Discrete and Continuous Methods for Packing Problems

Your Name

January 21, 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

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

where $E_{extoverlap}$ 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

Simulated annealing is grounded in equilibrium statistical mechanics.

1.4.1 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.

1.4.2 Metropolis Acceptance Rule

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

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

1.4.3 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.

1.5 Simulated Annealing as a Limit Process

Simulated annealing proceeds by lowering the temperature:

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

1.5.1 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.

1.6 Algorithmic Ingredients

Every simulated annealing algorithm consists of:

1. extbfState space: configurations x (and possibly L).

2. extbfEnergy: objective plus penalties.
3. extbfProposal kernel: how candidate states are generated.
4. extbfAcceptance rule: Metropolis criterion.
5. extbfCooling schedule: temperature decay.

Failure of any component compromises the method.

1.7 Energy Design for Circle Packing

Hard constraints destroy ergodicity. Instead we use soft penalties.

1.7.1 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:

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

1.7.2 Inquiry

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

1.8 Proposal Mechanisms

Common proposal strategies include:

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

A key heuristic is

$$extproposalscale \propto \sqrt{T}.$$

1.9 Optimizing the Square Size

Two approaches are common:

1.9.1 Feasibility Search

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

1.9.2 Joint Optimization

Optimize (x, L) jointly using

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

1.9.3 Inquiry

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

1.10 Common Failure Modes

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

1.11 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.

1.12 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 2

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 extbftrustworthiness: 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.).

2.1 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:

$$extTheory; \longrightarrow; extAlgorithmicInvariant; \longrightarrow; extUnitTest; \longrightarrow; extCode.$$

2.2 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.

2.3 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.

2.4 Design Specification: Na"ive Baseline

2.4.1 State

A state consists of

$$(X, L), \qquad X \in \mathbb{R}^{Nimes2},\ L > 0,$$

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

2.4.2 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:

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

2.4.3 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.

2.4.4 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.

2.4.5 Acceptance Rule

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

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

2.4.6 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.

2.5 Inquiry-Driven TDD Sequence

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

2.5.1 Exercise 1: Deterministic Randomness

extbfInvariant: identical seeds imply identical trajectories.

extbfTest:

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

2.5.2 Exercise 2: Energy Sanity

extbfInvariants:

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

2.5.3 Exercise 3: Symmetry

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

2.5.4 Exercise 4: Proposal Locality

extbfInvariants:

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

2.5.5 Exercise 5: Metropolis Correctness

extbfInvariants:

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

2.5.6 Exercise 6: Best-So-Far Monotonicity

extbfInvariant: the best recorded energy is nonincreasing over time.

2.5.7 Exercise 7: Temperature Effects

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

2.6 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.

2.7 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.

2.8 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.

2.9 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.

2.10 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 3

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.

3.1 From Deterministic to Stochastic Gradients

Consider the finite-sum optimization problem

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

3.1.1 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

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

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

3.2 Stochastic Gradient Descent

The SGD iteration is

$$heta_{k+1} = heta_k - \eta_k g_k.$$

3.2.1 Inquiry

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

3.2.2 Key Phenomena

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

3.3 Momentum as Time Averaging

To reduce variance and accelerate convergence, introduce momentum:

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

3.3.1 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.

3.4 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.

3.4.1 Inquiry

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

3.5 The Adam Optimizer

Adam combines momentum and adaptive scaling.

3.5.1 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:

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

3.5.2 Inquiry

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

3.6 Test-Driven Implementation Strategy

We implement Adam using TDD to avoid silent bugs.

3.6.1 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.

3.7 Exercise Sequence (TDD)

3.7.1 Exercise 1: Gradient Oracle

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

3.7.2 Exercise 2: Momentum Averaging

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

3.7.3 Exercise 3: Second-Moment Accumulator

Test that v_k tracks the empirical variance scale of gradients.

3.7.4 Exercise 4: Bias Correction

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

3.7.5 Exercise 5: Adam Step on Quadratic Bowl

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

3.8 Failure Modes

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

3.9 Conceptual Summary

Stochastic gradient methods trade deterministic descent for scalable, noise-tolerant optimization. Adam succeeds by combining three ideas: stochasticity for exploration, momentum for acceleration, and adaptive scaling for numerical stability. Understanding these components independently is essential for using Adam responsibly rather than as a black box.

Chapter 4

Test-Driven Adam in C (From Scratch)

Purpose

This chapter is a correctness-first, test-driven implementation of the Adam optimizer in extttC. 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?

4.1 Scope and Assumptions

We optimize parameters $heta \in \mathbb{R}^d$ 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).

4.2 Mathematical Specification

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

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t,\ v_t \quad = \beta_2 v_{t-1} + (1-\beta_2) g_t^{\odot 2},\ \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t},\ heta_t \quad = heta_{t-1} - \eta, \frac{}{\sqrt{}}$$

All operations are elementwise except scalar multiplications.

4.2.1 AdamW (Decoupled Weight Decay)

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

$$heta_t \leftarrow heta_t - \eta, \lambda, heta_{t-1}$$

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

4.3 Design Requirements (What Must Be True)

1. extbfBias correction correctness: first few steps must match the analytic formulas.
2. extbfSecond moment non-negativity: $v_t[i] \geq 0$ always.
3. extbfEpsilon placement: ε is added *outside* the square root: $\sqrt{\hat{v}} + \varepsilon$.
4. extbfNo hidden allocations: all buffers are allocated by the caller or in init.
5. extbfDeterministic stepping: given the same inputs, the update is bitwise deterministic (within floating-point expectations across compilers).

4.4 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
```

4.5 C API (Small and Testable)

4.5.1 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;
```

4.5.2 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: theta and grad are length d
void adam_step(adam_opt_t *opt, adam_t *theta, const adam_t *grad);
```

4.6 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 *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) {
theta[i] -= lr * wd * 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
theta[i] -= lr * (mhat / (sqrt(vhat) + eps));

}
}
```

4.7 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)
```

4.8 TDD Exercise Sequence

4.8.1 Exercise 1: Bias Correction on Constant Gradient

extbfGoal: 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

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

extbfTest: 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 theta[1] = {(adam_t)1.0};
const adam_t grad[1] = {(adam_t)2.0};

adam_step(&opt, 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(theta[0], expected, 1e-12);

return 0;
}
```

4.8.2 Exercise 2: Second Moment Non-Negativity

extbfTest: 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 theta[d];
```

```
adam_t g[d];
for (int i = 0; i < d; ++i) { 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, theta, g);
for (int i = 0; i < d; ++i) ASSERT_TRUE(opt.v[i] >= (adam_t)0);
}

return 0;
}
```

4.8.3 Exercise 3: Epsilon Placement Regression Test

extbfGoal: ensure $\sqrt{\hat{v}} + \varepsilon$ and not $\sqrt{\hat{v} + \varepsilon}$.

extbfTest idea: choose $\hat{v}$ extremely small (e.g. gradient nearly zero) so that the two expressions differ measurably.

4.8.4 Exercise 4: AdamW Decoupled Weight Decay

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

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

adam_step(&opt, theta, grad);
```

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

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

Minimize

$$f(heta) = frac12|heta|_2^2 \quad \Rightarrow \quad \nabla f(heta) = heta.$$

extbfTest: initialize $heta_0$ and run many Adam steps with $g_t = heta_t$; verify $|heta|$ 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 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(theta, d);
for (int k = 0; k < 2000; ++k) {
for (int i = 0; i < d; ++i) g[i] = theta[i];
adam_step(&opt, theta, g);
}
```

```
const adam_t n1 = norm2(theta, d);
ASSERT_TRUE(n1 < (adam_t)1e-2 * n0);
return 0;
}
```

4.9 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)
```

4.10 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 extttttheta and extttgrad buffers.

4.11 HPC Extensions (After Tests are Green)

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

4.12 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 5

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.

5.1 From Deterministic to Stochastic Gradients

Consider the finite-sum optimization problem

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

5.1.1 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

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

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

5.2 Stochastic Gradient Descent

The SGD iteration is

$$heta_{k+1} = heta_k - \eta_k g_k.$$

5.2.1 Inquiry

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

5.2.2 Key Phenomena

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

5.3 Momentum as Time Averaging

To reduce variance and accelerate convergence, introduce momentum:

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

5.3.1 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.

5.4 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.

5.4.1 Inquiry

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

5.5 The Adam Optimizer

Adam combines momentum and adaptive scaling.

5.5.1 Algorithm

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

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

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:

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

5.5.2 Inquiry

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

5.6 Test-Driven Implementation Strategy

We implement Adam using TDD to avoid silent bugs.

5.6.1 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.

5.7 Exercise Sequence (TDD)

5.7.1 Exercise 1: Gradient Oracle

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

5.7.2 Exercise 2: Momentum Averaging

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

5.7.3 Exercise 3: Second-Moment Accumulator

Test that v_k tracks the empirical variance scale of gradients.

5.7.4 Exercise 4: Bias Correction

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

5.7.5 Exercise 5: Adam Step on Quadratic Bowl

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

5.8 Failure Modes

extbfSymptom	extbfCause heightDivergence
Learning rate too large Slow convergence	β_1 too small Parameter drift
β_2 too large Poor generalization	Excessive adaptivity

5.9 Conceptual Summary

Stochastic gradient methods trade deterministic descent for scalable, noise-tolerant optimization. Adam succeeds by combining three ideas: stochasticity for exploration, momentum for acceleration, and adaptive scaling for numerical stability. Understanding these components independently is essential for using Adam responsibly rather than as a black box.

Chapter 6

Discrete Formulations of Packing via Independent Sets

6.1 From Continuous Packing to Discrete Candidates

We begin with a classical continuous packing problem: given a compact container $\Omega \subset \mathbb{R}^2$ (e.g. a square or disk) and identical circles of radius r, place as many circles as possible inside Ω without overlap.

In the continuous setting, the configuration space is infinite-dimensional, and the problem is highly nonconvex. To make algorithmic progress, we introduce a *discretization of configuration space*.

6.1.1 Candidate placements

Let $\mathcal{P} = \{p_1, \ldots, p_M\} \subset \Omega$ be a finite set of candidate circle centers. These may arise from:

- a uniform Cartesian grid,
- a union of rotated grids to reduce anisotropy,
- or a local patch extracted from a continuous solver.

Each candidate p_i represents the placement of a circle centered at p_i.
A candidate is *feasible* if the entire disk lies inside the container:

$$\operatorname{dist}(p_i, \partial\Omega) \geq r.$$

Only feasible candidates are retained.

6.1.2 Conflict detection

Two candidates p_i and p_j are said to be *in conflict* if placing circles at both locations would cause overlap:

$$\|p_i - p_j\| < 2r.$$

This pairwise condition captures all geometric constraints of the packing problem once the candidate set has been fixed.

6.2 Graph Construction

The discrete packing problem can now be represented as a graph.

Definition 6.1 (Conflict Graph)**.** Let $G = (V, E)$ be a graph where:

- Each vertex $i \in V$ corresponds to a candidate placement $p_i \in \mathcal{P}$.
- An edge $(i, j) \in E$ exists if and only if p_i and p_j are in conflict.

The graph G encodes all pairwise incompatibilities between candidate placements. Importantly, this graph is:

- geometric,
- sparse (conflicts are local),
- independent of the optimization method used later.

6.3 Packing as an Independent Set Problem

A valid packing corresponds to a selection of candidates such that no two selected placements conflict.

Definition 6.2 (Independent Set)**.** A subset $S \subset V$ is an *independent set* if no two vertices in S share an edge.

6.3.1 Maximum vs. maximal independent sets

Two notions are relevant:

- A *maximal* independent set cannot be extended by adding another vertex.
- A *maximum* independent set has the largest possible cardinality.

In packing problems, we are interested in the *maximum independent set* (MIS), since its cardinality corresponds to the maximum number of circles that can be placed using the candidate set.

Problem 6.3 (Discrete Packing via MIS)**.** Given a conflict graph $G = (V, E)$, find

$$\max_{S \subset V} |S| \quad extsuchthatSextisanindependentset.$$

This formulation is exact for the discretized problem and separates geometry (from graph construction) from combinatorial optimization.

6.4 Integer Linear Programming Formulation

The maximum independent set problem admits a standard integer programming formulation.

6.4.1 Binary decision variables

Introduce variables

$$x_i \in \{0, 1\}, \quad i \in V,$$

where $x_i = 1$ indicates that candidate p_i is selected.

6.4.2 MILP formulation

The packing problem becomes:

$$\max \quad \sum_{i \in V} x_i \tag{6.1}$$

$$extsubjectto \quad x_i + x_j \leq 1, \quad \forall (i, j) \in E, \tag{6.2}$$

$$x_i \in \{0, 1\}, \quad \forall i \in V. \tag{6.3}$$

Each constraint $x_i + x_j \leq 1$ enforces non-overlap for a conflicting pair. This formulation is exact and captures all geometric constraints implicitly through the graph.

6.5 LP Relaxation and Its Interpretation

Solving the MILP directly is computationally expensive. A standard relaxation replaces integrality by bounds:

$$x_i \in [0, 1].$$

6.5.1 LP relaxation

The relaxed problem is:

$$\max \quad \sum_{i \in V} x_i \tag{6.4}$$

$$extsubjectto \quad x_i + x_j \leq 1, \quad \forall (i,j) \in E, \tag{6.5}$$

$$0 \leq x_i \leq 1. \tag{6.6}$$

This linear program provides:

- an upper bound on the true packing number,
- fractional solutions that encode local packing density.

In geometric settings, fractional values often highlight regions of high structural order even before integrality is enforced.

6.6 Algorithmic Solution via Constraint Generation

The full conflict graph may contain a large number of edges. However, most constraints are never active in optimal solutions.

6.6.1 Lazy constraint generation

An efficient strategy is to generate constraints iteratively:

1. Start with variables $x_i \in [0,1]$ and a minimal constraint set.
2. Solve the LP.
3. Detect violated conflict constraints:

$$x_i + x_j > 1 \quad extforsomeconflictingpair(i,j).$$

4. Add the violated constraints.
5. Repeat until no violations remain.

Because conflicts are local, the number of active constraints remains manageable.

6.6.2 Transition to integer solutions

Once the LP relaxation stabilizes, integrality constraints are reinstated:

$$x_i \in \{0, 1\}.$$

The resulting problem is solved using a *branch-and-cut* strategy:

- LP relaxation provides bounds,
- branching enforces integrality,
- constraint generation continues as needed.

This approach yields exact solutions for local patch problems and provides certificates of optimality.

6.7 Interpretation and Scope

This discrete formulation has several important properties:

- It cleanly separates geometry from optimization.
- It applies equally to circles, polygons, and more general shapes.
- It is well suited for *local patch analysis*, where structure such as hexagonal order can emerge without being prescribed.

However, the method does not scale to large global packings and is best used as a:

- verification tool,
- local structure discovery method,
- or subproblem within a hybrid continuous–discrete pipeline.

In later chapters, we will combine this discrete machinery with continuous relaxation and periodic boundary conditions to study bulk packing structure.