TAOCP Vol. 2 + Numerical Recipes
Numerical Calculus Study Deck

Kanban-oriented StoryCards grouped by Phases

Kanban Board Overview

Kanban Columns & Tags

Board Columns:

- **Backlog** – All reading and implementation cards not yet started.
- **In Progress** – Cards currently being read, coded, or experimented on.
- **Review / Experiments** – Numerical experiments, derivations, error analysis, refactors.
- **Done / Consolidated** – Code implemented, tested, documented, and integrated into the library.

Recommended Tags (examples):

- **Book tags:** `Book:TAOCP2`, `Book:NR`.
- **Module tags:** `Module:core-fp`, `Module:core-poly`, `Module:interp`, `Module:quad`, `Module:ode`, `Module:nonlin`, `Module:linalg`, `Module:rand`, `Module:optim`, `Module:stats`.
- **Work type tags:** `Reading`, `Implementation`, `Experiment`, `Design`, `Refactor`.

Usage Pattern:

- Each *StoryCard* in this document corresponds to a Kanban card.
- Start with Phase 0 cards in *Backlog*. Pull into *In Progress* with a strict WIP limit (e.g., max 2–3 cards).
- Move to *Review / Experiments* once code compiles and basic tests run.
- Move to *Done / Consolidated* only when:
  - There is at least one unit test or numerical experiment.
  - The API is sketched and documented in the repo.
  - You have brief notes in your study log.

Phase 0 – Environment & Test Harness

Phase 0 Overview – Environment & Test Harness

Goal: Be able to quickly prototype, run, and visualize numerical methods from TAOCP Vol. 2 and Numerical Recipes.

Primary Focus:

- Project layout and build system.
- Test harness and plotting workflow.
- Basic utilities for timing and error measurement.

Module Tie-ins:

- `Module:all` (infrastructure for every module).

Exit Criteria:

- You can write a small numerical routine, compile it, run it, compute error metrics, and plot the results with minimal friction.
- The repo layout is stable enough that later phases only add modules, not re-architect the whole tree.

Phase 0 Card 1 – Set Up Numerical Playground Repo

Description: Create a dedicated repository to host your computational math system and numerical experiments.

Checklist:

☐ Create a root layout such as:

- `core/` (floating point, utilities, polynomials).
- `interp/` (interpolation and approximation).
- `integrate/` (quadrature / integration).
- `ode/` (ordinary differential equations).
- `opt/` (optimization).
- `rand/` (random numbers and Monte Carlo).
- `nonlin/` (root finding and nonlinear systems).
- `linalg/` (linear algebra).
- `stats/` (error/statistics tools).
- `examples/`, `tests/`, `docs/`.

☐ Add a basic C/C++ build system:

- `CMakeLists.txt` or equivalent build file.
- Configurable build type (Debug/Release).

☐ Integrate a simple test harness:

- Minimal custom test main, or
- Catch2 / GoogleTest wired into the build.

☐ Add a plotting hook:

- Helper to dump `(x, y)` data to CSV.
- Document how to plot via Python/Matplotlib or gnuplot.

Deliverables:

- Repository skeleton committed (e.g., Git).
- `README.md` describing layout, build, and plotting workflow.

Tags: `Infra`, `Module:all`, `Implementation`

Phase 0 Card 2 – Implement Timing & Error Metrics Utilities

Description: Provide reusable helpers for timing functions and computing numerical error metrics.

Checklist:

☐ Implement timing helpers:

- Wall-clock timer based on `std::chrono` (or platform equivalent).
- Scope-based timer to measure function runtimes.

☐ Implement error metric utilities:

- Absolute error: $|x - x_{\text{true}}|$.
- Relative error: $\frac{|x - x_{\text{true}}|}{|x_{\text{true}}|}$.
- Max norm over a dataset.
- RMS error over samples.

☐ Implement convergence-rate estimator:

- Given error values for step sizes $h, h/2, h/4$, estimate observed order p.

☐ Add minimal tests:

- Check that zero error is reported correctly.
- Check that known error cases behave as expected.

Deliverables:

- `core/timing.h`, `core/timing.cpp`.
- `core/error_metrics.h`, `core/error_metrics.cpp`.
- A small example program demonstrating timing and error calculations.

Tags: `Module:core-fp`, `Implementation`, `Experiment`

Phase 1 – Core Arithmetic, Error & Representation

Phase 1 Overview – Core Arithmetic, Error & Representation

Goal: Understand how the machine represents numbers and how that shapes numerical error in calculus algorithms.

Primary Sources:

- TAOCP Vol. 2, Chapter 4: arithmetic, floating-point, polynomial and power-series arithmetic.
- Numerical Recipes:
  - Chapter 1: error, accuracy, stability.
  - Chapter 5: evaluation of functions (polynomials, series).

Module Tie-ins:

- `core::fp` (floating-point utilities).
- `core::poly` (polynomial arithmetic and evaluation).

Exit Criteria:

- You can quantify and visualize rounding error and cancellation.
- You have a robust implementation of polynomial evaluation (Horner) with tests.

Phase 1 Card 1 – Read Floating-Point Arithmetic & Rounding

Description: Build theoretical understanding of floating-point representation, rounding, and error propagation.

Reading Tasks:

☐ TAOCP Vol. 2, sections 4.1–4.2:

- Positional number systems.
- Floating-point formats and rounding modes.

☐ Numerical Recipes, Section 1.1:

- Error, accuracy, and stability.
- Conditioning vs. algorithmic stability.

Notes Deliverable:

- A one-page summary covering:
  - Definition of machine epsilon for your platform.
  - Examples of catastrophic cancellation.
  - Distinction between well-conditioned problems and stable algorithms.

Tags: `Reading`, `Book:TAOCP2`, `Book:NR`, `Module:core-fp`

Phase 1 Card 2 – Implement Safe Floating-Point Utilities

Description: Translate the theory into a `core::fp` utility module.

Checklist:

☐ Implement `double ulp(double x)`:

- Approximate the distance between representable numbers around `x`.

☐ Implement `bool nearly_equal(double a, double b)`:

- Combine absolute and relative tolerance.

☐ Implement robust summation:

- Naive summation for baseline.
- Kahan summation or pairwise summation.

☐ Add tests:

- Show difference between naive and Kahan for ill-conditioned sums.
- Verify that `nearly_equal` behaves correctly for small and large magnitudes.

Deliverables:

- `core/fp.h`, `core/fp.cpp`.
- A small example that sums a sequence known to illustrate cancellation, with printed error statistics.

Tags: `Module:core-fp`, `Implementation`, `Experiment`

Phase 1 Card 3 – Read & Implement Polynomial Evaluation

Description: Use TAOCP and NR to implement efficient polynomial evaluation and test its numerical behavior.

Reading Tasks:

☐ TAOCP Vol. 2, section 4.6 (polynomial arithmetic, Horner's rule).

☐ Numerical Recipes, section 5.1 (polynomials and rational functions).

Implementation Tasks:

☐ Design a polynomial representation:

- Coefficient array in ascending order (constant term first).
- Or a small `Poly` class with degree and coefficients.

☐ Implement `poly_eval` using Horner's rule.

☐ Optionally implement polynomial addition and multiplication (for later Chebyshev work).

Experiment Tasks:

☐ Choose test polynomials (e.g., Chebyshev, Taylor expansions).

☐ Evaluate them at many points and compare:

- Direct evaluation vs. Horner's rule.
- Error vs. degree when approximating known functions.

Deliverables:

- `core/poly.h`, `core/poly.cpp`.
- Plots or tables of error vs. degree for at least one approximation problem.

Tags: `Module:core-poly`, `Reading`, `Implementation`, `Experiment`

Phase 2 – Interpolation & Approximation

Phase 2 Overview – Interpolation & Approximation

Goal: Build interpolation and approximation tools that will feed into differentiation, integration, and ODE/PDE solvers.

Primary Sources:

- TAOCP Vol. 2, sections 4.6 and 4.7 (polynomials and power series).
- Numerical Recipes, Chapter 3 (interpolation and extrapolation), and sections 5.8–5.10 (Chebyshev approximation).

Module Tie-ins:

- `interp::` (interpolation).
- `core::poly` (polynomial arithmetic).

Exit Criteria:

- You can construct interpolation schemes and Chebyshev approximations.
- You can compare errors for different node distributions and degrees.

Phase 2 Card 1 – Read Interpolation Basics

Description: Understand the basic interpolation methods and their properties.

Reading Tasks:

☐ Numerical Recipes, Chapter 3, sections 3.0–3.3:

- Table lookup, polynomial interpolation, cubic splines.

☐ Skim sections on multidimensional interpolation (3.6–3.7) for later.

Notes Deliverable:

- Short notes summarizing:
  - When polynomial interpolation is appropriate.
  - Why splines can be more stable.
  - Dangers of extrapolation.

Tags: `Reading`, `Book:NR`, `Module:interp`

Phase 2 Card 2 – Implement 1D Interpolation Module

Description: Implement core 1D interpolation routines.

Implementation Tasks:

☐ Implement polynomial interpolation:

- Given arrays `x[i]`, `y[i]`, construct coefficients or direct evaluation routines.

☐ Implement cubic splines:

- Natural or clamped boundary conditions.
- Precompute spline coefficients; expose an evaluation function.

☐ Design a clean API:

- `interp::poly(x, y)` returning an object with an `operator()`.
- `interp::spline(x, y)` similarly.

Testing & Experiments:

☐ Test on smooth functions (e.g., $\sin x$, $\exp x$) and less smooth ones.

☐ Compare interpolation error for different numbers of nodes.

Deliverables:

- `interp/interp.h`, `interp/interp.cpp`.
- Example program visualizing interpolated curves vs. ground truth.

Tags: `Module:interp`, `Implementation`, `Experiment`

Phase 2 Card 3 – Implement Chebyshev Approximation

Description: Implement Chebyshev-based function approximation and connect it to polynomial arithmetic.

Reading Tasks:

☐ Numerical Recipes, sections 5.8–5.10:

- Chebyshev approximation, derivatives, and integrals via coefficients.

☐ Revisit TAOCP Vol. 2, section 4.6 for efficient polynomial evaluation and composition.

Implementation Tasks:

☐ Implement routines to compute Chebyshev coefficients of a function on $[-1, 1]$.

☐ Implement evaluation of the Chebyshev approximation at arbitrary points.

☐ Provide helpers to differentiate/integrate the Chebyshev approximation.

Experiment Tasks:

☐ Approximate several functions (e.g., $\sin x$, $\exp x$, a Gaussian).

☐ Compare approximation error vs. degree for Chebyshev vs. naive polynomial interpolation.

Deliverables:

- `interp/chebyshev.h`, `interp/chebyshev.cpp`.
- Plots of error vs. polynomial degree and node distributions.

Tags: `Module:interp`, `Module:core-poly`, `Implementation`, `Experiment`

Phase 2 Card 4 – Experiment: Interpolation vs Degree & Nodes

Description: Dedicated experiments on the behavior of interpolation as degree and node distribution change.

Experiment Tasks:

☐ Choose several test functions:

- Smooth ($\sin x$, $\exp x$).
- With endpoint features or mild singularities.

☐ Compare:

- Equally spaced nodes vs. Chebyshev nodes.
- Polynomial interpolation vs. splines vs. Chebyshev approximation.

☐ Produce plots of:

- Max error vs. degree.
- Error distribution across the interval.

Deliverables:

- An `examples/interp_study.cpp` (or similar) program.
- A short write-up summarizing observations (Runge phenomenon, stability).

Tags: `Module:interp`, `Experiment`, `Analysis`

Phase 3 – Numerical Differentiation & Integration (Quadrature)

Phase 3 Overview – Numerical Differentiation & Integration

Goal: Directly attack core calculus tasks: derivative estimation and definite integrals.

Primary Sources:

- Numerical Recipes:
  - Section 5.7: numerical derivatives and error behavior.
  - Chapter 4: integration of functions (classical formulas, Romberg, Gaussian, adaptive, multidimensional).

Module Tie-ins:

- `diffint::deriv` (differentiation).
- `diffint::quad` (quadrature).

Exit Criteria:

- You have derivative estimators and quadrature routines with observed convergence rates matching theory.
- You can visualize error vs. step size or number of function evaluations.

Phase 3 Card 1 – Read Numerical Differentiation & Error Behavior

Description: Build conceptual understanding of finite differences and their errors.

Reading Tasks:

☐ Numerical Recipes, section 5.7:

- Forward, backward, and central differences.
- Step-size tradeoffs between truncation and round-off error.

Notes Deliverable:

- Short notes or a diagram showing:
  - Taylor-expansion-based derivation of finite-difference formulas.
  - Qualitative shape of error vs. step size (U-shaped curve).

Tags: `Reading`, `Book:NR`, `Module:diffint-deriv`

Phase 3 Card 2 – Implement Derivative Estimators

Description: Implement a small `diffint::deriv` module.

Implementation Tasks:

☐ Implement:

- `deriv_forward(f, x, h)`.
- `deriv_backward(f, x, h)`.
- `deriv_central(f, x, h)`.

☐ Optionally add Richardson extrapolation to boost accuracy.

☐ Use `core::fp` utilities for error comparisons.

Experiment Tasks:

☐ For known functions (e.g., $\sin x$, $\exp x$, polynomials):

- Sweep over step sizes h.
- Plot absolute error vs. h for each scheme.
- Show the "sweet spot" where truncation and round-off balance.

Deliverables:

- `diffint/deriv.h`, `diffint/deriv.cpp`.
- Example program producing error plots.

Tags: `Module:diffint-deriv`, `Implementation`, `Experiment`

Phase 3 Card 3 – Read Integration Methods Overview

Description: Survey classical quadrature methods.

Reading Tasks:

☐ Numerical Recipes, sections 4.0–4.4:

- Trapezoidal rule, Simpson's rule, Romberg integration.
- Basics of improper integrals.

Notes Deliverable:

- A small table summarizing:
  - Order of accuracy for each rule.
  - When each method is appropriate or problematic.

Tags: `Reading`, `Book:NR`, `Module:diffint-quad`

Phase 3 Card 4 – Implement Basic 1D Quadrature

Description: Implement basic 1D integration routines.

Implementation Tasks:

☐ Implement composite trapezoidal rule: `trap(f, a, b, n)`.

☐ Implement composite Simpson's rule: `simpson(f, a, b, n)`.

☐ Implement Romberg integration `romberg(f, a, b)` using trapezoidal estimates.

Experiment Tasks:

☐ Integrate known functions (polynomials, $\sin x$, $\exp x$):

- Compare numerical results to analytic integrals.
- Plot error vs. number of function evaluations.

Deliverables:

- `diffint/quad.h`, `diffint/quad.cpp`.
- Example program for integration convergence plots.

Tags: `Module:diffint-quad`, `Implementation`, `Experiment`

Phase 3 Card 5 – Implement Advanced Quadrature & Convergence Plots

Description: Extend the integration module with more advanced schemes and compare their performance.

Implementation Tasks:

☐ Implement Gaussian quadrature:

- `gauss_legendre(f, a, b, n)`.

☐ Implement an adaptive integrator:

- Subdivide intervals based on local error estimates.

☐ Optionally add multidimensional integration wrappers (tensor product or simple Monte Carlo).

Experiment Tasks:

☐ For several test integrals:

- Compare error vs. function evaluations for trapezoid, Simpson, Romberg, Gaussian, and adaptive methods.
- Summarize which method wins for which class of functions.

Deliverables:

- Extended `diffint/quad.*` with advanced methods.
- Plots or tables of convergence behavior.

Tags: `Module:diffint-quad`, `Implementation`, `Experiment`, `Analysis`

Phase 4 – Root Finding & Nonlinear Equations

Phase 4 Overview – Root Finding & Nonlinear Equations

Goal: Solve $f(x) = 0$ and small nonlinear systems using robust algorithms.

Primary Sources:

- Numerical Recipes, Chapter 9:
  - Bracketing methods, secant, Brent, Newton, and nonlinear systems.

Module Tie-ins:

- `nonlin::root` (root finding).
- `linalg::` (linear algebra for systems).
- `diffint::deriv` (for Newton's method derivatives).

Exit Criteria:

- You have a generic 1D root-finding API with multiple methods.
- You can solve small nonlinear systems using a Newton-style method.

Phase 4 Card 1 – Read Root-Finding Landscape

Description: Understand the tradeoffs between different root-finding methods.

Reading Tasks:

☐ Numerical Recipes, sections 9.0–9.4:

- Bracketing methods (bisection).
- Secant and Brent methods.
- Newton's method and its convergence properties.

Notes Deliverable:

- A small table summarizing:
  - Method type (bracketing vs. open).
  - Convergence rate.
  - Requirements (derivative? initial bracket?).

Tags: `Reading`, `Book:NR`, `Module:nonlin-root`

Phase 4 Card 2 – Implement 1D Root-Finding API

Description: Implement a small `nonlin::root` module with multiple strategies.

Implementation Tasks:

☐ Implement:

- `bisect(f, a, b, tol)`.
- `newton(f, df, x0, tol)`.
- `brent(f, a, b, tol)`.

☐ Provide a unified wrapper:

- `root_find(f, ...)` that selects method based on configuration.

☐ Use finite differences from `diffint::deriv` when analytic derivatives are not available.

Experiment Tasks:

☐ Test on several functions with known roots:

- Polynomials of moderate degree.
- Transcendental equations ($\sin x = x/2$, etc.).

☐ Measure iterations and robustness for different starting guesses.

Deliverables:

- `nonlin/root.h`, `nonlin/root.cpp`.
- Example programs illustrating method comparisons.

Tags: `Module:nonlin-root`, `Implementation`, `Experiment`

Phase 4 Card 3 – Implement Nonlinear System Solvers

Description: Solve small systems $F(\mathbf{x}) = 0$ using Newton-type methods.

Implementation Tasks:

☐ Implement Jacobian approximation via finite differences:

- Wrap `diffint::deriv` for multivariate functions.

☐ Implement a Newton system solver:

- At each iteration, solve $J(\mathbf{x}_k)\Delta\mathbf{x} = -F(\mathbf{x}_k)$ using `linalg::`.

☐ Implement simple globalisation strategies:

- Damped steps or basic line search to improve robustness.

Experiment Tasks:

☐ Test on simple nonlinear systems (e.g., intersections of circles, small mechanical systems).

☐ Track convergence behavior for different initial guesses.

Deliverables:

- `nonlin/system.h`, `nonlin/system.cpp`.
- Example programs showing convergence and failure cases.

Tags: `Module:nonlin-root`, `Module:linalg`, `Implementation`, `Experiment`

Phase 5 – Linear Algebra & Optimization

Phase 5 Overview – Linear Algebra & Optimization

Goal: Build the linear algebra backbone and basic optimization algorithms used by many calculus applications (ODE, PDE, fitting).

Primary Sources:

- Numerical Recipes:
  - Chapter 2: solution of linear algebraic equations.
  - Chapter 10: minimization or maximization of functions.

Module Tie-ins:

- `linalg::` (linear algebra).
- `optim::` (optimization).

Exit Criteria:

- You can solve small dense systems and simple least-squares problems.
- You can perform basic 1D and multidimensional optimization.

Phase 5 Card 1 – Read Linear System Solvers

Description: Understand practical algorithms for solving $A\mathbf{x} = \mathbf{b}$.

Reading Tasks:

☐ Numerical Recipes, sections 2.0–2.4:

- LU decomposition and Gaussian elimination.
- Banded systems and conditioning notes.

Notes Deliverable:

- Short summary of:
  - When LU is appropriate vs. Cholesky.
  - The role of pivoting and conditioning.

Tags: `Reading`, `Book:NR`, `Module:linalg`

Phase 5 Card 2 – Implement Linear Algebra Core

Description: Implement key linear algebra routines.

Implementation Tasks:

☐ Implement a small dense matrix and vector type, or wrap an existing library with a thin façade.

☐ Implement LU decomposition with partial pivoting.

☐ Implement a solver using LU factors for multiple right-hand sides.

☐ Implement Cholesky decomposition for symmetric positive-definite matrices.

Experiment Tasks:

☐ Test on small random systems and known test matrices.

☐ Compare residuals for different methods and confirm numerical stability.

Deliverables:

- `linalg/linalg.h`, `linalg/linalg.cpp`.
- Unit tests for LU and Cholesky solvers.

Tags: `Module:linalg`, `Implementation`, `Experiment`

Phase 5 Card 3 – Read Scalar & Multidimensional Optimization

Description: Review core ideas in 1D and multidimensional optimization.

Reading Tasks:

☐ Numerical Recipes, sections 10.0–10.4:

- 1D line searches.
- Simple multidimensional methods.

Notes Deliverable:

- Summary of:
  - Golden-section search and Brent-like methods.
  - Basic gradient-based methods and convergence criteria.

Tags: `Reading`, `Book:NR`, `Module:optim`

Phase 5 Card 4 – Implement Optimization Routines

Description: Implement basic 1D and multidimensional optimization algorithms.

Implementation Tasks:

☐ 1D minimization:

- Golden-section search.
- Brent-type method for robustness.

☐ Multidimensional:

- Gradient descent with line search.
- Optionally conjugate gradient or quasi-Newton variant.

Experiment Tasks:

☐ Run on benchmark functions:

- Quadratic bowls.
- Rosenbrock function.

☐ Track iteration counts, convergence behavior, and sensitivity to starting guess.

Deliverables:

- `optim/optim.h`, `optim/optim.cpp`.
- Example programs visualizing convergence trajectories (e.g., contour plots with paths).

Tags: `Module:optim`, `Implementation`, `Experiment`

Phase 6 – ODEs & PDEs (Dynamics & Fields)

Phase 6 Overview – ODEs & PDEs

Goal: Use everything above to solve dynamical systems (ODEs) and simple PDEs.

Primary Sources:

- Numerical Recipes:
  - Chapter 17: integration of ODEs.
  - Chapters 18 and 20: boundary value problems and PDEs (for stretch goals).

Module Tie-ins:

- `ode::ivp`, `ode::bvp`.
- `pde::` (for simple finite-difference PDE solvers).

Exit Criteria:

- You can integrate basic ODEs with fixed and adaptive step sizes.
- You have at least one simple PDE (e.g., 1D heat equation) implemented and visualized.

Phase 6 Card 1 – Read ODE IVP Basics

Description: Understand explicit methods and adaptive step-size control.

Reading Tasks:

☐ Numerical Recipes, sections 17.0–17.2:

- Runge–Kutta methods.
- Step-size control strategies.

Notes Deliverable:

- Short summary of:
  - Local vs. global truncation error.
  - Stability considerations for explicit methods.

Tags: `Reading`, `Book:NR`, `Module:ode`

Phase 6 Card 2 – Implement ODE IVP Solvers

Description: Implement core IVP solvers and test them on classic examples.

Implementation Tasks:

☐ Implement:

- Explicit Euler step.
- RK4 step.
- Simple adaptive RK method (e.g., embedded RK pair).

☐ Design an API:

- `ode::solve_ivp(f, t0, y0, t_end, h_initial, options)` returning sampled trajectory.

Experiment Tasks:

☐ Test on:

- Exponential decay.
- Harmonic oscillator.
- Simple nonlinear systems.

☐ Plot solution trajectories and compare against analytic solutions where available.

Deliverables:

- `ode/ivp.h`, `ode/ivp.cpp`.
- Example plots of ODE solutions.

Tags: `Module:ode`, `Implementation`, `Experiment`

Phase 6 Card 3 – Read Stiff ODEs & Multistep Methods

Description: Understand the issues around stiffness and multistep approaches.

Reading Tasks:

☐ Numerical Recipes, sections 17.5–17.6:

- Stiff ODEs.
- Multistep methods.

Notes Deliverable:

- Short summary of:
  - What stiffness is and why explicit methods fail.
  - Basic idea of implicit methods and stability regions.

Tags: `Reading`, `Book:NR`, `Module:ode`

Phase 6 Card 4 – Implement Simple Stiff ODE Solver

Description: Implement a basic implicit scheme using your linear algebra module.

Implementation Tasks:

☐ Implement implicit Euler step:

  – Use Newton iteration to solve the implicit equation at each step.

☐ Reuse `linalg::` and `nonlin::` modules.

Experiment Tasks:

☐ Test on stiff problems (e.g., simple reaction equations).

☐ Compare explicit vs. implicit schemes and illustrate instability vs. stability.

Deliverables:

- Extension of `ode/ivp.*` or a separate `ode/stiff.*`.
- Plots showing behavior on stiff vs. non-stiff examples.

Tags: `Module:ode`, `Module:linalg`, `Module:nonlin-root`, `Implementation`, `Experiment`

Phase 6 Card 5 – Read PDE Overview & Implement 1D Heat Equation

Description: Take a first step into PDEs with a finite-difference 1D heat equation.

Reading Tasks:

☐ Numerical Recipes, sections 20.0–20.2:

- Flux-conservative and diffusive PDEs.

Implementation Tasks:

☐ Discretize the 1D heat equation with:

- Explicit scheme (forward Euler in time, central differences in space).
- Implicit scheme if time allows.

☐ Use your linear algebra module for the implicit solves.

Experiment Tasks:

☐ Visualize temperature distribution over time.

☐ Compare stability and accuracy of explicit vs. implicit schemes.

Deliverables:

- `pde/heat1d.h`, `pde/heat1d.cpp`.
- Plots or animations of heat diffusion.

Tags: `Module:pde`, `Module:linalg`, `Implementation`, `Experiment`

Phase 7 – Random Numbers & Monte Carlo

Phase 7 Overview – Random Numbers & Monte Carlo

Goal: Use random numbers to drive Monte Carlo integration and stochastic simulations.

Primary Sources:

- TAOCP Vol. 2, Chapter 3: random numbers.
- Numerical Recipes, Chapter 7: random numbers and Monte Carlo.

Module Tie-ins:

- `rand::core`.
- `diffint::quad` (Monte Carlo as another integration approach).

Exit Criteria:

- You have a solid uniform RNG and basic distribution helpers.
- You can implement and compare Monte Carlo integration with deterministic quadrature.

Phase 7 Card 1 – Read PRNG Theory & Quality

Description: Understand the theoretical background for random number generation.

Reading Tasks:

☐ TAOCP Vol. 2, sections 3.1–3.3:

- Linear congruential generators and other PRNGs.
- Statistical tests and randomness criteria.

☐ Numerical Recipes, sections 7.0–7.4:

- Practical PRNGs and basic tests.

Notes Deliverable:

- Brief notes on:
  - Why naive generators fail.
  - Key properties of good RNGs (period, equidistribution).

Tags: `Reading`, `Book:TAOCP2`, `Book:NR`, `Module:rand`

Phase 7 Card 2 – Implement Core RNG Module

Description: Implement uniform PRNGs and common distributions.

Implementation Tasks:

☐ Implement a high-quality uniform RNG (or wrap the C++ standard library in a controlled way).

☐ Provide:

- `rand::uniform()`, `rand::normal()`, `rand::exponential()`, etc.

☐ Support seeding and independent streams.

Experiment Tasks:

☐ Perform basic statistical checks:

- Histograms for various distributions.
- Simple correlation and autocorrelation checks.

Deliverables:

- `rand/rand.h`, `rand/rand.cpp`.
- Small test programs visualizing distribution histograms.

Tags: `Module:rand`, `Implementation`, `Experiment`

Phase 7 Card 3 – Implement Monte Carlo Integrators

Description: Use your RNG to implement Monte Carlo integration.

Implementation Tasks:

☐ Implement simple Monte Carlo integrators:

- `mc_integrate(f, domain, n_samples)`.

☐ Optionally add quasi-random sequences (e.g., Sobol or Halton).

Experiment Tasks:

☐ Compare Monte Carlo vs. deterministic quadrature from Phase 3:

- Error vs. number of samples.
- When Monte Carlo is preferable (e.g., higher-dimensional integrals).

Deliverables:

- `diffint/mc_quad.h`, `diffint/mc_quad.cpp` or integrated into `diffint::quad`.
- Plots comparing convergence rates.

Tags: `Module:rand`, `Module:diffint-quad`, `Implementation`, `Experiment`

Phase 8 – Capstone Projects

Phase 8 Overview – Capstone Pipelines

Goal: Execute small end-to-end projects that exercise multiple modules at once.

Capstone Themes:

- Nonlinear boundary-value ODE problems.
- PDE-based models coupled with optimization.

Exit Criteria:

- At least one nontrivial project is fully implemented, tested, and documented.
- You can explain how each module participates in the pipeline.

Phase 8 Card 1 – Capstone: Nonlinear Boundary-Value ODE

Description: Couple root finding, quadrature, and ODE solvers to handle a nonlinear BVP.

Pipeline Sketch:

- Model a simple physical or mathematical system (e.g., nonlinear beam, shooting method BVP).
- Use:
  - `ode::ivp` for shooting.
  - `nonlin::root` for matching boundary conditions.
  - `diffint::quad` if necessary for quantities defined by integrals.

Tasks:

☐ Define the differential equation and boundary conditions.

☐ Implement a shooting method:

- Convert BVP to IVP.
- Use root finding to adjust initial conditions.

☐ Validate against known or reference solutions if available.

Deliverables:

- An `examples/bvp_capstone.cpp` (or similar) program.
- A short write-up (1–2 pages) explaining model, methods, and results.

Tags: `Module:ode`, `Module:nonlin-root`, `Module:diffint-quad`, `Capstone`

Phase 8 Card 2 – Capstone: PDE + Optimization

Description: Combine PDE solving with parameter optimization.

Pipeline Sketch:

- Choose a simple steady-state PDE model (e.g., Laplace or Poisson).
- Introduce a parameter to be fitted (e.g., diffusion coefficient, source term amplitude).
- Use:
  - `pde::` for discretization and solution.
  - `optim::` to fit the parameter to synthetic or real data.
  - `stats::` (if implemented) to evaluate goodness of fit.

Tasks:

☐ Implement the PDE solver for the chosen model.

☐ Define an objective function measuring mismatch between solution and data.

☐ Optimize the parameter using methods from `optim::`.

☐ Analyze sensitivity to noise and initial guesses.

Deliverables:

- An `examples/pde_optim_capstone.cpp` (or similar) program.
- A brief report summarizing model, numerical methods, and optimization results.

Tags: `Module:pde`, `Module:optim`, `Module:stats`, `Capstone`

Module Reference (Quick Mapping)

Module `core::fp` – Floating-Point & Basic Arithmetic

Responsibilities:

- Provide floating-point utilities: `ulp`, `nearly_equal`.
- Robust summation and dot-product routines.
- Error metrics: absolute, relative, norms.

Book Mapping:

- TAOCP Vol. 2, Chapter 4 (arithmetic, floating-point).
- Numerical Recipes, Chapter 1 (error, accuracy, stability).

Module `core::poly` – Polynomial & Power-Series Arithmetic

Responsibilities:

- Represent polynomials and power series.
- Implement Horner evaluation, addition, multiplication, composition.
- Support Chebyshev and other polynomial-based approximations.

Book Mapping:

- TAOCP Vol. 2, sections 4.6–4.7 (polynomials, power series).
- Numerical Recipes, section 5.1 and 5.8–5.12 (polynomials, Chebyshev methods).

Module `rand::core` – Random Numbers & Distributions

Responsibilities:

- High-quality uniform PRNGs.
- Transformations to normal, exponential, and other distributions.
- Quasi-random sequences for Monte Carlo.

Book Mapping:

- TAOCP Vol. 2, Chapter 3 (random numbers).
- Numerical Recipes, Chapter 7 (random numbers and Monte Carlo).

Module `interp::` – Interpolation & Approximation

Responsibilities:

- Table lookup and interpolation in 1D (and later multidim).
- Polynomial, rational, spline, and Chebyshev interpolation.

Book Mapping:

- TAOCP Vol. 2, sections 4.6–4.7 (as algebraic backbone).
- Numerical Recipes, Chapter 3 and sections 5.8–5.10 (interpolation, Chebyshev).

Module `diffint::deriv` – Numerical Differentiation

Responsibilities:

- Finite-difference derivatives (forward, backward, central, higher order).
- Step-size selection and error modeling.
- Optional Richardson extrapolation.

Book Mapping:

- Numerical Recipes, section 5.7 (numerical derivatives).

Module `diffint::quad` – Numerical Integration & Monte Carlo

Responsibilities:

- 1D quadrature: trapezoid, Simpson, Romberg.
- Gaussian and adaptive quadrature.
- Multidimensional and Monte Carlo integration (ties to `rand::core`).

Book Mapping:

- Numerical Recipes, Chapter 4 (integration of functions).
- Numerical Recipes, sections 7.7–7.9 (Monte Carlo integration).

Module `linalg::` – Linear Algebra Core

Responsibilities:

- Dense linear systems: LU, Cholesky, possibly QR/SVD.
- Banded and special matrices.
- Supporting routines for eigenproblems as needed.

Book Mapping:

- Numerical Recipes, Chapter 2 (linear equations).
- Numerical Recipes, Chapter 11 (eigensystems), if used.

Module `nonlin::root` – Root Finding & Nonlinear Systems

Responsibilities:

- 1D root solvers: bisection, secant, Brent, Newton.
- Nonlinear systems via Newton and globalized variants.

Book Mapping:

- Numerical Recipes, Chapter 9 (root finding and nonlinear systems).

Module `optim::` – Optimization & Minimization

Responsibilities:

- 1D minimization: golden-section, Brent.
- Multidimensional unconstrained optimization (gradient-based or simplex methods).

Book Mapping:

- Numerical Recipes, Chapter 10 (minimization or maximization of functions).

Module `spectral::` – FFT & Spectral Tools

Responsibilities:

- FFT and inverse FFT.
- Spectral differentiation/integration.
- Convolution and filtering primitives.

Book Mapping:

- Numerical Recipes, Chapters 12–13 (FFT and spectral applications).

Module `ode::ivp`, `ode::bvp` – Ordinary Differential Equations

Responsibilities:

- IVP solvers: explicit/implicit, adaptive, stiff and non-stiff.
- BVP solvers: shooting methods, relaxation methods.

Book Mapping:

- Numerical Recipes, Chapters 17–18 (ODEs and BVPs).

Module `pde::` – Partial Differential Equations

Responsibilities:

- Finite-difference solvers for simple PDEs (1D/2D).
- Explicit/implicit schemes, stability helpers, and boundary conditions.

Book Mapping:

- Numerical Recipes, Chapter 20 (PDEs).

Module `stats::` – Statistical Tools & Error Modeling

Responsibilities:

- Descriptive statistics and correlation.
- Least squares and error/convergence analysis for experiments.

Book Mapping:

- Numerical Recipes, Chapters 14–15 (statistics and data modeling).