Discrete and Continuous Methods for Packing Problems

Your Name

January 22, 2026

Contents

Preface

This book develops a unified framework for packing problems, combining discrete optimization, graph theory, and continuous methods from applied mathematics.

Chapter 1

Simulated Annealing for Circle Packing in a Square

Guiding Question

How can a stochastic, temperature–driven algorithm reliably solve deterministic, highly non–convex geometric optimization problems where gradient–based methods fail?

This chapter establishes the *theoretical and conceptual foundation* for simulated annealing (SA) through an inquiry–based lens. We use the geometric problem of packing N identical circles into a square of minimal side length as a concrete running example. The goal is not to implement SA yet, but to understand *why it works*, *what assumptions it relies on*, and *which design choices matter* before turning to test–driven C implementations in later chapters.

1.1 The Optimization Problem

We consider the problem of packing N circles of radius r into a square of side length L, minimizing L subject to non–overlap and boundary constraints.

1.1.1 Configuration Space

A configuration is described by

$$x = (x_1, y_1, \ldots, x_N, y_N) \in \mathbb{R}^{2N},$$

with constraints

$$|(x_i, y_i) - (x_j, y_j)| \geq 2r, \quad i \neq j, \qquad r \leq x_i, y_i \leq L - r.$$

The feasible set is a subset of $\mathbb{R}^{2N}$ defined by pairwise distance constraints and box constraints.

1.2 Why Classical Optimization Fails

1.2.1 Inquiry

- Is the feasible set convex?
- Is the objective function differentiable everywhere?
- How does the number of local minima scale with N?

Each question reveals a structural obstruction:

- The feasible set is *highly non–convex*.
- Contact events between circles introduce *nonsmoothness*.
- The number of metastable configurations grows combinatorially with N.

These are not numerical pathologies but geometric ones. Gradient–based methods fail *by design*, not merely by poor tuning.

1.3 From Optimization to Sampling

The key conceptual shift of simulated annealing is to replace deterministic descent with *probabilistic exploration*.

1.3.1 Energy Formulation

We introduce a scalar energy

$$E(x, L) = E_{\text{overlap}}(x) + \alpha L,$$

where E_{overlap} penalizes violations of geometric constraints and $\alpha > 0$ balances feasibility against compactness.

1.3.2 Key Question

Why should we ever accept a move that increases E?

The answer is central: rejecting all uphill moves traps the algorithm in local minima. Allowing energy–increasing moves enables escape from metastable states.

1.4 Statistical Mechanics Perspective

Chapter 2

Simulated Annealing for Circle Packing in a Square

Guiding Question

How can a stochastic, temperature–driven algorithm reliably solve deterministic, highly non–convex geometric optimization problems where gradient–based methods fail?

This chapter establishes the *theoretical and conceptual foundation* for simulated annealing (SA) through an inquiry–based lens. We use the geometric problem of packing N identical circles into a square of minimal side length as a concrete running example. The goal is not to implement SA yet, but to understand *why it works*, *what assumptions it relies on*, and *which design choices matter* before turning to test–driven C implementations in later chapters.

2.1 The Optimization Problem

We consider the problem of packing N circles of radius r into a square of side length L, minimizing L subject to non–overlap and boundary constraints.

2.1.1 Configuration Space

A configuration is described by

$$x = (x_1, y_1, \ldots, x_N, y_N) \in \mathbb{R}^{2N},$$

with constraints

$$|(x_i, y_i) - (x_j, y_j)| \geq 2r, \quad i \neq j, \qquad r \leq x_i, y_i \leq L - r.$$

The feasible set is a subset of $\mathbb{R}^{2N}$ defined by pairwise distance constraints and box constraints.

2.2 Why Classical Optimization Fails

2.2.1 Inquiry

- Is the feasible set convex?
- Is the objective function differentiable everywhere?
- How does the number of local minima scale with N?

Each question reveals a structural obstruction:

- The feasible set is *highly non–convex*.
- Contact events between circles introduce *nonsmoothness*.
- The number of metastable configurations grows combinatorially with N.

These are not numerical pathologies but geometric ones. Gradient–based methods fail *by design*, not merely by poor tuning.

2.3 From Optimization to Sampling

The key conceptual shift of simulated annealing is to replace deterministic descent with *probabilistic exploration*.

2.3.1 Energy Formulation

We introduce a scalar energy

$$E(x, L) = E_{\text{overlap}}(x) + \alpha L,$$

where E_{overlap} penalizes violations of geometric constraints and $\alpha > 0$ balances feasibility against compactness.

2.3.2 Key Question

Why should we ever accept a move that increases E?

The answer is central: rejecting all uphill moves traps the algorithm in local minima. Allowing energy–increasing moves enables escape from metastable states.

2.4 Statistical Mechanics Perspective

Simulated annealing is grounded in equilibrium statistical mechanics.

2.4.1 Gibbs Measure

For temperature $T > 0$, define

$$\pi_T(x) = Z_T^{-1} e^{-E(x)/T},$$

where Z_T is the normalizing constant.

- High T: broad exploration of configuration space.
- Low T: concentration near global minima.

Optimization emerges as the zero–temperature limit of sampling.

2.4.2 Metropolis Acceptance Rule

Given a current state x and a proposal x', accept x' with probability

$$p = \min\bigl(1, e^{-(E(x')-E(x))/T}\bigr).$$

2.4.3 Inquiry

- Why does this rule preserve π_T as an invariant distribution?
- What role does detailed balance play?

These questions will later justify the correctness of the algorithm independent of implementation.

2.5 Simulated Annealing as a Limit Process

Simulated annealing proceeds by lowering the temperature:

$$T_0 > T_1 > \cdots > T_k \to 0.$$

2.5.1 Theoretical Guarantee

With logarithmic cooling, $T_k \sim c/\log k$, simulated annealing converges almost surely to a global minimizer. Although impractical, this result explains *why* the method can work at all.

2.6 Algorithmic Ingredients

Every simulated annealing algorithm consists of:

1. **State space**: configurations x (and possibly L).

2. **Energy**: objective plus penalties.
3. **Proposal kernel**: how candidate states are generated.
4. **Acceptance rule**: Metropolis criterion.
5. **Cooling schedule**: temperature decay.

Failure of any component compromises the method.

2.7 Energy Design for Circle Packing

Hard constraints destroy ergodicity. Instead we use soft penalties.

2.7.1 Pairwise Overlap Penalty

For distance d between circle centers,

$$\phi(d) = \begin{cases} (2r - d)^2, & d < 2r, \\ 0, & d \geq 2r. \end{cases}$$

Total overlap energy:

$$E_{\text{overlap}}(x) = \sum_{i<j} \phi(|x_i - x_j|) + \sum_i \phi_{\text{wall}}(x_i).$$

2.7.2 Inquiry

- Why quadratic penalties instead of infinite barriers?
- How does penalty scaling interact with temperature?

2.8 Proposal Mechanisms

Common proposal strategies include:

- local Gaussian perturbations of a single circle,
- occasional global reshuffling,
- temperature–dependent step sizes.

A key heuristic is

$$\textbf{proposal scale} \propto \sqrt{T}.$$

2.9 Optimizing the Square Size

Two approaches are common:

2.9.1 Feasibility Search

Fix L, test feasibility via SA, and perform an outer binary search on L.

2.9.2 Joint Optimization

Optimize (x, L) jointly using

$$E(x, L) = E_{\text{overlap}}(x) + \alpha L.$$

2.9.3 Inquiry

- Why must α be temperature–aware?
- What failure modes arise if L shrinks too early?

2.10 Common Failure Modes

Symptom	Cause
Freezing	Cooling too fast
Jittering	Proposal scale too large
Poor minima	Bad initialization
Slow convergence	Poor energy scaling

2.11 Exercises

1. Prove detailed balance for the Metropolis acceptance rule.
2. Show that hard constraints break ergodicity.
3. Compare simulated annealing and basin hopping on small N.
4. Design a temperature–dependent penalty function.
5. Explain why SA only superficially resembles stochastic gradient descent.

2.12 Conceptual Summary

Simulated annealing replaces deterministic descent with controlled stochastic exploration. Its validity comes from statistical mechanics; its effectiveness comes from careful algorithmic design.

In later chapters, this conceptual foundation will be translated into *test–driven C implementations*, where correctness precedes performance.

Chapter 3

Test-Driven Simulated Annealing: From Theory to a Na"ive C Baseline

Purpose

This chapter translates the conceptual foundations of simulated annealing developed in Chapter 1 into a *correctness-first implementation workflow*. The emphasis is not speed or sophistication, but **trustworthiness**: every algorithmic component is introduced via inquiry and then validated through test-driven development (TDD).

By the end of the chapter, we obtain a minimal simulated annealing solver for 2D circle packing that is:

- reproducible,
- instrumented,
- mathematically faithful to the theory,
- and suitable as a baseline for later optimization (OpenMP, CUDA, etc.).

3.1 How This Chapter Fits the Narrative

Chapter 1 answered the question *why* simulated annealing is appropriate for nonconvex geometric optimization. This chapter answers a different question:

> *How do we implement simulated annealing so that each theoretical assumption is explicitly checked and enforced by tests?*

The organizing principle is:

$$extTheory \longrightarrow \text{Algorithmic Invariant} \longrightarrow \text{Unit Test} \longrightarrow \text{Code}.$$

3.2 Learning Outcomes

After completing this chapter, you should be able to:

- implement a minimal Metropolis kernel and annealing loop;
- encode geometric constraints using smooth penalty energies;
- write tests that enforce probabilistic and geometric invariants;
- diagnose failures using acceptance-rate and energy traces;
- extend the baseline toward feasibility search and joint (x, L) optimization.

3.3 Project Skeleton

We use a deliberately small and explicit structure. Each module corresponds to one theoretical component from Chapter 1.

```
sa_circlepack/
include/                  (C headers in later chapters)
src/
energy.py              # E(x,L)
propose.py             # proposal kernel
metropolis.py          # acceptance rule
anneal.py              # annealing loop
rng.py                 # deterministic randomness
tests/
test_energy.py
test_propose.py
test_metropolis.py
test_anneal.py
test_reproducibility.py
```

Although this chapter uses Python for rapid iteration, the structure mirrors the C implementations developed later.

3.4 Design Specification: Na"ive Baseline

3.4.1 State

A state consists of

$$(X, L), \qquad X \in \mathbb{R}^{N\times 2},\ L > 0,$$

where X stores circle centers and L is the square side length.

3.4.2 Energy

We implement exactly the energy introduced in Chapter 1:

$$E(X, L) = E_{\text{pair}}(X) + E_{\text{wall}}(X, L) + \alpha L.$$

Pairwise overlap penalty:

$$E_{\text{pair}}(X) = \sum_{i<j} \phi(|x_i - x_j|), \qquad \phi(d) = \max(0, 2r - d)^2.$$

Wall penalty:

$$E_{\text{wall}}(X, L) = \sum_{i} \sum_{k=1}^{2} \Big[\max(0, r - x_{i,k})^2 + \max(0, x_{i,k} - (L - r))^2\Big].$$

3.4.3 Inquiry

- Why must $E \geq 0$ always?
- Why must feasible configurations yield $E_{\text{pair}} = E_{\text{wall}} = 0$?

These questions become explicit unit tests.

3.4.4 Proposal Kernel

A proposal perturbs exactly one circle:

$$x'_k = x_k + \delta, \qquad \delta \sim \mathcal{N}(0, \sigma^2 I_2).$$

This choice enforces locality and ergodicity. In the naive baseline, proposals may be clipped to remain within a bounding box to avoid numerical blow-up.

3.4.5 Acceptance Rule

Given $\Delta E = E' - E$ at temperature T, we apply the Metropolis criterion:

$$\mathbb{P}(\text{accept}) = \min(1, e^{-\Delta E/T}).$$

3.4.6 Cooling Schedule

We use geometric cooling:

$$T_{k+1} = \gamma T_k, \qquad \gamma \in (0, 1).$$

This is not theoretically optimal, but it is sufficient for a correctness baseline.

3.5 Inquiry-Driven TDD Sequence

Each exercise introduces a single invariant implied by the theory and enforces it via tests.

3.5.1 Exercise 1: Deterministic Randomness

Invariant: identical seeds imply identical trajectories.

Test:

- Repeated calls with the same seed produce identical proposals and acceptance decisions.

3.5.2 Exercise 2: Energy Sanity

Invariants:

1. $E(X, L) \geq 0$ for all X.
2. Well-separated interior configurations yield zero penalty.

3.5.3 Exercise 3: Symmetry

Invariant: permuting circle indices leaves E_{pair} unchanged.

3.5.4 Exercise 4: Proposal Locality

Invariants:

1. Exactly one circle moves per proposal.
2. Empirical variance of steps scales with σ^2.

3.5.5 Exercise 5: Metropolis Correctness

Invariants:

1. Downhill moves are always accepted.
2. Uphill acceptance frequencies match $e^{-\Delta E/T}$.

3.5.6 Exercise 6: Best-So-Far Monotonicity

Invariant: the best recorded energy is nonincreasing over time.

3.5.7 Exercise 7: Temperature Effects

Observation: higher initial temperature improves escape from overlaps on small N.

3.6 Diagnostics and Instrumentation

Acceptance rate is a primary diagnostic:

- early phase: 0.2–0.6,
- late phase: 0.01–0.2.

Deviations indicate mismatches between proposal scale, temperature, and energy magnitude.

3.7 Minimal Interfaces

The following interfaces are intentionally narrow:

```
energy.energy(X, L, r, alpha) -> float
energy.pair_energy(X, r) -> float
energy.wall_energy(X, L, r) -> float

propose.propose_move(X, L, r, sigma, rng)
-> (X_new, moved_index)

metropolis.accept(delta_E, T, rng) -> bool

anneal.run(X0, L0, r, alpha, T0, gamma, n_steps, sigma, seed)
-> dict(trace, best_state, best_energy, accept_rate)
```

Each function corresponds to a single theoretical concept from Chapter 1.

3.8 Implementation Notes

- Keep functions pure; pass RNG objects explicitly.
- Prefer clarity over vectorization in the baseline.
- Log energy, acceptance, and temperature at every step.
- Add performance optimizations *only after* all tests pass.

3.9 Stretch Goals

1. Adaptive proposal scaling based on acceptance rate.
2. Two-stage annealing schedules.
3. Feasibility search via outer bisection in L.
4. Incremental energy updates for $\mathcal{O}(N)$ proposals.

3.10 Checkpoint

At this stage, you should have a solver where:

- all tests pass deterministically;
- the algorithm reflects the assumptions of Chapter 1;
- failures are explainable via diagnostics;
- the code is ready to be translated into C in subsequent chapters.

This completes the transition from *theory* to *validated implementation*. In Chapter 3, we repeat this process for gradient-based optimization, culminating in a fully test-driven C implementation of Adam.

Chapter 4

Inquiry-Based Implementation of Stochastic Gradient Descent with Adam

Guiding Question

How can noisy, partial gradient information be systematically transformed into a stable, scalable optimization method for high-dimensional, nonconvex problems?

This chapter develops stochastic gradient descent (SGD) and the Adam optimizer through an inquiry-based sequence. The emphasis is on understanding the mathematical role of noise, momentum, and adaptivity, and on implementing Adam correctly from first principles.

4.1 From Deterministic to Stochastic Gradients

Consider the finite-sum optimization problem

$$\min_{\theta \in \mathbb{R}^d} F(\theta) := \frac{1}{N} \sum_{i=1}^{N} f_i(\theta).$$

4.1.1 Inquiry

- What happens if N is very large?
- Is it necessary to evaluate all f_i at every iteration?
- What if we replace the full gradient with an estimator?

This leads to stochastic gradients. Given a random index i_k, define

$$g_k := \nabla f_{i_k}(\theta_k).$$

Then $\mathbb{E}[g_k] = \nabla F(\theta_k)$, but g_k has nonzero variance.

4.2 Stochastic Gradient Descent

The SGD iteration is

$$\theta_{k+1} = \theta_k - \eta_k g_k.$$

4.2.1 Inquiry

- Why does SGD converge despite noisy gradients?
- What role does the learning rate η_k play?
- Why must $\eta_k \to 0$ in theory but not always in practice?

4.2.2 Key Phenomena

- Gradient noise acts as implicit regularization.
- SGD escapes shallow local minima and saddle points.
- Variance limits the achievable accuracy.

4.3 Momentum as Time Averaging

To reduce variance and accelerate convergence, introduce momentum:

$$v_{k+1} = \beta v_k + (1-\beta) g_k,$$
$$heta_{k+1} = \theta_k - \eta v_{k+1}.$$

4.3.1 Inquiry

- Why does averaging gradients reduce noise?
- How is momentum related to low-pass filtering?
- Why can momentum overshoot minima?

Momentum can be interpreted as discretizing a second-order differential equation with damping.

4.4 Adaptive Learning Rates

Different coordinates may have gradients of very different scales. Adaptive methods address this by normalizing updates.

Define the second-moment accumulator

$$s_{k+1} = \beta_2 s_k + (1 - \beta_2) g_k^2,$$

where the square is taken elementwise.

4.4.1 Inquiry

- Why does dividing by $\sqrt{s_k}$ stabilize training?
- What happens when gradients are sparse?
- Why is per-coordinate adaptivity dangerous?

4.5 The Adam Optimizer

Adam combines momentum and adaptive scaling.

4.5.1 Algorithm

Initialize $m_0 = 0$, $v_0 = 0$.

$$m_{k+1} = \beta_1 m_k + (1 - \beta_1) g_k,$$
$$v_{k+1} = \beta_2 v_k + (1 - \beta_2) g_k^2.$$

Bias correction:

$$\hat{m}_{k+1} = \frac{m_{k+1}}{1 - \beta_1^{k+1}}, \qquad \hat{v}_{k+1} = \frac{v_{k+1}}{1 - \beta_2^{k+1}}.$$

Update:

$$\theta_{k+1} = \theta_k - \eta \frac{\hat{m}_{k+1}}{\sqrt{\hat{v}_{k+1}} + \varepsilon}.$$

4.5.2 Inquiry

- Why is bias correction necessary?
- What goes wrong without ε?
- Why does Adam often converge faster but generalize worse?

4.6 Test-Driven Implementation Strategy

We implement Adam using TDD to avoid silent bugs.

4.6.1 Core Invariants to Test

- Determinism given a fixed random seed.
- Shape consistency of all tensors.
- Non-negativity of second-moment estimates.
- Correct bias correction at small k.

4.7 Exercise Sequence (TDD)

4.7.1 Exercise 1: Gradient Oracle

Write a test that checks a stochastic gradient estimator is unbiased on a quadratic function.

4.7.2 Exercise 2: Momentum Averaging

Verify that m_k equals an exponential moving average of past gradients.

4.7.3 Exercise 3: Second-Moment Accumulator

Test that v_k tracks the empirical variance scale of gradients.

4.7.4 Exercise 4: Bias Correction

On a constant gradient, verify that $\hat{m}_k$ converges immediately to the true gradient.

4.7.5 Exercise 5: Adam Step on Quadratic Bowl

Check that Adam converges to the minimizer of $f(\theta) = |\theta|^2$ from random initialization.

4.8 Failure Modes

Symptom **Cause**

extbfSymptom	**Cause**
Divergence	Learning rate too large
Slow convergence	β_1 too small
Parameter drift	β_2 too large
Poor generalization	Excessive adaptivity

height

extbfSymptom	**Cause**
Divergence	Learning rate too large
Slow convergence	β_1 too small
Parameter drift	β_2 too large
Poor generalization	Excessive adaptivity

Chapter 5

Test-Driven Adam in C (From Scratch)

Purpose

This chapter is a correctness-first, test-driven implementation of the Adam optimizer in `C`. The emphasis is on:

- a minimal, auditable implementation (no external ML frameworks),
- deterministic behavior and numerical stability,
- unit tests that catch the common silent bugs (bias correction, moments, epsilon placement, shape/stride errors),
- extensibility toward HPC settings (SIMD/OpenMP) *after* tests are green.

Guiding Question

How do we implement Adam in plain C so that (i) every mathematical step is testable, and (ii) the code is suitable as a performance baseline for later optimization?

5.1 Scope and Assumptions

We optimize parameters $\theta \in \mathbb{R}^d$ given a gradient vector $g \in \mathbb{R}^d$ supplied by a *gradient oracle*. This chapter focuses on the optimizer only. We will implement:

- Adam state and update step;
- optional weight decay (decoupled AdamW form);

- float/double support via a typedef;
- a tiny test harness (no third-party dependencies required).

5.2 Mathematical Specification

Given hyperparameters $\eta > 0$, $\beta_1, \beta_2 \in (0, 1)$, and $\varepsilon > 0$, Adam performs:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^{\odot 2}, \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \\ heta_t &= \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \end{aligned}$$

All operations are elementwise except scalar multiplications.

5.2.1 AdamW (Decoupled Weight Decay)

If using AdamW with weight decay $\lambda \geq 0$:

$$heta_t \leftarrow \theta_{t-1} - \eta \lambda \theta_{t-1}$$

in addition to the Adam step above. (Decoupled decay is preferred over adding $\lambda\theta$ to gradients.)

5.3 Design Requirements (What Must Be True)

1. **Bias correction correctness:** first few steps must match the analytic formulas.
2. **Second moment non-negativity:** $v_t[i] \geq 0$ always.
3. **Epsilon placement:** ε is added *outside* the square root: $\sqrt{\hat{v}} + \varepsilon$.
4. **No hidden allocations:** all buffers are allocated by the caller or in init.
5. **Deterministic stepping:** given the same inputs, the update is bitwise deterministic (within floating-point expectations across compilers).

5.4 Project Skeleton

A minimal repository layout:

```
adam_c/
include/
adam.h
adam_types.h
test_harness.h
src/
adam.c
adam_init.c
tests/
test_adam_bias.c
test_adam_moments.c
test_adam_quadratic.c
test_adam_adamw.c
Makefile
```

5.5 C API (Small and Testable)

5.5.1 Type Configuration

Use a single scalar type.

```
// adam_types.h
#pragma once
#include <stddef.h>

#ifndef ADAM_SCALAR_T
#define ADAM_SCALAR_T double
#endif

typedef ADAM_SCALAR_T adam_t;
```

5.5.2 Optimizer State

```
// adam.h
#pragma once
#include "adam_types.h"

typedef struct {
size_t d;        // dimension
adam_t lr;       // eta
```

```
adam_t beta1;
adam_t beta2;
adam_t eps;
adam_t weight_decay; // lambda (AdamW); 0 disables

// time step (starts at 0, increments on each step)
unsigned long long t;

// moments
adam_t *m; // length d
adam_t *v; // length d

// cached powers for bias correction
adam_t beta1_pow; // beta1^t
adam_t beta2_pow; // beta2^t
} adam_opt_t;

// init: caller provides buffers m and v of length d
int adam_init(adam_opt_t *opt, size_t d,
adam_t lr, adam_t beta1, adam_t beta2, adam_t eps,
adam_t weight_decay,
adam_t *m_buf, adam_t *v_buf);

// reset moments and time
void adam_reset(adam_opt_t *opt);

// single step: t\theta and grad are length d
void adam_step(adam_opt_t *opt, adam_t *t\theta, const adam_t *grad);
```

5.6 Implementation (Reference)

src/adam$_{i}nit.c$

```
#include "adam.h"

static void zero_vec(adam_t *x, size_t d) {
for (size_t i = 0; i < d; ++i) x[i] = (adam_t)0;
}
```

```
int adam_init(adam_opt_t *opt, size_t d,
adam_t lr, adam_t beta1, adam_t beta2, adam_t eps,
adam_t weight_decay,
adam_t *m_buf, adam_t *v_buf) {
if (!opt || !m_buf || !v_buf || d == 0) return -1;
opt->d = d;
opt->lr = lr;
opt->beta1 = beta1;
opt->beta2 = beta2;
opt->eps = eps;
opt->weight_decay = weight_decay;
opt->t = 0;
opt->m = m_buf;
opt->v = v_buf;
opt->beta1_pow = (adam_t)1;
opt->beta2_pow = (adam_t)1;
zero_vec(opt->m, d);
zero_vec(opt->v, d);
return 0;
}

void adam_reset(adam_opt_t *opt) {
if (!opt) return;
opt->t = 0;
opt->beta1_pow = (adam_t)1;
opt->beta2_pow = (adam_t)1;
for (size_t i = 0; i < opt->d; ++i) {
opt->m[i] = (adam_t)0;
opt->v[i] = (adam_t)0;
}
}
```

src/adam.c

```
#include "adam.h"
#include <math.h>

void adam_step(adam_opt_t *opt, adam_t *t\theta, const adam_t *grad) {
const size_t d = opt->d;
```

```
// t := t + 1 and update cached powers
opt->t += 1;
opt->beta1_pow *= opt->beta1;
opt->beta2_pow *= opt->beta2;

const adam_t one = (adam_t)1;
const adam_t b1 = opt->beta1;
const adam_t b2 = opt->beta2;
const adam_t lr = opt->lr;
const adam_t eps = opt->eps;

const adam_t inv_bias1 = one / (one - opt->beta1_pow);
const adam_t inv_bias2 = one / (one - opt->beta2_pow);

// Optional decoupled weight decay (AdamW)
if (opt->weight_decay != (adam_t)0) {
const adam_t wd = opt->weight_decay;
for (size_t i = 0; i < d; ++i) {
t\theta[i] -= lr * wd * t\theta[i];
}
}

// Moment updates + parameter step
for (size_t i = 0; i < d; ++i) {
const adam_t g = grad[i];

// m_t, v_t
const adam_t m = b1 * opt->m[i] + (one - b1) * g;
const adam_t v = b2 * opt->v[i] + (one - b2) * (g * g);
opt->m[i] = m;
opt->v[i] = v;

// bias corrected
const adam_t mhat = m * inv_bias1;
const adam_t vhat = v * inv_bias2;

// epsilon outside sqrt
```

```
t\theta[i] -= lr * (mhat / (sqrt(vhat) + eps));

}
}
```

5.7 A Tiny Test Harness (No Dependencies)

include/test$_h$arness.h

```
#pragma once
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define ASSERT_TRUE(cond) do {
if (!(cond)) {
fprintf(stderr, "ASSERT_TRUE failed: %s (%s:%d)\n", #cond, **FILE**, **LINE**);
exit(1);
}
} while (0)

#define ASSERT_NEAR(a,b,tol) do {
double _da = (double)(a);
double _db = (double)(b);
double _dt = fabs(_da - _db);
if (_dt > (tol)) {
fprintf(stderr, "ASSERT_NEAR failed: |%s-%s|=%g > %g (%s:%d)\n", #a, #b, _dt, (double)(tol),
exit(1);
}
} while (0)
```

5.8 TDD Exercise Sequence

5.8.1 Exercise 1: Bias Correction on Constant Gradient

Goal: catch the most common Adam bug.
Set $g_t \equiv g$ constant, $m_0 = v_0 = 0$. Show analytically:

$$m_t = (1 - \beta_1^t)g, \quad \hat{m}_t = g.$$

Similarly $\hat{v}_t = g^2$. Therefore the update should be

$$\theta_t = \theta_{t-1} - \eta, \frac{g}{|g| + \varepsilon}.$$

Test: in 1D, with $g = 2$, verify the very first step matches the formula within tolerance.

tests/test$_{a}dam_{b}ias.c$

```
#include "adam.h"
#include "test_harness.h"

int main(void) {
adam_t m[1], v[1];
adam_opt_t opt;
ASSERT_TRUE(adam_init(&opt, 1, (adam_t)0.1, (adam_t)0.9, (adam_t)0.999,
(adam_t)1e-8, (adam_t)0.0, m, v) == 0);

adam_t t\theta[1] = {(adam_t)1.0};
const adam_t grad[1] = {(adam_t)2.0};

adam_step(&opt, t\theta, grad);

// For constant g, bias-corrected mhat=g and vhat=g^2 at t=1.
const adam_t expected = (adam_t)1.0 - (adam_t)0.1 * ((adam_t)2.0 / ((adam_t)2.0 + (adam_t)1e-
ASSERT_NEAR(t\theta[0], expected, 1e-12);

return 0;
}
```

5.8.2 Exercise 2: Second Moment Non-Negativity

Test: feed arbitrary gradients and verify $v[i] \geq 0$ always.

tests/test$_{a}dam_{m}oments.c$

```
#include "adam.h"
#include "test_harness.h"

int main(void) {
enum { d = 8 };
adam_t m[d], v[d];
```

```
adam_opt_t opt;
ASSERT_TRUE(adam_init(&opt, d, (adam_t)1e-2, (adam_t)0.9, (adam_t)0.99,
(adam_t)1e-8, (adam_t)0.0, m, v) == 0);

adam_t t\theta[d];
adam_t g[d];
for (int i = 0; i < d; ++i) { t\theta[i] = (adam_t)0; g[i] = (adam_t)(i - 3); }

for (int k = 0; k < 100; ++k) {
for (int i = 0; i < d; ++i) g[i] = (adam_t)((k + 1) * (i - 3));
adam_step(&opt, t\theta, g);
for (int i = 0; i < d; ++i) ASSERT_TRUE(opt.v[i] >= (adam_t)0);
}

return 0;
}
```

5.8.3 Exercise 3: Epsilon Placement Regression Test

Goal: ensure $\sqrt{\hat{v}} + \varepsilon$ and not $\sqrt{\hat{v} + \varepsilon}$.
Test idea: choose $\hat{v}$ extremely small (e.g. gradient nearly zero) so that the two expressions differ measurably.

5.8.4 Exercise 4: AdamW Decoupled Weight Decay

Test: with $g = 0$, Adam step should be zero but AdamW should decay parameters by $\theta \leftarrow (1 - \eta\lambda)\theta$.

tests/test$_{a}dam_{a}damw.c$

```
#include "adam.h"
#include "test_harness.h"

int main(void) {
adam_t m[1], v[1];
adam_opt_t opt;
const adam_t lr = (adam_t)0.1;
const adam_t wd = (adam_t)0.5;
ASSERT_TRUE(adam_init(&opt, 1, lr, (adam_t)0.9, (adam_t)0.999,
(adam_t)1e-8, wd, m, v) == 0);
```

```
adam_t t\theta[1] = {(adam_t)2.0};
const adam_t grad[1] = {(adam_t)0.0};

adam_step(&opt, t\theta, grad);

const adam_t expected = (adam_t)2.0 * ((adam_t)1.0 - lr * wd);
ASSERT_NEAR(t\theta[0], expected, 1e-12);
return 0;
}
```

5.8.5 Exercise 5: End-to-End on a Quadratic Bowl

Minimize

$$f(\theta) = \frac{1}{2}|\theta|_2^2 \quad \Rightarrow \quad \nabla f(\theta) = \theta.$$

Test: initialize θ_0 and run many Adam steps with $g_t = \theta_t$; verify $|\theta|$ decreases below a threshold.

tests/test$_{a}dam_{q}uadratic.c$

```
#include "adam.h"
#include "test_harness.h"

static adam_t norm2(const adam_t *x, size_t d) {
adam_t s = (adam_t)0;
for (size_t i = 0; i < d; ++i) s += x[i]*x[i];
return (adam_t)sqrt((double)s);
}

int main(void) {
enum { d = 4 };
adam_t m[d], v[d];
adam_opt_t opt;
ASSERT_TRUE(adam_init(&opt, d, (adam_t)1e-1, (adam_t)0.9, (adam_t)0.999,
(adam_t)1e-8, (adam_t)0.0, m, v) == 0);

adam_t t\theta[d] = {(adam_t)5.0,(adam_t)-3.0,(adam_t)2.0,(adam_t)-1.0};
adam_t g[d];
```

```
const adam_t n0 = norm2(t\theta, d);
for (int k = 0; k < 2000; ++k) {
for (int i = 0; i < d; ++i) g[i] = t\theta[i];
adam_step(&opt, t\theta, g);
}
const adam_t n1 = norm2(t\theta, d);
ASSERT_TRUE(n1 < (adam_t)1e-2 * n0);
return 0;
}
```

5.9 Makefile (Minimal)

```
CC ?= gcc
CFLAGS ?= -O2 -std=c11 -Wall -Wextra -Iinclude
LDFLAGS ?= -lm

SRC = src/adam.c src/adam_init.c

TESTS =
tests/test_adam_bias
tests/test_adam_moments
tests/test_adam_adamw
tests/test_adam_quadratic

all: $(TESTS)

tests/%: tests/%.c $(SRC)
$(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

test: all
@for t in $(TESTS); do echo "[RUN] $$t"; $$t; echo "[OK ] $$t"; done

clean:
rm -f $(TESTS)
```

5.10 Debugging Checklist

When a test fails, the most likely issues are:

- using β^t with wrong t indexing (off-by-one);
- bias-correction computed using the *old* power rather than updated power;
- epsilon placed inside the square root;
- integer truncation when computing norms or tolerances;
    - accidental aliasing between exttttθandextttgradbuffers.

5.11 HPC Extensions (After Tests are Green)

1. **Incremental vectorization:** replace the inner loop with SIMD intrinsics.
2. **OpenMP:** parallelize the parameter dimension for large d.
3. **Mixed precision:** keep moments in float, accumulate in double.
4. **Fused kernels:** combine weight decay, moments, and update in one pass.

5.12 Checkpoint

By the end of this chapter you should have a C repository where:

- extttmake test executes all tests and returns success;
- the Adam step matches analytic bias-corrected formulas at early iterations;
- the quadratic-bowl end-to-end test demonstrates stable convergence;
- the code is ready to be optimized without sacrificing correctness.

Chapter 6

Optimizing the Square Size: Feasibility and Joint Optimization

Up to this point, simulated annealing has been used to optimize circle positions inside a *fixed* square of side length L. We now address a more fundamental geometric question:

> What is the smallest square side length $L^\star$ for which N identical circles of radius r can be packed without overlap?

This chapter develops two complementary solver strategies:

1. **Feasibility search**: treat L as a parameter, test feasibility via SA, and search for the minimal feasible L.
2. **Joint optimization**: include L directly in the optimization state and minimize a soft energy that trades off feasibility and size.

Both approaches are built using the same inquiry-driven workflow:

$$\text{Geometry} \longrightarrow \text{Invariant} \longrightarrow \text{Test} \longrightarrow \text{Code}.$$

Throughout, we distinguish between:

- *guaranteed bounds* derived from geometry, and
- *heuristic behavior* introduced by stochastic optimization.

6.1 Why Optimizing L Is Hard

Packing problems are inherently nonconvex. Even for fixed L, feasibility is not a simple constraint satisfaction problem; it must be discovered algorithmically.
Optimizing L adds a second layer of difficulty:

- reducing L increases boundary pressure,
- smaller L amplifies overlap penalties,
- feasibility becomes a probabilistic outcome of the inner solver.

We therefore separate the problem into two solver paradigms.

6.2 Approach I: Feasibility Search

6.2.1 Problem Formulation

For a fixed L, define *feasibility* as:

$$E_{\text{pair}}(X) = 0 \quad \text{and} \quad E_{\text{wall}}(X, L) = 0.$$

Numerically, we test:

$$E_{\text{pair}}(X) \leq \varepsilon, \qquad E_{\text{wall}}(X, L) \leq \varepsilon,$$

for a small tolerance ε.
The feasibility solver asks:

For which values of L does simulated annealing find a feasible configuration?

This motivates an outer search in L.

6.2.2 Guaranteed Lower Bounds

Before running any optimization, geometry alone already constrains L.

Diameter Bound

At minimum, a single circle must fit:

$$L \geq 2r.$$

This is a trivial but unavoidable constraint.

Area Bound

The total area of all circles is:

$$A_{\text{circles}} = N\pi r^2.$$

Since the square has area L^2, any feasible packing must satisfy:

$$L^2 \geq N\pi r^2 \quad \Rightarrow \quad L \geq r\sqrt{N\pi}.$$

This bound is *necessary* and therefore guaranteed.

Density-Aware Bound (Optional)

The maximal packing density of equal circles in the plane is

$$\delta_{\max} = \frac{\pi}{2\sqrt{3}} \approx 0.9069.$$

Even ignoring boundary effects,

$$L^2 \geq \frac{N\pi r^2}{\delta_{\max}} \quad \Rightarrow \quad L \geq r\sqrt{\frac{N\pi}{\delta_{\max}}}.$$

Boundary effects only reduce achievable density, so this remains a valid lower bound.

Combined lower bound. In practice, we use:

$$L_{\text{lo}} = \max\left(2r,\ r\sqrt{N\pi}\right),$$

optionally replacing the area bound with the density-aware bound.

6.2.3 Guaranteed Upper Bounds

Lower bounds alone are insufficient; feasibility search also requires a value of L that is *known* to be feasible.

Grid Construction

A simple constructive packing places circles on a square grid with spacing $2r$:

$$x_{ij} = (r + 2ri,\ r + 2rj).$$

Let

$$k = \lceil \sqrt{N} \rceil.$$

Then a $k \times k$ grid fits inside a square of side length:

$$L_{\text{grid}} = 2rk.$$

This configuration has:

$$E_{\text{pair}} = 0, \qquad E_{\text{wall}} = 0,$$

and therefore provides a *constructive feasibility witness.*

Upper bound. We set:

$$L_{\text{hi}} = 2r\lceil\sqrt{N}\rceil.$$

This bound is crude but guaranteed and deterministic.

6.2.4 Outer Bisection on L

With a bracket $[L_{\text{lo}}, L_{\text{hi}}]$ in hand, we perform an outer bisection search.

1. Set $L_{\text{mid}} = \frac{1}{2}(L_{\text{lo}} + L_{\text{hi}})$.
2. Run simulated annealing at fixed L_{mid}.
3. If a feasible configuration is found:
   - record the witness,
   - set $L_{\text{hi}} \leftarrow L_{\text{mid}}$.
4. Otherwise, set $L_{\text{lo}} \leftarrow L_{\text{mid}}$.

Because simulated annealing is stochastic, feasibility is a *probabilistic predicate.* To reduce false negatives, the solver may retry each L_{mid} with multiple deterministic seeds.
The algorithm terminates when:

$$L_{\text{hi}} - L_{\text{lo}} \leq \text{tolerance}.$$

6.3 Approach II: Joint Optimization of (X, L)

An alternative is to optimize positions and box size simultaneously.

6.3.1 Energy Model

We define a soft objective:

$$E(X, L) = E_{\text{pair}}(X) + E_{\text{wall}}(X, L) + \alpha L,$$

with $\alpha > 0$ controlling the pressure to shrink the square.

Key properties:

- $E(X, L) \geq 0$ always,
- feasibility corresponds to $E_{\text{pair}} = E_{\text{wall}} = 0$,
- reducing L is rewarded only when geometry allows it.

6.3.2 Joint Proposal Kernel

The state space is now (X, L). At each SA step:

- with probability p, propose a move of a single circle position,
- with probability $1 - p$, propose a move in L:

$$L' = \max(2r,\ L + \delta_L), \qquad \delta_L \sim \mathcal{N}(0, \sigma_L^2).$$

Acceptance is handled by the same Metropolis rule using the correct ΔE.

6.3.3 Interpretation

Joint optimization replaces hard feasibility with a soft competition:

- early at high temperature, L can shrink aggressively,
- overlaps and wall violations are temporarily tolerated,
- at low temperature, only feasible shrinkage survives.

Unlike feasibility search, this approach produces a *single trajectory* rather than an outer–inner loop, but its outcome depends sensitively on α and the proposal scales.

6.4 Comparison of the Two Approaches

	Feasibility Search	Joint Optimization
Guarantees	Explicit feasibility	Soft, parameter-dependent
Bounds	Required	Not required
Complexity	Outer × inner	Single SA run
Interpretability	Clear witnesses	Continuous trade-off
Sensitivity	Low (with retries)	High (α, σ_L)

In practice:

- feasibility search is preferable when correctness and reproducibility matter;
- joint optimization is useful for exploratory or heuristic searches.

6.5 Checkpoint

At the end of this chapter, you should be able to:

- derive guaranteed lower and upper bounds for $L^\star$;
- construct feasibility witnesses without optimization;
- implement an outer bisection loop driven by simulated annealing;
- understand the trade-offs of joint (X, L) optimization;
- distinguish mathematical guarantees from heuristic behavior.

In the next chapter, we revisit these solvers from a gradient-based perspective and contrast stochastic annealing with deterministic optimization methods.

Chapter 7

C Implementation of Feasibility and Joint Optimization Solvers

The previous chapter developed two solver strategies for optimizing the square side length:

1. feasibility search via an outer loop on L,
2. joint optimization of (X, L) using simulated annealing.

This chapter translates those ideas into a *minimal, test-driven C implementation* that extends the existing simulated annealing baseline.
As before, the emphasis is not performance, but:

- correctness,
- explicit invariants,
- deterministic behavior under fixed seeds,
- and instrumentation suitable for scientific reasoning.

All additions are layered *on top of* the existing SA core, rather than rewriting it.

7.1 Design Principles for the Extension

Before writing code, we make several design commitments.

1. **No changes to the inner SA logic unless required.** The fixed-L simulated annealing kernel remains valid and reusable.
2. **Feasibility is a predicate, not a heuristic.** It is expressed as a function of (X, L) and tested independently.

3. **Bounds come from geometry, not tuning.** Lower and upper bounds for L are computed deterministically.

4. **Solvers compose existing components.** New solvers call the existing SA driver instead of duplicating logic.

These principles keep the codebase modular and auditable.

7.2 New Supporting Primitives

7.2.1 Feasibility Predicate

The most basic new operation is a feasibility test.

Definition. A configuration (X, L) is feasible if:

$$E_{\text{pair}}(X) = 0 \quad \text{and} \quad E_{\text{wall}}(X, L) = 0,$$

up to a small numerical tolerance ε.

C interface.

```
int is_feasible(const Vec2* X, size_t N,
                double r, double L, double eps);
```

Design notes.

- This function is pure and deterministic.
- It relies only on energy components already implemented.
- It is used by:
  - the feasibility solver,
  - tests for geometric sanity,
  - post-run validation.

Tests.

- Known non-overlapping interior configurations return feasible.
- Any overlap or wall violation returns infeasible.
- Degenerate cases ($N = 1$, $L < 2r$) behave correctly.

7.2.2 Lower and Upper Bounds on L

To initialize feasibility search, we compute a deterministic bracket.

Lower bounds

Two guaranteed lower bounds are implemented:

- Diameter bound:
$$L \geq 2r.$$
- Area bound:
$$L \geq r\sqrt{N\pi}.$$

In code:

```
double lower_bound_area(size_t N, double r);
double lower_bound_basic(size_t N, double r);
```

The solver uses:

$$L_{\text{lo}} = \max(2r,\ r\sqrt{N\pi}).$$

Upper bound via grid construction

A constructive packing places circles on a square grid with spacing $2r$.

C interface.

```
double upper_bound_grid(size_t N, double r);
void init_grid(Vec2* X, size_t N, double r, double L);
```

This guarantees:

$$L_{\text{hi}} = 2r\lceil\sqrt{N}\rceil$$

and provides an explicit feasible witness.

Tests.

- Grid initialization yields zero pair and wall energy.
- Returned $L_{\text{hi}} \geq L_{\text{lo}}$ for all N.

7.3 Feasibility Solver in C

7.3.1 High-Level Structure

The feasibility solver performs an outer bisection on L, using the existing fixed-L SA kernel as a feasibility oracle.

Interface.

```
FeasibleSolveResult solve_feasible_bisect(
    size_t N, double r,
    double L_lo, double L_hi,
    double eps_feas,
    double T0, double gamma,
    size_t n_steps, double sigma,
    unsigned long seed,
    int retries_per_L
);
```

Algorithm.

1. Initialize $(L_{\text{lo}}, L_{\text{hi}})$ from geometric bounds.
2. While $L_{\text{hi}} - L_{\text{lo}}$ exceeds tolerance:
   (a) Set $L_{\text{mid}} = \frac{1}{2}(L_{\text{lo}} + L_{\text{hi}})$.
   (b) Run SA at fixed L_{mid}.
   (c) If a feasible configuration is found:
      - record the witness,
      - set $L_{\text{hi}} \leftarrow L_{\text{mid}}$.
   (d) Otherwise set $L_{\text{lo}} \leftarrow L_{\text{mid}}$.

Handling stochasticity. Because SA is probabilistic, feasibility tests may fail spuriously. The solver supports multiple retries per L, using deterministic seed offsets to preserve reproducibility.

Diagnostics. Each bisection iteration may log:

- tested L,
- feasibility result,
- best energy achieved,
- best E_{pair} and E_{wall}.

7.4 Joint Optimization Solver in C

7.4.1 Extended State Space

Joint optimization treats (X, L) as a single state.

State variables.

- $X \in \mathbb{R}^{N\times 2}$,
- $L \geq 2r$.

Energy.

$$E(X, L) = E_{\text{pair}}(X) + E_{\text{wall}}(X, L) + \alpha L.$$

7.4.2 Proposal Kernel

At each step:

- with probability p, propose a circle move (as in the baseline),
- with probability $1 - p$, propose:

$$L' = \max(2r,\ L + \delta_L), \qquad \delta_L \sim \mathcal{N}(0, \sigma_L^2).$$

Only the wall energy and αL term change for an L-move.

7.4.3 C Interface

```
JointAnnealResult anneal_joint_run(
    const Vec2* X0, size_t N,
    double L0, double r, double alpha,
    double T0, double gamma,
    size_t n_steps,
    double sigma_x, double sigma_L,
    double p_move_x,
    unsigned long seed
);
```

Trace logging. The joint solver records:

- $E(t)$,

- $L(t)$,
- temperature,
- acceptance decisions,
- move type (position vs. L).

This allows direct inspection of how L evolves during annealing.

7.5 Testing Strategy

The new solvers introduce both mathematical and algorithmic invariants.

7.5.1 Mathematical tests

- Lower bounds are never violated.
- Grid upper bound is always feasible.
- ΔE for L-moves matches recomputed energy differences.

7.5.2 Algorithmic tests

- Bisection interval shrinks monotonically.
- Best-so-far feasible L never increases.
- Joint solver respects $L \geq 2r$ at all times.

7.5.3 Empirical sanity checks

- Larger α produces smaller final L on average.
- Acceptance rate decreases as temperature cools.

These are treated as regression tests with tolerant thresholds, not strict invariants.

7.6 Checkpoint

At the end of this chapter, you should have:

- a principled feasibility solver built from geometry and SA,
- a joint (X, L) annealer with explicit proposal semantics,

- deterministic, testable C code extending the original baseline,
- diagnostics that explain solver behavior rather than obscure it.

The next chapter turns to *performance*: spatial acceleration, parallelism, and the cost of correctness.