand residual vector $d := (d_x, d_z, d_\tau, d_s, d_\kappa)$, compute a Newton-like direction $(\Delta x, \Delta s, \Delta z, \Delta\tau, \Delta\kappa)$ from:

Linearized primal/dual equations

$$\begin{aligned} P\Delta x + A^\top \Delta z + q\Delta\tau &= d_x \\ A\Delta x + \Delta s - b\Delta\tau &= -d_z \\ (2P\xi + q)^\top \Delta x + b^\top \Delta z - (\xi^\top P\xi)\Delta\tau + \Delta\kappa &= -d_\tau \end{aligned}$$

Linearized complementarity

$$H\Delta z + \Delta s = -d_s, \qquad \kappa\Delta\tau + \tau\Delta\kappa = -d_\kappa,$$

where H is the (block-diagonal) scaling matrix (Section 6 / 11).

5.4 Condensation to the quasi-definite KKT solve (what you implement)

Eliminate Δs and $\Delta\kappa$ to obtain a reduced 3-block system:

$$\begin{bmatrix} P & A^\top & q \\ -A & H & b \\ -(q+2P\xi)^\top & -b^\top & \xi^\top P\xi + \kappa/\tau \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta z \\ \Delta\tau \end{bmatrix} = \begin{bmatrix} d_x \\ d_z - d_s \\ d_\tau - d_\kappa/\tau \end{bmatrix}.$$

Then recover:

$$\Delta s = -d_s - H\Delta z, \qquad \Delta\kappa = -\frac{d_\kappa + \kappa\Delta\tau}{\tau}.$$

5.4.1 Two-solve strategy (fast, sparse) Solve two linear systems that share the same left-hand side KKT matrix:

$$\begin{bmatrix} P & A^\top \\ A & -H \end{bmatrix} \begin{bmatrix} \Delta x_1 & \Delta x_2 \\ \Delta z_1 & \Delta z_2 \end{bmatrix} = \begin{bmatrix} d_x & -q \\ -(d_z - d_s) & b \end{bmatrix}.$$

Then compute:

$$\Delta\tau = \frac{d_\tau - d_\kappa/\tau + (2P\xi + q)^\top \Delta x_1 + b^\top \Delta z_1}{\kappa/\tau + \xi^\top P\xi - (2P\xi + q)^\top \Delta x_2 - b^\top \Delta z_2},$$

and finally:

$$\Delta x = \Delta x_1 + \Delta\tau\Delta x_2, \qquad \Delta z = \Delta z_1 + \Delta\tau\Delta z_2.$$

Why this matters: you factorize the sparse KKT matrix once per iteration, then do a small number of triangular solves for multiple RHS vectors.

---

6. Scaling matrices H

6.1 Symmetric cones: Nesterov–Todd (NT) scaling

For symmetric cones (NonNeg, SOC, PSD), compute a scaling such that:

- H is SPD and block-diagonal across cone blocks
- $Hz \approx s$ and $H^{-1}s \approx z$ (exact for the NT scaling)

In implementation terms: for each cone block, build a `ScalingBlock` that can apply: - Hv - $H^{-1}v$
Important performance note:
For SOC blocks with dimension > 4, a dense H block can be expensive and can destroy sparsity. You will likely want a specialized representation (see §10.1) rather than materializing a dense `d×d` matrix for large SOCs.

6.2 Nonsymmetric cones (EXP/POW): primal-dual BFGS scaling

For nonsymmetric cones, we cannot use NT scaling. We use the "two-secant-equation" scaling with a low-rank BFGS update, detailed in §11.4.

---

7. Predictor–corrector RHS definitions (affine + combined)

Define the **linear equation residuals** (from the HSDE equations):

$$\begin{aligned} r_x &= Px + A^\top z + q\tau \\ r_z &= Ax + s - b\tau \\ r_\tau &= \frac{1}{\tau}x^\top Px + q^\top x + b^\top z + \kappa. \end{aligned}$$

7.1 Affine step RHS

Affine step = Newton step toward feasibility with $\mu = 0$.
Set: - $d_x = r_x$ - $d_z = r_z$ - $d_\tau = r_\tau$ - $d_s = s$ - $d_\kappa = \kappa\tau$
Solve for Δ^{aff} using §5.4.
Compute the affine step size α_{aff} (Section 8).

7.2 Corrector parameter σ (recommended)

A robust choice (Clarabel-style):

$$\sigma = (1 - \alpha_{aff})^3.$$

(Alternative classic choice is $\sigma = (\mu_{aff}/\mu)^3$; both work, but the above is simple and stable in practice.)

7.3 Combined step RHS

Combined step = affine step + centering + higher-order correction.
Set: - $d_x = (1-\sigma)r_x$ - $d_z = (1-\sigma)r_z$ - $d_\tau = (1-\sigma)r_\tau$
Scalar complementarity correction:

$$d_\kappa = \kappa\tau + \Delta\kappa^{aff}\Delta\tau^{aff} - \sigma\mu.$$

Cone complementarity correction d_s differs by cone symmetry:

7.3.1 Symmetric cones (Mehrotra correction) Let W be the NT scaling factor and λ the scaled slack/dual "central" element such that in scaled coordinates $\lambda \circ \lambda \approx s \circ z$. (You implement this per cone.)

Define:

$$\eta = (W^{-1}\Delta s^{aff}) \circ (W\Delta z^{aff})$$

(Jordan product; for NonNeg this is elementwise product.)

Then:

$$d_s = W^\top \left(\lambda \backslash \left(\lambda \circ \lambda + \eta - \sigma\mu e\right)\right).$$

Here λ \ v denotes the solution of the Jordan equation $\lambda \circ u = v$ for u.
For NonNeg, this is elementwise division.

7.3.2 Nonsymmetric cones (EXP/POW): 3rd-order correction For nonsymmetric blocks, use:

$$d_s = s + \sigma\mu\nabla f^*(z) + \eta$$

where the higher-order correction is:

$$\eta = -\frac{1}{2}\nabla^3 f^*(z)[\Delta z,\ (\nabla^2 f^*(z))^{-1}\Delta s].$$

Because EXP/POW are 3D, we implement this **exactly and cheaply** (see §11.5).

---

8. Step size selection and neighborhoods

8.1 Basic fraction-to-boundary

Compute maximal step sizes to remain strictly interior:

- $\alpha_s = \min_b \alpha_{max}(K_b, s_b, \Delta s_b)$
- $\alpha_z = \min_b \alpha_{max}(K_b^*, z_b, \Delta z_b)$
- $\alpha_\tau = +\infty$ or $-\tau/\Delta\tau$ if $\Delta\tau < 0$
- $\alpha_\kappa = +\infty$ or $-\kappa/\Delta\kappa$ if $\Delta\kappa < 0$

Then:

$$\alpha = \min\{1,\ 0.99 \cdot \min(\alpha_s, \alpha_z, \alpha_\tau, \alpha_\kappa)\}.$$

8.2 Symmetric cones: closed-form α_max per block

- NonNeg: $\min_{i:\Delta s_i<0} -s_i/\Delta s_i$
- SOC: quadratic boundary solve (Section 11.3.2)
- PSD: eigenvalue bound (Section 11.5.2)

8.3 Nonsymmetric cones: backtracking + neighborhood check

For EXP/POW blocks: - Start $\alpha = 1$. - While not interior for *both* primal and dual: $\alpha \leftarrow \beta\alpha$ with $\beta \approx 0.8$. - Apply final safety factor: $\alpha \leftarrow 0.99\alpha$.

Additionally, when any nonsymmetric cones exist, enforce a **central neighborhood** condition (proximity metric). Implementation choice: - Use the standard proximity metric used in modern nonsymmetric-cone IPMs (Clarabel-style). - If the neighborhood condition fails, reduce α (backtracking) until satisfied.

---

9. Linear algebra and KKT solve engineering

9.1 KKT matrix

Per iteration, we factorize:

$$K = \begin{bmatrix} P & A^\top \\ A & -H \end{bmatrix}.$$

9.2 Quasi-definiteness via static regularization

To guarantee quasi-definiteness (and thus stable LDL^T), add static diagonal shifts:

$$K_\epsilon = \begin{bmatrix} P + \epsilon I & A^\top \\ A & -(H + \epsilon I) \end{bmatrix}.$$

- Choose $\epsilon =$ `static_reg` (e.g., 10^{-9} to 10^{-7} in double precision).
- Keep ϵ deterministic and report it in diagnostics.

9.3 Dynamic pivot regularization

During factorization: - Enforce $|D_{ii}| \geq \epsilon_d$ with $\epsilon_d =$ `dynamic_reg_min_pivot`. - If violated, apply a diagonal bump to the affected pivot.
Record number of dynamic bumps; regressions in this count are a strong sign of numerical issues.

9.4 Factorization backends (Rust)

Define a backend trait so you can swap QDLDL / CHOLMOD / Pardiso / GPU later:

```
### 4.3 Determinism contract (v1)

Determinism matters for CI, regression tests, and reproducible research. The solver must be deterministic **by defa

Continuous solver:
- Use a deterministic sparse ordering by default (e.g., AMD) and keep tie-breaks deterministic.
- Reuse the same symbolic factorization and do not let backend threads change pivot ordering.
- If `threads > 1`, ensure parallel reductions are deterministic (fixed chunking + stable summation order), or docu
threading may slightly change iterates.

Mixed-integer layer:
- Deterministic node selection and branching tie-break rules.
- Expose `seed` for any randomized heuristics, but keep default heuristics deterministic.

Numerical note:
- Deterministic does **not** mean “bitwise identical across CPUs/backends”; it means the algorithmic choices and ti
breaks are fixed so results are stable within a backend/platform.

rust
pub trait LinearSolveBackend {
    type SparseMat;
    type Factor;

    fn symbolic(&mut self, pattern: &Self::SparseMat);
    fn factorize(&mut self, values: &Self::SparseMat) -> Result<Self::Factor, SolveStatus>;
    fn solve_in_place(&self, factor: &Self::Factor, rhs: &mut [f64]);
    fn spmv(&self, a: &Self::SparseMat, x: &[f64], y: &mut [f64], alpha: f64, beta: f64);
}
```

9.5 Multi-RHS solves

In §5.4, the two-solve strategy requires multiple RHS solves per iteration: - factorize once - solve RHS matrix with 2 (or more) right-hand sides

Implement as: - either repeated triangular solves - or a packed multi-RHS solve if backend supports it

9.6 Iterative refinement

Optional but valuable: - compute residual $r = b - Kx$ - solve $K\delta = r$ - update $x \leftarrow x + \delta$

This is especially valuable for: - GPU mixed precision - ill-conditioned instances

---

10. Presolve and scaling

10.1 Ruiz equilibration

Apply Ruiz scaling to improve conditioning: - scale rows/cols of `A` (and optionally `P`) - iterate `ruiz_iters` times - store diagonal scalings to unscale solution and certificates

10.2 Basic presolve (v1)

- Validate cone partitions and dimensions.
- Remove empty cones.
- Normalize power cone α into (0,1).
- Detect fixed variables from bounds (optional).
- Normalize/clean NaNs/infs in input (reject invalid).

10.3 PSD chordal decomposition (v2)

For sparse PSD blocks: - chordal decomposition into cliques - add consistency constraints - clique merging heuristics - “compact / range-space” conversions (design for later)

---

11. Cone kernel implementation contract

11.1 Common trait

```
pub trait ConeKernel {
    fn dim(&self) -> usize;

    fn is_interior_primal(&self, s: &[f64]) -> bool;
    fn is_interior_dual(&self, z: &[f64]) -> bool;

    fn step_to_boundary_primal(&self, s: &[f64], ds: &[f64]) -> f64;
    fn step_to_boundary_dual(&self, z: &[f64], dz: &[f64]) -> f64;

    fn barrier_grad_primal(&self, s: &[f64], grad_out: &mut [f64]);
    fn barrier_hess_apply_primal(&self, s: &[f64], v: &[f64], out: &mut [f64]);

    // For symmetric cones, these can delegate to primal.
    fn barrier_grad_dual(&self, z: &[f64], grad_out: &mut [f64]);
    fn barrier_hess_apply_dual(&self, z: &[f64], v: &[f64], out: &mut [f64]);

    // Update scaling block used to build H for this cone block.
    fn scaling_update(&self, s: &[f64], z: &[f64], out: &mut ScalingBlock);
}
```

11.2 Numerical conventions

- Any `NaN` → treat as **not interior**.
- "Strict interior" means margin > 0 with a safety buffer:
  - use a tolerance like `1e-12 * max(1, norm(s))` inside cone checks.
- Never allocate in cone kernels; pass scratch buffers.

11.3 Zero cone (equality constraints)

Primal cone: $\{0\}^k$ (no barrier, no interior).
Implementation: - Treat as a special block where `H=0` and no step-to-boundary restriction. - Do **not** include in ν. - Ensure KKT regularization makes the system solvable.

11.4 Nonnegative cone $\mathbb{R}^n_+$

Barrier:

$$f(s) = -\sum_i \log(s_i)$$

Gradient: $(\nabla f)_i = -1/s_i$
Hessian apply: $(\nabla^2 f(s)v)_i = v_i/s_i^2$
Step-to-boundary:

$$\alpha_{\max} = \min_{i:\Delta s_i<0} -s_i/\Delta s_i.$$

NT scaling: - $H = \text{diag}(s_i/z_i)$ - $H^{-1} = \text{diag}(z_i/s_i)$

11.5 SOC cone

SOC of dimension d: (t, x) with $t \geq \|x\|$.

11.5.1 Barrier

$$f(t, x) = -\log(t^2 - \|x\|^2)$$

Let $u = t^2 - \|x\|^2$.
Gradient:

$$\partial_t f = -\frac{2t}{u}, \qquad \partial_x f = \frac{2x}{u}.$$

Hessian apply:

$$\nabla^2 f = \frac{2}{u}\begin{bmatrix} -1 & 0 \\ 0 & I \end{bmatrix} + \frac{4}{u^2}\begin{bmatrix} t \\ -x \end{bmatrix}\begin{bmatrix} t \\ -x \end{bmatrix}^\top$$

So for $v = (v_t, v_x)$: - $a = tv_t - x^\top v_x$ - $\text{out}_t = (-2/u)v_t + (4/u^2)ta$ - $\text{out}_x = (2/u)v_x + (4/u^2)(-x)a$

11.5.2 Step-to-boundary (primal/dual)

Find max α such that:

$$(t + \alpha\Delta t)^2 - \|x + \alpha\Delta x\|^2 > 0, \quad t + \alpha\Delta t > 0.$$

Define: - $a = \Delta t^2 - \|\Delta x\|^2$ - $b = 2(t\Delta t - x^\top \Delta x)$ - $c = t^2 - \|x\|^2 > 0$
Solve $a\alpha^2 + b\alpha + c = 0$.
Take the smallest positive root as the boundary; also enforce $t + \alpha\Delta t > 0$.

11.5.3 NT scaling for SOC (implementation guidance) Implement SOC Jordan algebra operations:

- Jordan product:
$$(t, x) \circ (u, v) = (tu + x^\top v,\ tv + ux)$$
- Identity $e = (1, 0)$
- Determinant $\det(t, x) = t^2 - \|x\|^2$
- Spectral values (eigenvalues):
$$\lambda_1 = t + \|x\|, \quad \lambda_2 = t - \|x\|$$

  Interior iff $\lambda_2 > 0$.

Implement (closed form, no iteration): - Jordan square root $\sqrt{u}$ via $\sqrt{\lambda_1}$, $\sqrt{\lambda_2}$ - Jordan inverse u^{-1} via λ_1^{-1}, λ_2^{-1} - Quadratic representation apply:
$$P(w)y = 2w \circ (w \circ y) - (w \circ w) \circ y$$

Concrete NT scaling procedure (symmetric cones):
Given interior s, z in the SOC block:

1) Compute $s^{1/2}$.

2) Compute $u := P(s^{1/2})\, z$.

3) Compute $u^{-1/2}$.

4) Define the NT scaling point:
$$w := P(s^{1/2})\, u^{-1/2}.$$

For the SOC (and all symmetric cones), this w yields an NT scaling satisfying:

$$H(w)\, s = z, \qquad H(w)^{-1}\, z = s$$

for a suitable scaling map $H(w)$ built from $P(w)$.

Convention used in this design:
We build the KKT block H so that for the NonNeg cone it equals $\mathrm{diag}(s/z)$.
That corresponds to using $H = H(w)^{-1}$ (equivalently H is "the map that appears in the condensed KKT system", not the map that sends $s \mapsto z$).

In practice for SOC you should implement, per block: - `apply_H(v)` and `apply_Hinv(v)` using $P(w)$ (and/or its inverse) **without** materializing dense matrices when possible.

Performance note: for large SOC blocks, a dense `d×d` H can destroy sparsity and cost $O(d^2)$ per apply. Prefer a structured representation or a factored form if SOC dimensions commonly exceed ~50–100.

11.6 Rotated SOC (RSOC)

RSOC constraint: $2uv \geq \|w\|^2, u \geq 0, v \geq 0$

Reduce to SOC via:
$$t = u + v, \quad x = [u - v;\ \sqrt{2}\, w]$$

then $t \geq \|x\|$.

Implement as presolve so the kernel only needs SOC.

11.7 PSD cone S_+^n with svec storage

Represent symmetric X as `svec(X)` length $n(n+1)/2$: - diag entries unchanged - off-diagonals scaled by $\sqrt{2}$ so that $\langle X, Y \rangle = \mathrm{svec}(X)^\top \mathrm{svec}(Y)$.

11.7.1 Barrier and derivatives

$$f(X) = -\log\det(X), \qquad \nabla f(X) = -X^{-1}, \qquad \nabla^2 f(X)[V] = X^{-1}VX^{-1}.$$

Implementation (v1, dense): - convert svec → dense symmetric X - compute Cholesky $X = LL^\top$ - use triangular solves to apply X^{-1} and $X^{-1}(\cdot)X^{-1}$

11.7.2 Step-to-boundary Max α such that $X + \alpha\Delta X \succ 0$.
Let $M = X^{-1/2}\Delta X X^{-1/2}$, symmetric. Then: - if $\lambda_{\min}(M) \geq 0$: $\alpha_{\max} = +\infty$ - else $\alpha_{\max} = -1/\lambda_{\min}(M)$
Compute $\lambda_{\min}$ with dense eigensolver (v1) or Lanczos (v2).

11.7.3 NT scaling for PSD Compute:

$$W = X^{1/2}\left(X^{1/2}ZX^{1/2}\right)^{-1/2}X^{1/2}.$$

Then H is the linear map:

$$H[V] = WVW.$$

Implementation: - compute $X^{1/2}$ via eigendecomposition or Cholesky-based sqrt - compute $M = X^{1/2}ZX^{1/2}$, then $M^{-1/2}$ - build W - apply $V \mapsto WVW$ in matrix space, map back to svec

---

12. Nonsymmetric cones: EXP and POW (full details)

12.1 Exponential cone (EXP)

12.1.1 Definition (primal) Use CVXPY/Clarabel ordering:

$$K_{\text{exp}} = \text{cl}\{(x, y, z) : y > 0,\ ye^{x/y} \leq z\}.$$

12.1.2 Dual cone (useful for interior checks) A convenient closed form:

$$K^*_{\text{exp}} = \text{cl}\{(u, v, w) : u < 0,\ w \geq -u\exp(v/u - 1)\}.$$

(Here $w > 0$ is implied by the inequality when $u < 0$.)

12.1.3 Numerically stable primal interior test Given $s = (x, y, z)$, define:

$$\psi(s) := y\log(z/y) - x.$$

Then $s \in \text{int}(K_{\text{exp}})$ iff: - $y > 0$ - $z > 0$ - $\psi(s) > 0$
This avoids cancellation from $z - ye^{x/y}$.

12.1.4 Barrier (primal) and derivatives Use the standard 3-self-concordant log-homogeneous barrier:

$$f_{\text{exp}}(x, y, z) = -\log(\psi(x, y, z)) - \log(y) - \log(z), \qquad \psi = y\log(z/y) - x.$$

Let $\psi = y\log(z/y) - x$.
Then:

$$\nabla\psi = (-1,\ \log(z/y) - 1,\ y/z)$$

$$H_\psi = \nabla^2\psi = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1/y & 1/z \\ 0 & 1/z & -y/z^2 \end{bmatrix}.$$

Gradient:

$$\nabla f_{\text{exp}}(s) = -\frac{1}{\psi}\nabla\psi + (0,\ -1/y,\ -1/z).$$

Hessian:

$$\nabla^2 f_{\text{exp}}(s) = \frac{1}{\psi^2}\nabla\psi\,\nabla\psi^\top - \frac{1}{\psi}H_\psi + \text{diag}(0,\ 1/y^2,\ 1/z^2).$$

All are 3×3 and cheap.

12.1.5 Dual-map oracle via tiny Newton solve (needed for scaling and η) For nonsymmetric scaling you need $\nabla f^*(z)$ and $\nabla^2 f^*(z)$. Use Legendre relations:

For $z \in \text{int}(K^*_{\text{exp}})$, define:

$$x_z = \arg\min_{x\in\text{int}(K_{\text{exp}})} \phi(x) := z^\top x + f_{\text{exp}}(x).$$

Then:

$$\nabla f^*_{\text{exp}}(z) = -x_z, \qquad \nabla^2 f^*_{\text{exp}}(z) = \left[\nabla^2 f_{\text{exp}}(x_z)\right]^{-1}.$$

Compute x_z by Newton on:

$$r(x) := z + \nabla f_{\text{exp}}(x) = 0.$$

Newton step: - Solve $\nabla^2 f_{\text{exp}}(x)\Delta x = -r(x)$. - Backtrack line search to keep $x + \alpha\Delta x \in \text{int}(K_{\text{exp}})$. - Stop when $\|r(x)\|_\infty \le \varepsilon_{\text{dualmap}}$ (e.g. 1e-10).

Warm start: reuse the previous iteration's x_z.

12.2 3D Power cone (POW)

12.2.1 Definition (primal + dual) Primal:

$$K^\alpha_{\text{pow}} = \{(x, y, z) : x \ge 0,\ y \ge 0,\ x^\alpha y^{1-\alpha} \ge |z|\}, \quad \alpha \in (0, 1).$$

Dual:

$$(K^\alpha_{\text{pow}})^* = \{(u, v, w) : u \ge 0,\ v \ge 0,\ (u/\alpha)^\alpha (v/(1-\alpha))^{1-\alpha} \ge |w|\}.$$

12.2.2 Interior test Define:

$$\psi(x, y, z) = x^{2\alpha} y^{2(1-\alpha)} - z^2.$$

Then interior iff $x > 0$, $y > 0$, $\psi > 0$.

(Compute $x^{2\alpha}y^{2(1-\alpha)}$ via logs for stability.)

12.2.3 Barrier and derivatives (improved 3-self-concordant) Use:

$$f_{\text{pow}}(x, y, z) = -\log(\psi(x, y, z)) - (1-\alpha)\log(x) - \alpha\log(y), \quad \psi = x^{2\alpha}y^{2(1-\alpha)} - z^2.$$

Let: - $a = 2\alpha$, $b = 2(1-\alpha) = 2 - a$ - $p = x^a y^b$, $\psi = p - z^2$

Gradient of ψ:

$$\nabla\psi = \left(\frac{ap}{x},\ \frac{bp}{y},\ -2z\right).$$

Hessian of p:

$$\nabla^2 p = \begin{bmatrix} \frac{a(a-1)p}{x^2} & \frac{abp}{xy} & 0 \\ \frac{abp}{xy} & \frac{b(b-1)p}{y^2} & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

So:

$$\nabla^2\psi = \nabla^2 p + \text{diag}(0, 0, -2).$$

Then:

$$\nabla f_{\text{pow}}(s) = -\frac{1}{\psi}\nabla\psi + \left(-\frac{1-\alpha}{x},\ -\frac{\alpha}{y},\ 0\right),$$

$$\nabla^2 f_{\text{pow}}(s) = \frac{1}{\psi^2}\nabla\psi\,\nabla\psi^\top - \frac{1}{\psi}\nabla^2\psi + \text{diag}\left(\frac{1-\alpha}{x^2},\ \frac{\alpha}{y^2},\ 0\right).$$

12.2.4 Dual-map oracle (same pattern as EXP) Compute x_z solving:

$$z + \nabla f_{\text{pow}}(x) = 0$$

with Newton + backtracking in the primal interior.

Then:

$$\nabla f^*_{\text{pow}}(z) = -x_z, \qquad \nabla^2 f^*_{\text{pow}}(z) = \left[\nabla^2 f_{\text{pow}}(x_z)\right]^{-1}.$$

---

13. Scaling objects and KKT H-block representation

13.1 ScalingBlock representation

```
pub enum ScalingBlock {
    Zero { dim: usize },

    Diagonal { d: Vec<f64> },         // H = diag(d)
    Dense { h: Vec<f64>, n: usize },  // H dense n×n

    Soc { /* structured representation */ },
    Psd { /* store W or a factorization */ },

    Nonsym3D { h: [f64; 9] },         // H 3×3 row-major
}

impl ScalingBlock {
    pub fn apply(&self, v: &[f64], out: &mut [f64]);
    pub fn apply_inv(&self, v: &[f64], out: &mut [f64]);
}
```

13.2 Bounds as cones (continuous + MIP)

Support bounds $l \le x \le u$ by adding linear inequality rows + NonNeg slack:

- $x_i \le u$: add row $e_i^\top x + s = u,\ s \ge 0$
- $x_i \ge l$: add row $-e_i^\top x + s = -l,\ s \ge 0$

Keep everything in one conic canonical form.

---

14. Nonsymmetric scaling (BFGS) — equation-complete spec (EXP/POW)

For EXP and POW blocks we implement a primal-dual scaling satisfying two secant equations.

14.1 Required oracles per 3D block

You must have: - `grad_primal(s) = ∇f(s)` and `hess_primal(s) = ∇²f(s)` (closed form, §12) - `dual_map(z)` returning: - $\tilde{s} = -\nabla f^*(z)$ (primal point) - $H^* = \nabla^2 f^*(z)$ (3x3 SPD), computed as $(\nabla^2 f(\tilde{s}))^{-1}$

Compute `dual_map(z)` via the tiny inner Newton solve (§12.1.5 / §12.2.4).

14.2 Shadow points

Given current $(s, z) \in \text{int}(K) \times \text{int}(K^*)$:

- $\tilde{z} := -\nabla f(s) \in \text{int}(K^*)$
- $\tilde{s} := -\nabla f^*(z) \in \text{int}(K)$

Define: - $Z := [z, \tilde{z}] \in \mathbb{R}^{3\times 2}$ - $S := [s, \tilde{s}] \in \mathbb{R}^{3\times 2}$

14.3 BFGS scaling matrix H (3×3 SPD)

We want H to satisfy:

$$Hz = s, \qquad H\tilde{z} = \tilde{s}.$$

Choose an "anchor" SPD matrix H_a. Strong default:

$$H_a := \mu\,\nabla^2 f^*(z) = \mu\,H^*.$$

Then compute:

$$H = Z(Z^\top S)^{-1}Z^\top + H_a - H_a S(S^\top H_a S)^{-1}S^\top H_a.$$

Implementation notes: - $Z^\top S$ and $S^\top H_a S$ are 2×2 → invert explicitly. - If either 2×2 matrix is near singular: - damp $H_a \leftarrow H_a + \delta I$ - or skip low-rank update and set $H \leftarrow H_a$ - Symmetrize: $H \leftarrow \frac{1}{2}(H + H^\top)$ - SPD check via Cholesky; fallback to $H_a + \delta I$ if needed

Store H in `ScalingBlock::Nonsym3D`.

14.4 Unit initialization (critical for robustness)

For any nonsymmetric cone blocks, initialize each 3D block at a well-centered point:

- EXP block:
$$s_0 = z_0 \approx (-1.051383,\ 0.556409,\ 1.258967)$$
- POW block:
$$s_0 = z_0 = (\sqrt{1+\alpha},\ \sqrt{2-\alpha},\ 0).$$

Use these when creating the initial interior iterate.

---

15. 3rd-order correction for nonsymmetric cones (EXP/POW) — implementable

15.1 Correction term

For each nonsymmetric 3D block, use:

$$\eta = -\frac{1}{2}\nabla^3 f^*(z)[\Delta z,\ (\nabla^2 f^*(z))^{-1}\Delta s].$$

This is used in the combined-step definition of d_s (§7.3.2).

15.2 Compute η using only primal derivatives (practical)

Even when f^* is not closed form, use Legendre relations.

Let: - $x := -\nabla f^*(z)$ (from the dual-map Newton solve) - $H_x := \nabla^2 f(x)$ - $H^* := \nabla^2 f^*(z) = H_x^{-1}$

Then compute:

1. $p := -H^*\,\Delta z$
2. $u := \nabla^3 f(x)[p, \Delta s]$ (a 3-vector; contraction)
3. $\eta := \frac{1}{2}\,H^*\,u$

So you need routines to compute $u = \nabla^3 f(x)[p, q]$ for EXP and POW.

15.3 Generic contraction formula for -log(ψ)

Both EXP and POW barriers have the form:

$$f(s) = -\log(\psi(s)) + h(s)$$

with scalar ψ and simple h.

Let (evaluated at $\bar{x}$): - $s = \psi(\bar{x})$ (scalar) - $g = \nabla\psi(\bar{x}) \in \mathbb{R}^3$ - $H = \nabla^2\psi(\bar{x}) \in \mathbb{R}^{3\times 3}$ - $T[p,q] = \nabla^3\psi(\bar{x})[p,q] \in \mathbb{R}^3$

Define scalars: - $a = g^\top p$ - $b = g^\top q$ - $c = p^\top Hq$

Then for $g_0 = -\log(\psi)$, the contraction

$$u_0 := \nabla^3 g_0(\bar{x})[p,q]$$

can be computed as:

$$u_0 = \frac{b}{s^2}(Hp) + \frac{c}{s^2}g - \frac{2ab}{s^3}g + \frac{a}{s^2}(Hq) - \frac{1}{s}T[p,q].$$

Finally:

$$u = u_0 + \nabla^3 h(\bar{x})[p,q].$$

This avoids building a full 3×3×3 tensor.

15.4 Specialization: EXP

For EXP:

$$\psi = y\log(z/y) - x.$$

We already have $g = \nabla\psi$ and $H = \nabla^2\psi$ in §12.1.4.

Third-derivative contraction $T[p,q]$ (all x-derivatives vanish):

- $T_x = 0$
- $T_y = \frac{1}{y^2}p_y q_y - \frac{1}{z^2}p_z q_z$
- $T_z = -\frac{1}{z^2}(p_z q_y + p_y q_z) + \frac{2y}{z^3}p_z q_z$

For $h = -\log y - \log z$: - $u_x^{(h)} = 0$ - $u_y^{(h)} = -\frac{2}{y^3}p_y q_y$ - $u_z^{(h)} = -\frac{2}{z^3}p_z q_z$

Then compute $u = u_0 + u^{(h)}$ and $\eta = \frac{1}{2}H^* u$.

15.5 Specialization: POW

For POW:

$$\psi = x^a y^b - z^2, \qquad a = 2\alpha,\ b = 2(1-\alpha).$$

Let $P = x^a y^b$.

Third derivatives come only from P (since $-z^2$ has zero 3rd derivative).

Precompute:

$$P_{xxx} = a(a-1)(a-2)\frac{P}{x^3}, \quad P_{xxy} = ab(a-1)\frac{P}{x^2 y}, \quad P_{xyy} = ab(b-1)\frac{P}{xy^2},$$

$$P_{yyy} = b(b-1)(b-2)\frac{P}{y^3}, \quad P_{yxx} = P_{xxy}, \quad P_{yxy} = P_{xyy}.$$

Then for direction vectors p, q:

- $T_z = 0$
- $T_x = P_{xxx}p_x q_x + P_{xxy}(p_x q_y + p_y q_x) + P_{xyy}p_y q_y$
- $T_y = P_{yxx}p_x q_x + P_{yxy}(p_x q_y + p_y q_x) + P_{yyy}p_y q_y$

For $h = -(1-\alpha)\log x - \alpha\log y$: - $u_x^{(h)} = -\frac{2(1-\alpha)}{x^3}p_x q_x$ - $u_y^{(h)} = -\frac{2\alpha}{y^3}p_y q_y$ - $u_z^{(h)} = 0$

Then compute $u = u_0 + u^{(h)}$ and $\eta = \frac{1}{2}H^* u$.

15.6 Engineering recommendations

- Unit test these 3D kernels heavily (finite differences + random interior points).
- If analytic 3rd-derivative code is too risky initially, implement a debug-only finite-difference fallback for u, but don't ship it as default.

---

16. Termination criteria and statuses

All termination checks should be done on **unscaled** data (after undoing Ruiz scaling).

Let:

$$\bar{x} = x/\tau, \quad \bar{s} = s/\tau, \quad \bar{z} = z/\tau.$$

Residuals:

$$r_p = A\bar{x} + \bar{s} - b, \qquad r_d = P\bar{x} + A^\top \bar{z} + q.$$

Objectives:

$$g_p = \frac{1}{2}\bar{x}^\top P\bar{x} + q^\top \bar{x}, \qquad g_d = -\frac{1}{2}\bar{x}^\top P\bar{x} - b^\top \bar{z}.$$

16.1 Optimality

Declare `Optimal` if: - $\|r_p\|_\infty \le \varepsilon_f \cdot \max(1, \|b\|_\infty + \|\bar{x}\| + \|\bar{s}\|)$ - $\|r_d\|_\infty \le \varepsilon_f \cdot \max(1, \|q\|_\infty + \|\bar{x}\| + \|\bar{z}\|)$ - $|g_p - g_d| \le \varepsilon_f \cdot \max(1, \min(|g_p|, |g_d|))$

Map ε_f to `tol_feas` / `tol_gap`.

16.2 Infeasibility certificates (τ → 0 regime)

When τ becomes small, use unnormalized variables to test certificates.

Primal infeasibility (typical pattern): - $b^\top z < -\varepsilon_{i,a}$ - $\|A^\top z\| \le \varepsilon_{i,r} \cdot \max(1, \|x\| + \|z\|) \cdot |b^\top z|$ - $z \in K^*$ (or approximately)

Dual infeasibility (typical pattern): - $q^\top x < -\varepsilon_{i,a}$ - $\|Px\| \le \varepsilon_{i,r} \cdot \max(1, \|x\|) \cdot |q^\top x|$ - $\|Ax + s\| \le \varepsilon_{i,r} \cdot \max(1, \|x\| + \|s\|) \cdot |q^\top x|$ - $s \in K$ (or approximately)

16.3 NumericalError

Return `NumericalError` if: - factorization fails even after increasing regularization - NaNs appear - step size stalls below a minimum threshold for many iterations

Always include diagnostics: - last residuals, μ, τ, κ, regularization counts

---

17. Mixed-integer design (Branch-and-Bound)

17.1 Reality check (important)

A pure IPM-based continuous relaxation inside B&B will be **much** weaker than commercial MILP engines for MILP-heavy workloads.
v1 goal is "basic discrete support," not "Gurobi replacement for MILP."

17.2 Node structure

```
pub struct BnbNode {
    pub bounds: Vec<VarBound>,
    pub depth: usize,
    pub lower_bound: f64,             // relaxation objective
    pub warm_start: Option<WarmStart>, // x,s,z,τ,κ
}
```

17.3 Tree management

- Node selection: best-bound priority queue
- Branching: most fractional variable (or strong branching later)
- Pruning:
  - infeasible node
  - bound ≥ incumbent - mip_tol
  - time / node / depth limit

17.4 Heuristics

- Rounding + repair
- Diving
- Feasibility pump (v2)
- Local branching (v3)

17.5 Warm starts

Warm start child nodes using parent's solution: - apply bound changes to x (projection) - recompute s from $Ax+s = b\tau$ - shift into cone interior if needed

17.6 OA / cuts roadmap (v2+)

For MISOCP: - add SOC tangent cuts at violated points
For EXP/POW: - add supporting hyperplanes (epigraph linearizations)
A serious mixed-integer product will need: - cut pools - presolve - strong branching - primal heuristics - separation

---

18. C ABI design (stable)

18.1 C structs

```
typedef struct solver_handle solver_handle;

typedef struct {
  int max_iter;
  double tol_feas;
  double tol_gap;
  double tol_infeas;
  double static_reg;
  double dynamic_reg_min_pivot;
  int ruiz_iters;
  int verbose;
  int threads;
  unsigned long long seed;
} solver_settings;
```

18.2 Functions

```
solver_handle* solver_create(const solver_settings* s);

int solver_load_problem(
    solver_handle* h,
    // sparse CSC for P, A
    // arrays for q, b
    // cone specs
    // integrality, bounds
);

int solver_solve(solver_handle* h);

int solver_get_solution(
    solver_handle* h,
    double* x_out,
    double* s_out,
    double* z_out
);

void solver_destroy(solver_handle* h);
```

---

19. Python bindings and CVXPY integration

19.1 Python API

Expose: - `solve(P, q, A, b, cones, settings) -> (x, s, z, info)`
Inputs: - `scipy.sparse.csc_matrix` for sparse matrices - conic dims in a CVXPY-compatible dict format

19.2 CVXPY plugin

Implement the standard CVXPY solver interface: - `apply(data)` — map canonicalized problem to solver inputs - `solve_via_data(data, warm_start, verbose, solver_opts)` - `invert(solution)` — map back to CVXPY primal/dual format
Focus on correctness; performance comes from `solver-core`.

---

20. Benchmarking and test harness

20.1 Benchmark suites

Continuous: - NETLIB LP - Maros–Mészáros QP - QPLIB (subset) - Mittelmann SOCP suites - CBLIB (conic benchmark library) - SDPLIB (for PSD/SDP)
Mixed-integer: - MIPLIB (MILP/MIQP) - (Subset) mixed-integer conic instances (where available)

20.2 Formats

- MPS / QPS
- CBF (CBLIB)
- SDPA/SDPLIB

20.3 Runner CLI

Provide `solver-bench`:

```
solver-bench run \
  --suite netlib \
  --time-limit-ms 60000 \
  --out results/netlib.jsonl
```

```
solver-bench compare \
  --baseline ecos \
  --candidate ours \
  --metric time \
  --out report/netlib_time_profile.html
```

Each JSONL record includes: - instance id - status - solve time + breakdown - iterations - residuals / gap - objective

20.4 Regression testing

- Curate `smoke/` set (~50 instances across cones)
- CI: correctness + generous runtime ceilings
- Nightly: full suites + performance tracking

---

21. Testing strategy

21.1 Cone kernel unit tests (mandatory)

For each cone: - interior check correctness - step-to-boundary correctness - barrier grad via finite differences - Hessian-vector products via finite differences

21.2 3D cone finite difference templates

- Gradient:
$$\frac{f(x+\epsilon e_i)-f(x-\epsilon e_i)}{2\epsilon}$$
- Hessian-vector:
$$\frac{\nabla f(x+\epsilon v)-\nabla f(x-\epsilon v)}{2\epsilon}\approx \nabla^2 f(x)v$$

Pick $\epsilon \approx 10^{-6}\cdot \max(1,\|x\|)$.

21.3 Dual-map oracle tests (EXP/POW)

For random interior z: - compute x_z - verify $\|z+\nabla f(x_z)\|$ is tiny - verify $H^* \approx (\nabla^2 f(x_z))^{-1}$

21.4 KKT solve tests

- small random QPs with known solution
- compare against reference solvers (in Python tests)

21.5 Parser fuzzing

- fuzz MPS/CBF parsers
- reject invalid input without panics

---

22. Performance engineering checklist

22.1 Memory

- Pre-allocate all iteration work vectors
- Reuse KKT sparsity pattern and symbolic factorization
- Avoid cloning; use in-place ops

22.2 Parallelism

- Parallelize cone blocks (SOC/EXP/POW)
- Parallelize SpMV and residual computations
- Factorization parallelism depends on backend

22.3 Numerics

- stable quadratic root formulas (SOC)
- `exp`/`log` guards (EXP/POW)
- PSD eigen computations with robust fallback
- iterative refinement (especially GPU)

22.4 Instrumentation

Log per-iteration: - residual norms - μ, τ, κ - step size α - factorization time / solve time - regularization bumps

---

23. GPU backend roadmap (architecture hooks)

23.1 Backend requirements

GPU backend must implement: - sparse factorization/solve (vendor or custom) - SpMV - batched dense kernels for 3D cones - mixed precision + refinement

23.2 Memory layout

Group s/z by cone family: - contiguous SoA arrays for EXP/POW (x[], y[], z[]) - SOC blocks grouped by dimension for batched kernels

23.3 Mixed precision

- factorize in FP32/TF32
- refine in FP64
- termination checks in FP64

---

24. Implementation milestones

Milestone A: continuous v0.1 (SOC + LP/QP)

- Zero, NonNeg, SOC
- HSDE + NT scaling
- sparse LDL^T backend
- benchmark vs ECOS

Milestone B: nonsymmetric cones

- EXP + POW kernels
- dual-map oracles
- BFGS scaling + 3rd-order correction
- benchmark CBLIB / CVXPY workloads

Milestone C: MIP v0.1

- B&B with bounds-as-constraints
- warm starts + basic heuristics
- benchmark MIPLIB subset

Milestone D: PSD + chordal

- dense PSD
- chordal decomposition presolve
- benchmark SDPLIB

Milestone E: GPU prototype

- backend trait implemented
- SOC/NonNeg + KKT solve on GPU
- mixed precision refinement

---

25. References / reading list (practical)

- Clarabel (interior-point method for conic QPs; HSDE; nonsymmetric scaling)
- GPU acceleration work inspired by CuClarabel-style mixed parallel strategies
- MOSEK documentation/papers for exponential and power cone barriers and numerics
- CVXPY solver interface docs
- ECOS / ECOS_BB as baseline conic IPM + B&B

---

26. "Hello world" sketches

26.1 Rust

```
let prob = ProblemData { /* fill P,q,A,b,cones */ };
let settings = SolverSettings { max_iter: 200, tol_feas: 1e-8, tol_gap: 1e-8, ..Default::default() };
let result = solver_core::solve(&prob, &settings)?;
println!("{:?} obj={}", result.status, result.obj_val);
```

26.2 C

```
solver_settings s = {0};
s.max_iter = 200;
s.tol_feas = 1e-8;
s.tol_gap  = 1e-8;
s.ruiz_iters = 10;
s.verbose = 1;

solver_handle* h = solver_create(&s);
solver_load_problem(h, /* ... */);
solver_solve(h);
solver_get_solution(h, x, svec, z);
solver_destroy(h);
```

26.3 Python + CVXPY

```
import cvxpy as cp
import numpy as np

x = cp.Variable(100)
P = np.eye(100)
prob = cp.Problem(cp.Minimize(0.5*cp.quad_form(x, P)), [x >= 0, cp.sum(x) == 1])
prob.solve(solver="SOLVERNAME")
print(prob.value)
```