Tools for Numerical Computing

Brian Howell

IN PROGRESS

September 20, 2023

Introduction:

I had the fortunate opportunity to pursue my graduate studies at UC Berkeley under the direction of Tarek Zohdi. Throughout my time, I was exposed to a handful of useful tools from applied mathematics, machine learning, optimization, and high performance computing. This document is an attempt to transcribe and compile my notes from a 5-year period into a reference guide for myself.

Disclaimer:

The intended audience was myself. That being said, I have shared sections of this document with colleagues and friends who claimed it was helpful in their studies. Maybe it will be helpful to you.

This is a live document that I intend to continue to update, which means there are likely to be some errors. If you find mistakes and feel like telling me, I will be grateful and happy to hear from you, even for the most trivial of errors. You can reach me by email at bhowell@berkeley.edu.

Contents

Section 1: Recommended Readings

1.1 Mathematics

Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems

This book introduces finite difference methods for both ordinary differential equations (ODEs) and partial differential equations (PDEs) and discusses the similarities and differences between algorithm design and stability analysis for different types of equations. A unified view of stability theory for ODEs and PDEs is presented, and the interplay between ODE and PDE analysis is stressed. The text emphasizes standard classical methods, but several newer approaches also are introduced and are described in the context of simple motivating examples. Exercises and student projects are available on the book's webpage, along with Matlab mfiles for implementing methods. Readers will gain an understanding of the essential ideas that underlie the development, analysis, and practical use of finite difference methods as well as the key concepts of stability theory, their relation to one another, and their practical implications. The author provides a foundation from which students can approach more advanced topics.

A Finite Element Primer

The approach used herein is to introduce the basic concepts first in one dimension, then move on to three dimensions. A relatively informal style is adopted in this guide. These notes are intended as a "starting point," which can be later augmented by the large array of rigorous, detailed books in the area of Finite Element analysis. Through teaching finite element classes for a number of years at UC Berkeley, my experience has been that the fundamental mathematics, such as vector calculus, linear algebra, and basic mechanics, exemplified by linearized elasticity, cause conceptual problems that impede the understanding of the Finite Element Method. Thus, appendices on these topics have been included. Finally, I am certain that, despite painstaking efforts, there remain errors of one sort or another. Therefore, I would be grateful if readers who find such flaws would contact me at zohdi@berkeley.edu.

1.2 Optimization

Optimization Models:

Emphasizing practical understanding over the technicalities of specific algorithms, this elegant textbook is an accessible introduction to the field of optimization, focusing on powerful and reliable convex optimization techniques. Students and practitioners will learn how to recognize, simplify, model and solve optimization problems - and apply these principles to their own projects. A clear and self-contained introduction to linear algebra demonstrates core mathematical concepts in a way that is easy to follow, and helps students to understand their practical relevance. Requiring only a basic understanding of geometry, calculus, probability and statistics, and striking a careful balance between accessibility and rigor, it enables students to quickly understand the material, without being overwhelmed by complex mathematics. Accompanied by numerous end-of-chapter problems, an online solutions manual for instructors, and relevant examples from diverse fields including engineering, data science, economics, finance, and management, this is the perfect introduction to optimization for undergraduate and graduate students.

Convex Optimization

Convex optimization problems arise frequently in many different fields. A comprehensive introduction to the subject, this book shows in detail how such problems can be solved numerically with great efficiency. The focus is on recognizing convex optimization problems and then finding the most appropriate technique for solving them. The text contains many worked examples and homework exercises and will appeal to students, researchers and practitioners in fields such as engineering, computer science, mathematics, statistics, finance, and economics.

Algorithms for Optimization

This book offers a comprehensive introduction to optimization with a focus on practical algorithms. The book approaches optimization from an engineering perspective, where the objective is to design a system that optimizes a set of metrics subject to constraints. Readers will learn about computational approaches for a range of challenges, including searching high-dimensional spaces, handling problems where there are multiple competing objectives, and accommodating uncertainty in the metrics. Figures, examples, and exercises convey the intuition behind the mathematical approaches. The text provides concrete implementations in the Julia programming language.

1.3 Machine Learning and Data Science

Gaussian Processes for Machine Learning

Gaussian processes (GPs) provide a principled, practical, probabilistic approach to learning in kernel machines. GPs have received increased attention in the machine-learning community over the past decade, and this book provides a long-needed systematic and unified treatment of theoretical and practical aspects of GPs in machine learning. The treatment is comprehensive and self-contained, targeted at researchers and students in machine learning and applied statistics. The book deals with the supervised-learning problem for both regression and classification, and includes detailed algorithms. A wide variety of covariance (kernel) functions are presented and their properties discussed. Model selection is discussed both

from a Bayesian and a classical perspective. Many connections to other well-known techniques from machine learning and statistics are discussed, including support-vector machines, neural networks, splines, regularization networks, relevance vector machines and others. Theoretical issues including learning curves and the PAC-Bayesian framework are treated, and several approximation methods for learning with large datasets are discussed. The book contains illustrative examples and exercises, and code and datasets are available on the Web. Appendixes provide mathematical background and a discussion of Gaussian Markov processes.

Decision Making Under Uncertainty Theory and Application

Many important problems involve decision making under uncertainty—that is, choosing actions based on often imperfect observations, with unknown outcomes. Designers of automated decision support systems must take into account the various sources of uncertainty while balancing the multiple objectives of the system. This book provides an introduction to the challenges of decision making under uncertainty from a computational perspective. It presents both the theory behind decision making models and algorithms and a collection of example applications that range from speech recognition to aircraft collision avoidance.

Section 2: Maths

2.1 Linear Algebra

2.1.1 Definitions

Definition 2.1: Null space

The null space of a matrix $\mathbf{A}_{m\times n}$, Null($\mathbf{A}$), is the set of vectors whose multiplication with $\mathbf{A}$ results in the zero vector.

$$\{\mathbf{x} \in \mathbb{R}^n | \mathbf{A}\mathbf{x} = \mathbf{0}\} \tag{1}$$

Definition 2.2: Column space

The column space of a matrix $\mathbf{A}_{m\times n}$, Col($\mathbf{A}$), is the set of all possible vectors $\mathbf{b}$ that result from multiplying $\mathbf{A}$ by any vector $\mathbf{x} \in \mathbb{R}^n$.

$$\{\mathbf{b} \in \mathbb{R}^m | \mathbf{A}\mathbf{x} = \mathbf{b}\} \tag{2}$$

2.2 Calculus

Credit: rdhuff@berkeley.edu

2.2.1 Fundamentals

2.2.2 Einstein summation notation

Einstein summation notation is a shorthand way of writing mathematical expressions that involve repeated summations or products over indices. It is widely used in fields such as physics and engineering to write complex equations in a more concise and elegant form.

In Einstein summation notation, repeated indices are automatically summed over, unless they appear once in a subscript and once in a superscript. This convention is known as the Einstein summation convention. The indices are typically Greek letters for time indices, and Latin letters for spatial indices.

In this notation, scalars, vectors, and matrices are represented as follows:

- Scalars are represented by a single symbol, such as a or f.
- Vectors are represented by lowercase Latin letters with indices, such as v_i or u_j. The indices can take on values of 1, 2, ..., n, where n is the dimension of the vector.
- Matrices are represented by uppercase Latin letters with two indices, such as A_{ij} or B_{kl}. The first index refers to the row number and the second index refers to the column number.

Using this notation, we can write linear algebra equations more compactly and with less clutter. For example, the dot product of two vectors $\mathbf{u}$ and $\mathbf{v}$ can be written as:

$$\mathbf{u} \cdot \mathbf{v} = u_i v_i \tag{3}$$

Similarly, the matrix-vector product $\mathbf{y} = A\mathbf{x}$ can be written as:

$$y_i = A_{ij} x_j \tag{4}$$

Einstein summation notation can also be used for expressing more complex operations such as matrix multiplication, transpose, and inverse.

Kronecker Delta:

$$\delta_{ij} = \mathbf{e}_i \cdot \mathbf{e}_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{5}$$

Dot Product:

$$\mathbf{u} \cdot \mathbf{v} = (u_i \mathbf{e}_i) \cdot (v_j \mathbf{e}_j) = u_i v_j \mathbf{e}_i \cdot \mathbf{e}_j = u_i v_j \delta_{ij} = u_i v_i \tag{6}$$

Alternating (Epsilon or Levi-Civita) Symbol:

$$\varepsilon_{ijk} = \begin{cases} 1 & i,j,k \text{ is an even permutation} \\ 0 & i = j \text{ or } i = k \text{ or } j = k \\ -1 & i,j,k \text{ is an odd permutation} \end{cases} \tag{7}$$

Cross Product:

$$u \times v = \varepsilon_{ijk} u_j v_k e_i \tag{8}$$

$$(u \times v)i = \varepsilon ijk u_j v_k \tag{9}$$

Relationship between the Alternating Symbol and the Kronecker Delta:

$$\varepsilon_{ijk} \varepsilon_{ilm} = \delta_{jl} \delta_{km} - \delta_{jm} \delta_{kl} \tag{10}$$

Tensor (Dyadic) Product:

$$(a \otimes b)u = a(b \cdot u) \tag{11}$$
$$(a \otimes b)_{ij} = a_i b_j \tag{12}$$
$$(a \otimes b)(a \otimes b) = a(b \cdot a) \otimes b \tag{13}$$
$$(a \otimes b) = (a_i e_i) \otimes (b_j e_j) = a_i b_j e_i \times e_j \tag{14}$$
$$I = \delta_{ij} e_i \otimes e_j = e_i \otimes e_i \tag{15}$$

Gradient of Scalar Field: The gradient of a scalar field is a vector that points in the direction of the greatest increase of the function at a given point. It is denoted by the symbol ∇.

In Cartesian coordinates, the gradient of a scalar function $f(x, y, z)$ is defined as:

$$\nabla f = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}, \tag{16}$$

where $\mathbf{i}, \mathbf{j}$, and $\mathbf{k}$ are the unit vectors along the x, y, and z axes, respectively.

More generally, the gradient of a scalar field $f(\mathbf{x})$ in n-dimensional space can be written as:

$$\nabla f = \frac{\partial f}{\partial x_1}\mathbf{e_1} + \frac{\partial f}{\partial x_2}\mathbf{e_2} + \cdots + \frac{\partial f}{\partial x_n}\mathbf{e_n}, \tag{17}$$

where $\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$ are the unit vectors along the n coordinate axes.

By convention, if we wish to represent the gradient of a scalar function $f(\mathbf{x})$ as a vector in direct notation, we write it as a column vector:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{18}$$

This makes sense because the gradient is a vector quantity, and vectors are typically represented as column vectors.

Example. Let's consider the scalar function $f(\mathbf{x}) = \mathbf{x}^T A \mathbf{x}$, where $A \in \mathbb{R}^{n \times n}$. The gradient of this function with respect to $\mathbf{x}$ is:

$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial x_i} e_i \tag{19}$$
$$= \frac{\partial}{\partial x_i}(x_j a_{jk} x_k) e_i \tag{20}$$
$$= (2a_{ij}x_j)e_i + (a_{jk}x_k)\frac{\partial x_j}{\partial x_i} e_i \tag{21}$$
$$= (A + A^T)_{ij} x_j e_i, \tag{22}$$

where we used the product rule for partial differentiation and the fact that A is symmetric. The expression $(A+A^T)_{ij}$ represents the (i, j)-th entry of the matrix $A + A^T$.

Gradient of Vector Field:

$$\nabla v = \frac{\partial v_i}{\partial x_j} e_i \otimes e_j \tag{23}$$

2.3 Differential Equations

2.3.1 Parabolic Partial Differential Equations

Parabolic partial differential equations (PDEs) play a fundamental role in modeling various phenomena, ranging *from black holes to Black-Scholes*. More specifically, the form of parabolic PDEs elegantly captures the dynamics of a large range processes such as heat and chemical diffusion, Schrödinger's equation, and option pricing in financial mathematics. One of the most commonly used numerical methods for solving such equations is the finite difference method (FDM). FDM is a simple, efficient, and robust technique that discretizes the PDE on a grid and approximates the derivatives using finite differences [**?**]. This method has been extensively studied and applied in various fields of science and engineering for over a century. In this section of the dissertation, I discuss the theoretical foundations and practical implementation of FDM for parabolic PDEs.

$$\underbrace{u_t}_{\text{time differential}} - \underbrace{\nabla_x \cdot (\kappa \nabla_x u))}_{\text{diffusion of quantity } u} = \underbrace{f_m}_{\text{source terms}} \tag{24}$$

The general form of the classic heat equation is shown in Equation (24) where u models temperature or chemical concentration at a given point in space and time, κ is the heat capacity or diffusion coefficient (possibly temporally and/or spatially dependent), and f_m, $\forall m = 1, ..., M$ nodes are the source term (also possibly temporally and/or spatially dependent). To solve the heat equation numerically, it is necessary to specify both initial conditions and boundary conditions. The initial conditions specify the temperature distribution within the domain at time zero, while the boundary conditions specify the behavior of the solution at the boundaries of the domain. Dirichlet boundary conditions prescribe the temperature on the boundary, while Neumann boundary conditions prescribe the flux through the boundary. These boundary conditions are required because they provide the necessary information to obtain a unique solution to the heat equation. Without boundary conditions, the solution would not be well-defined, and there would be an infinite number of possible solutions that satisfy the PDE. Therefore, the appropriate choice of boundary conditions is crucial for obtaining physically meaningful solutions to the heat equation.

Concretely, a well-defined parabolic PDE consists of:

1. PDE: $u_t - \nabla_x \cdot (\kappa \nabla_x u) = f, \quad x \in \Omega, \quad t > 0$
2. Initial condition: $u(x, t = 0) = \eta(x), \quad x \in \Omega$
3. Boundary conditions: $\Gamma = \partial\Omega$
   - Dirichlet boundary condition: $u = u_D$ on $\Gamma \times (0, T)$ which directly prescribes the temperature or chemical concentration.
   - Neumann boundary condition: $(\kappa \nabla_x u) \cdot \boldsymbol{n} = g_M$ on $\Gamma \times (0, T)$ which prescribes the heat or diffusion flux across the boundary.

2.3.2 Finite Difference Grid

For simple geometries, the finite difference grid provides a straight-forward approach to discretizing the domain into a finite number of points or nodes. Each node represents a discrete location where the solution is evaluated, and the solution at each node is approximated using a finite difference approximation (example in Figure 1).

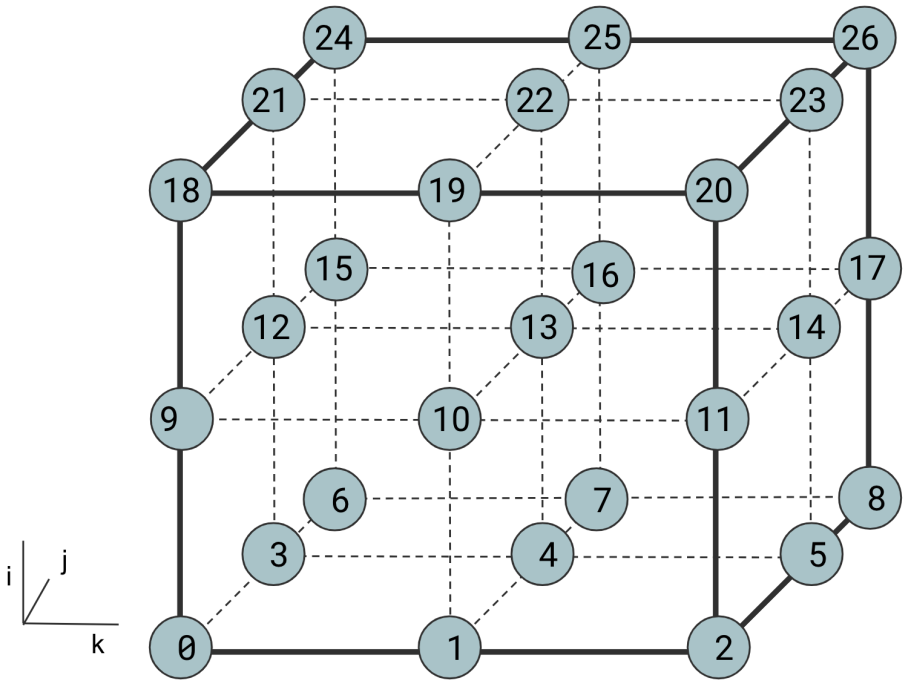


Figure 1: An example 3D finite difference grid where each node is equally spaced from neighboring nodes by some amount $\Delta x = \frac{L}{M-1}$, where L is the length of the domain across a single axis, and M are the total number of nodes along that same axis.

The accuracy of the FDM depends on the *size of the grid* and *the order of the finite difference approximation* used. In this way, the spatial domain of the PDE is transformed into a discrete set of equations that can be solved numerically using linear algebra techniques. The choice of grid size and finite difference approximation is critical in achieving a stable and accurate numerical solution.

2.3.3 Finite Difference Methods

Forward Difference in Time, Centered in Space (FTCS)

We seek to find an approximate solution $U_m^n \approx u(x_m, t_n)$ by approximating the diffusion term in Equation (24) which will enable us to "march forward in time", computing $U_m^{n+1}\ \forall\ m = 1, \ldots, M$ as a function of U_m^n at the previous time step.

FTCS seeks to find an approximate solution $U_m^n \approx u(x_m, t_n)$ by spatially discretizing the domain into a finite number of equally spaced grid points and approximates the spatial derivative at each point using a centered difference formula. The temporal derivative is approximated using a forward difference formula, and the solution at each grid point is updated at each time step based on the values at its neighboring points in space and the previous time step.

To obtain the FTCS approximation for $x \in \mathcal{R}^1$, we first consider a second-order center-difference approximation [**?**] of the Laplacian of the flux:

$$\nabla_x \cdot \boldsymbol{q} = \frac{\partial q}{\partial x} \approx \frac{q(x_1 + h/2) - q(x_1 - h/2)}{h} \tag{25}$$

where $\boldsymbol{q} = \kappa \nabla_x u$ and $h = \Delta x$. Assuming κ is constant in space, each partial flux can be further approximated by applying a successive first-order one-sided approximation:

$$q\big(x + h/2\big) \approx \kappa \frac{u(x+h) - u(x)}{h} \tag{26}$$

$$q\big(x - h/2\big) \approx \kappa \frac{u(x) - u(x-h)}{h} \tag{27}$$

yielding the centered-in-space approximation:

$$\frac{\partial}{\partial x}\left(\kappa\frac{\partial u}{\partial x}\right) = \kappa u_{xx} \approx \kappa\frac{u(x-h) - 2u(x) + u(x+h)}{h^2}. \tag{28}$$

Finally, the time derivative on the left-hand side of Equation (24) is approximated using a first-order one-sided approximation

$$U_m^{n+1} = U_m^n + \frac{k\kappa}{h^2}\left(U_{m-1}^n - 2U_m^n + U_{m+1}^n\right) + \frac{k}{h^2}f(U_m^n, t_n) \tag{29}$$

where $k = \Delta t$. This method is more commonly referred to as the *Forward Euler* time stepping scheme. The FTCS method is considered to be a first-order accurate in time, and second-order accurate in space. Since the right-hand side of Equation (29) is only dependent on U_m^n at the previous time step (all of which are known), this scheme is considered to be in an *explicit method* since U_m^{n+1} can be explicitly computed.

Trapezoidal Method

Like the FTCS method, the trapezoidal method also employs a second-order center-difference approximation for the spatial discretization of the diffusion term in Equation (24). However, rather than using an exclusive forward difference scheme, the trapezoidal method employs a weighted average of backward and forward differences in time parameterized by $\phi \in [0, 1]$:

$$\begin{aligned} U_m^{n+1} = \quad & U_m^n \\ & + \phi\frac{k}{h^2}\left(\kappa(U_{m-1}^n - 2U_m^n + U_{m+1}^n) + f(U_m^n, t_n)\right) \\ & + (1-\phi)\frac{k}{h^2}\left(\kappa\underbrace{(U_{m-1}^{n+1} - 2U_m^{n+1} + U_{m+1}^{n+1}) + f(U_m^{n+1}, t_n)}_{U_m^{n+1}\ is\ unknown}\right). \end{aligned} \tag{30}$$

When $\phi = 1/2$, the trapezoidal method reduces to the *Crank-Nicholson* method. The trapezoidal/Crank-Nicholson method is considered to be a second-order accurate in time, and second-order accurate in space. Since the right-hand side of Equation (29) is only dependent on both U_m^n at the previous time step (all of which are known) and U_m^{n+1} at the next time step (all of which are unknown), this scheme is considered to be in an *implicit method* since U_m^{n+1} must be implicitly computed.

Example: Crank-Nicholson for linear PDEs

Consider the simple parabolic PDE:

$$u_t = u_{xx}. \tag{31}$$

The Crank-Nicholson method discretizing Equation (31) yields:

$$U_m^{n+1} = U_m^n + \frac{k}{2h^2}\left(U_{m-1}^n - 2U_m^n + U_{m+1}^n + U_{m-1}^{n+1} - 2U_m^{n+1} + U_{m+1}^{n+1}\right). \tag{32}$$

In the case in which the PDE is linear, we can move all terms containing the unknown solutions U_m^{n+1} at the text time step to the left-hand side and leave all of the known solutions U_m^n at the previous time step on the right-hand side:

$$-\frac{k}{2h^2}U_{m-1}^{n+1} + (1+2\frac{k}{2h^2})U_m^{n+1} - \frac{k}{2h^2}U_{m+1}^{n+1} = \underbrace{\frac{k}{2h^2}U_{m-1}^n + (1-2\frac{k}{2h^2})U_m^n - \frac{k}{2h^2}U_{m+1}^n}_{known\ \boldsymbol{b}}. \tag{33}$$

Our known solutions at the previous and next time steps can be organized as vectors $\boldsymbol{U}_m^n, \boldsymbol{U}_m^{n+1} \in \mathcal{R}^{m_{nodes}}$ respectively. This means we can re-write the left-hand side of Equation (33) as system of M equations, and the right-hand side as a single column vector:

$$\underbrace{\begin{bmatrix} (1+2\frac{k}{2h^2}) & -\frac{k}{2h^2} & 0 & \dots & & 0 \\ -\frac{k}{2h^2} & (1+2\frac{k}{2h^2}) & -\frac{k}{2h^2} & \dots & & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & -\frac{k}{2h^2} & (1+2\frac{k}{2h^2}) & -\frac{k}{2h^2} \\ 0 & & \dots & & -\frac{k}{2h^2} & (1+2\frac{k}{2h^2}) \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} U_1^{n+1} \\ U_2^{n+1} \\ \vdots \\ U_{M-1}^{n+1} \\ U_M^{n+1} \end{bmatrix}}_{\boldsymbol{x}} = \boldsymbol{b} \tag{34}$$

where

$$\boldsymbol{b} = \begin{bmatrix} \frac{k}{2h^2}\left(g_0(t_n) + g_0(t_{n+1})\right) + (1-\frac{k}{2h^2})U_1^n + \frac{k}{2h^2}U_2^n \\ \frac{k}{2h^2}U_1^n + (1-2\frac{k}{2h^2})U_2^n + \frac{k}{2h^2}U_3^n \\ \vdots \\ \frac{k}{2h^2}U_{M-2}^n + (1-2\frac{k}{2h^2})U_{M-1}^n + \frac{k}{2h^2}U_M^n \\ \frac{k}{2h^2}U_{M-1}^n + (1-2\frac{k}{2h^2}) + \frac{k}{2h^2}\left(g_1(t_n) + g_1(t_{n+1})\right) \end{bmatrix}$$

which reduces to solving $\boldsymbol{Ax} = \boldsymbol{b}$. Since $\boldsymbol{A}$ is tri-diagonal, this problem can be solved as efficiently as an explicit method.

2.3.4 Analysis of Numerical Methods

Local Truncation Error and Order of Accuracy

Numerical methods provide approximate solutions for ODEs/PDEs, and the accuracy of the solutions are dependent on the time discretization $k = \Delta t$ and spatial discretization $h = \Delta x$. After taking a single time step, some amount of error is introduced to the solution, also referred to as the local truncation error (LTE). This provides a method for analyzing the order of accuracy of a numerical method and how the quickly the error decreases as $h,\ k \to 0$. The LTE for the simple ODE $u_t = \lambda u + f(t)$ with an initial value $u(0) = u_0$ for the Backward Euler time discretization method is:

$$\tau_n = \frac{u(t_n) - u(t_{n-1})}{k} - \lambda u(t_n) - f(t_n), \quad \forall n = 1, ..., N \tag{35}$$

Order of accuracy is a measure of how well a finite difference method approximates a solution to a differential equation and describes how quickly the error between the approximate solution and the true solution decreases as $k,\ h \to 0$. Intuitively, a method will converge to the true solution at rate proportional to the order of accuracy. For example, a method with second order of accuracy in space will have an error that decreases proportional to the square of the grid spacing. Although we *expect* the approximate solution to approach the true solution such that the error of the method $||\tau(x,t)|| \to 0$, for convergence to the true solution to occur, the temporal and spatial discretization must satisfy a the two properties: consistency and stability.

Consistency

As both $h,\ k \to 0$, the we *expect* the approximate solution to approach the true solution such that the LTE of the method $||\boldsymbol{\tau}|| \to 0$. If this is indeed true, the discretization method is said to be *consistent*, meaning that the discrete equation approximates the same process as the underlying PDE as the temporal and spatial grids become finer at a rate that is at least as fast as $h,\ k \to 0$.

More concretely, *consistency* requires:

$$||\boldsymbol{\tau}||_\infty = \max_{n=1,\dots,N} |\tau_n| \ \to 0 \text{ as } \Delta t \to 0. \tag{36}$$

In the *LTE and order of accuracy for the Crank-Nicolson Method* example, the LTE is shown to be $\mathcal{O}(k^2 + h^2)$. A method is said to be *consistent* since the global order of accuracy will agree with the order of the LTE:

$$U_m^n - u(X,T) = \mathcal{O}(k^2 + h^2). \tag{37}$$

To determine whether a method is consistent, we need to analyze LTE. The following are two examples for deriving the LTE and order of accuracy for the *Forward in Time, Centered in Space* and *Crank-Nicolson* methods, and how the error corresponds to $k,\ h$.

Example: LTE and order of accuracy for FTCS method
Consider the simple parabolic PDE $u_t = u_{xx}$. The resulting error for a chosen time step k and for a chosen spatial step h for Equation (29) is defined as:

$$\tau(x,t) = \frac{U_m^{n+1} - U_m^n}{k} - \frac{\kappa}{h^2}\left(U_{m-1}^n - 2U_m^n + U_{m+1}^n\right). \tag{38}$$

To obtain the LTE, insert the true solution:

$$\tau(x,t) = \frac{u(x,t+k) - u(x,t)}{k} - \frac{\kappa}{h^2}\left(u(x-h,t) - 2u(x,t) + u(x+h,t)\right) \tag{39}$$

For the first term on the right-hand side, we care about the error relative to k, and for the second term we care about the error relative to h. If we make the assumption that u is smooth, we can expand each term around $k,\ h$ respectively: Assuming that u is sufficiently smooth and differentiable, each u term containing increment in k and h can be expanded in a Taylor series:

$$u(x,t+k) = u + ku_t + \frac{1}{2}k^2u_{tt} + \frac{1}{6}k^3u_{ttt} + \mathcal{O}(t^4) \tag{40}$$

$$u(x-h,t) = u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + \mathcal{O}(h^4) \tag{41}$$

$$u(x+h,t) = u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + \mathcal{O}(h^4). \tag{42}$$

Inserting each Taylor series expansion from Equation (40), Equation (41), and Equation (42) into Equation (39) yields:

$$\tau(x,t) = \left(\underbrace{u_t}_{=\,u_{xx}} + \underbrace{\frac{1}{2}ku_{tt}}_{=\,u_{txx}=\,u_{xxxx}} + \mathcal{O}(k^2)\right) - \left(u_{xx} + \frac{1}{12}h^2u_{xxxx} + \mathcal{O}(h^3)\right) \tag{43}$$

Recall that $u_t = u_{xx}$ and $u_{tt} = u_{xxxx}$ allowing for additional terms to simplify:

$$\tau(x,t) = \left(\frac{1}{2}k - \frac{1}{12}h^2\right)u_{xxxx} = \mathcal{O}(k + h^2).$$

This derivation shows that the error for the FTCS method is expected to increase linearly with k and to increase with the power of two with h. This is more commonly stated to be *first-order accurate in time, and second-order accurate in space.*

Example: LTE and order of accuracy for the Crank-Nicolson method
Once again, consider the simple parabolic PDE $u_t = u_{xx}$. The resulting error for a chosen time step k and for a chosen spatial step h is defined as:

$$\tau(x,t) = \frac{U_m^{n+1} - U_m^n}{k} + \frac{1}{2h^2}\left(U_{m-1}^n - 2U_m^n + U_{m+1}^n + U_{m-1}^{n+1} - 2U_m^{n+1} + U_{m+1}^{n+1}\right). \tag{44}$$

To obtain the LTE, insert the true solution:

$$\begin{aligned}\tau(x,t) \;=\; & \frac{u(x,t+k) - u(x,t)}{k} \\ & + \frac{1}{2h^2}\Big(u(x-h,t) - 2u(x,t) + u(x+h,t) \\ & \qquad + \; u(x-h,t+k) - 2u(x,t+k) + u(x+h,t+k)\Big).\end{aligned} \tag{45}$$

Once again assuming the true solution u is sufficiently smooth and differentiable, the Taylor series expansion for $u(x,t+k)$, $u(x-h,t)$ and $u(x+h,t)$ can be recycled from Equation (40), Equation (41), Equation (42). For $u(x-h,t+k)$ and $u(x+h,t+k)$, a multi-variable Taylor series expansion is required:

$$u(x-h,t+k) = u - hu_x + ku_t + \frac{1}{2}h^2u_{xx} - hku_{tx} + \frac{1}{2}k^2u_{tt} + \mathcal{O}(h^3,k^3,h^2k,hk^2) \tag{46}$$

$$u(x+h,t+k) = u + hu_x + ku_t + \frac{1}{2}h^2u_{xx} + hku_{tx} + \frac{1}{2}k^2u_{tt} + \mathcal{O}(h^3,k^3,h^2k,hk^2). \tag{47}$$

Inserting each Taylor series expansion from Equations 40-42 and Equations 46-47 into Equation (45) and a lot of tedious cancellation of terms yields:

$$\tau(x,t) = \mathcal{O}(k^2,h^2). \tag{48}$$

This derivation shows that the error for the Crank-Nicolson method is expected to increase with the power of two with k and to increase with the power of two with h. This is more commonly stated to be *second-order accurate in time, and second-order accurate in space.*

Stability

Small variations in the initial and boundary conditions may cause sensitive numerical methods to be unstable. These small LTEs may cause the approximate solution to diverge from the true solution over time since each error may compound on itself. For time-dependent ODEs, the concept of *absolute stability* allows us to analyze terms that recursively amplify or dampen a solution after a time step. The stability theory for time-dependent PDEs can be related to the stability theory for time-dependent ODEs through the method of lines (MOL) discretization of the PDE (see Figure 2). The MOL approach first discretizes in space alone, resulting in a system of ODEs that can be solved using methods for ODEs. This system of ODEs is known as a semi-discrete method since it is discretized in space but not yet in time.

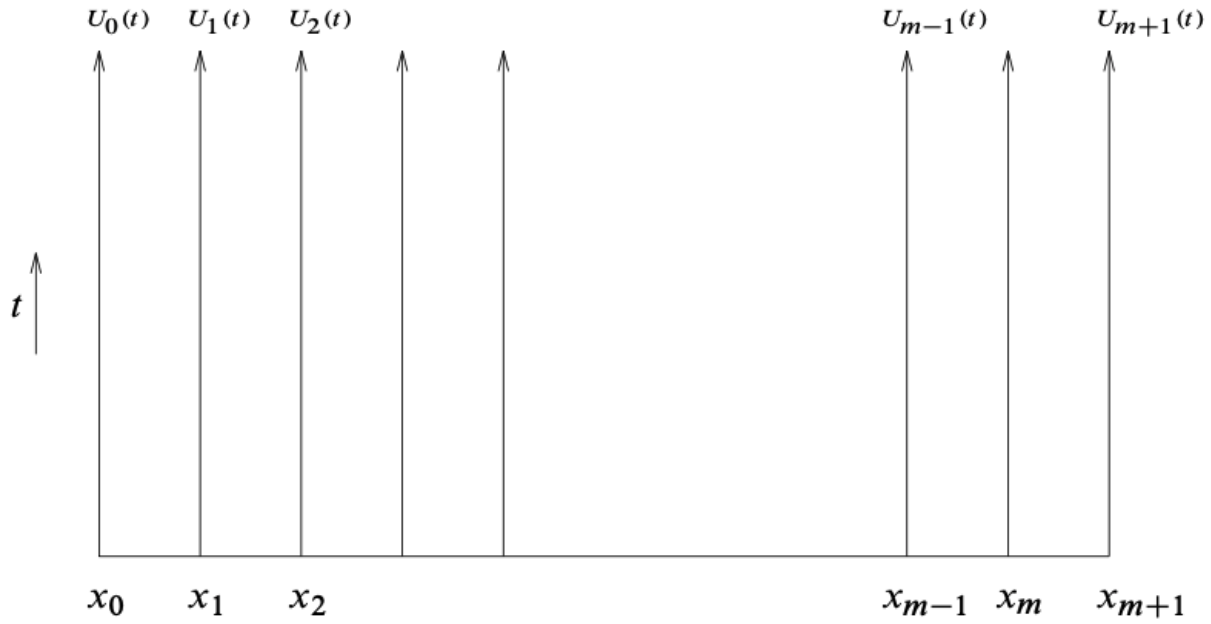


Figure 2: Method of lines interpretation. $U_i(t)$ is the solution along the line forward in time at the grid point x_i [**?**].

Once system of semi-discrete ODE's has been obtained from the MOL, we can apply our chosen time stepping scheme to obtain fully discrete method in which we can isolate the amplifying or dampening factor that affects stability. For ODE absolute stability, this factor is a scalar value (see *Absolute Stability* example below). However, once the MOL has been applied to a PDE, we obtain a system of equations that can be represented as matrices and vectors (if linear). This means the amplifying/dampening factor can be described by analyzing eigenvalues of matrices.

The following provides explicit examples for *Absolute stability*, *Stability Analysis of FTCS* and *Stability analysis of Crank-Nicolson*.

Example: Absolute stability of Forward Euler
The notion of *absolute stability* states that an ordinary differential equation (ODE) is absolutely stable under a numerical method if the numerical solution remains bounded as the step size or time step approaches zero, regardless of initial conditions. Consider the following classic ODE:

$$u_t = \lambda u(t) + g(t), \quad u(t_0) = \eta \tag{49}$$

and its discretization using Forward Euler time stepping

$$U^{n+1} = (1 + k\lambda)U^n + kg(t) \tag{50}$$

where the error

$$k\tau^n = u(t_{n+1}) - u(t_n) - k\big(\lambda u(t_n) + g(t_n)\big) \tag{51}$$

and its LTE:

$$\tau^n = \frac{1}{2}ku_{tt}(t_n) + \mathcal{O}(k^2). \tag{52}$$

To obtain the global error, subtract the following:

$$u(t_{n+1}) = (1 + k\lambda)u(t_n) + kg(t_n) + k\tau^n \tag{53}$$
$$U^{n+1} = (1 + k\lambda)U^n + kg(t_n) \tag{54}$$

to get

$$E^{n+1} = (1 + k\lambda)E^n - k\tau^n. \tag{55}$$

As the numerical method marches in time, the error is recursively scaled by $(1 + k\lambda)$. For $N + 1$ total time steps, the final global error is:

$$E^{N+1} = (1 + k\lambda)^N E^N - k\tau^N. \tag{56}$$

where the superscript N above E represents the second to last time step and the superscript N above $(1 + k\lambda)$ obnoxiously represents a power. We expect the to see an exponential grown factor any k for which $|1 + \lambda k| > 1$. Therefore we require $|1 + k\lambda| \leq 1$ for the Forward Euler method to be absolutely stable. More generally, since λ maybe complex, the region of absolute stability is a disk of radius 1, centered at the point -1 (see Figure 3).

Example: Absolute stability of Trapezoidal
Once again, consider the following classic ODE:

$$u_t = \lambda u(t) + g(t), \quad u(t_0) = \eta \tag{57}$$

and its discretization using Trapezoidal time stepping

$$U^{n+1} = U^n + \frac{k}{2}\big(\lambda U^n + g(t)\big) + \frac{k}{2}\big(\lambda U^{n+1} + g(t_{n+1})\big) \tag{58}$$

and solving for U^{n+1} and inserting exact solution yields:

$$U^{n+1} = \frac{(1+\frac{k\lambda}{2})}{(1-\frac{k\lambda}{2})}U^n + \frac{\frac{k}{2}(g^n + g^{n+1})}{(1-\frac{k\lambda}{2})} \tag{59}$$

$$U(t_{n+1}) = \frac{(1+\frac{k\lambda}{2})}{(1-\frac{k\lambda}{2})}U(t_n) + \frac{\frac{k}{2}(g^n + g^{n+1})}{(1-\frac{k\lambda}{2})} + k\tau^n. \tag{60}$$

Subtracting the two and analyzing for the global error:

$$E^{N+1} = \rho^N E^N - k\tau^N, \quad \textit{where } \rho = \frac{(1+\frac{k\lambda}{2})}{(1-\frac{k\lambda}{2})}. \tag{61}$$

where the superscript N above E represents the second to last time step and the superscript N above ρ obnoxiously represents a power. For absolute stability, we require $|\rho| < 1$. By induction, it is simple to see that $|\rho| < 1 \ \forall \ \lambda k < 0$. This means that the trapezoidal method's stability region is the entire left-hand plane in Figure 3.

Example: Stability analysis of FTCS

As with the analysis of *absolute stability* for an ODE, analysis can be performed on the FTCS method by using first spatially discretizing in space using *Method of Lines* (MOL) to obtain a system of ODEs to interpret the method as a time integration methods of ODEs.
For the simple heat equation $u_t = u_{xx},\ u(t=0) = u_0$, the MOL for FTCS yields a semi-discrete ODE:

$$u_t = \frac{1}{h^2}(U^n_{m-1} - 2U^n_m + U^n_{m+1}), \quad m = 1, \dots, M. \tag{62}$$

After temporally discretizing, the boundary conditions can be explicitly encoded in the matrix form:

$$\boldsymbol{U}^{n+1} = (\boldsymbol{I} + k\boldsymbol{A})\boldsymbol{U}^n + k\boldsymbol{g} \tag{63}$$

where

$$\boldsymbol{A} = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -2 \end{bmatrix}, \quad \boldsymbol{g} = \frac{1}{h^2}\begin{bmatrix} g_0(t) = U_0(t) \\ 0 \\ \vdots \\ 0 \\ g_1(t) = U_M(t) \end{bmatrix}. \tag{64}$$

As in the *absolute stability* example, we can compute the final global error:

$$\boldsymbol{E}^{N+1} = (\boldsymbol{I} + k\boldsymbol{A})^N \boldsymbol{E}^N - k\boldsymbol{\tau}^N \tag{65}$$

where the superscript N above $\boldsymbol{E}$ represents the second to last time step and the superscript N above $(\boldsymbol{I} + k\boldsymbol{A})$ obnoxiously represents a power. More importantly than the inconsistent notation are the eigenvalues of $\boldsymbol{A}$ which determine whether the global error converges or diverges. For the FTCS method to be stable, we require that $k\lambda_m \in \mathcal{S}$. For this simple heat equation, the eigenvalues of $\boldsymbol{A}$ are:

$$\lambda_m = \frac{2}{h^2}\big(\cos(m\pi h) - 1\big), \quad \forall m = 1, \dots, M. \tag{66}$$

where $h = \frac{1}{m+1}$. The magnitude of $|\lambda_m|$ is largest when $m = M$ and resulting in the eigenvalue:

$$\lambda_M \approx \frac{2}{h^2}\big(cos(M\pi h) - 1\big) = \frac{-4}{h^2}. \tag{67}$$

Therefore we require that $-k\frac{4}{h^2} \in \mathcal{S}$, which for Forward Euler corresponds to $-2 \le -k\frac{4}{h^2} \le 0$, or $\frac{k}{h^2} \le \frac{1}{2}$.

Example: Stability analysis of Crank-Nicolson

As with the analysis of *absolute stability* for an ODE, analysis can be performed on the FTCS method by using first spatially discretizing in space using *Method of Lines* (MOL) to obtain a system of ODEs to interpret the method as a time integration methods of ODEs.
For the simple heat equation $u_t = u_{xx},\ u(t=0) = u_0$, the MOL for FTCS yields a semi-discrete ODE:

$$u_t = \frac{1}{h^2}(U^n_{m-1} - 2U^n_m + U^n_{m+1}), \quad m = 1, \dots, M. \tag{68}$$

Convergence

The Dahlquist Equivalence Theorem is the primary tool for assessing whether or not a numerical scheme is convergent. Using the concepts of consistency and zero stability alone, we can draw a conclusion about the convergence. To summarize from above, we have the following concepts:

- Consistency: In the limit $k \to 0$, the method gives a consistent discretization of the ordinary differential equation.
- Zero stability: In the limit $k \to 0$, the method has now solutions that grow unbounded as $N = T/k \to \infty$.

The Dahlquist Equivalence Theorem guarantees that a method is consistent and stable is convergent, and also that convergent method is consistent and stable.

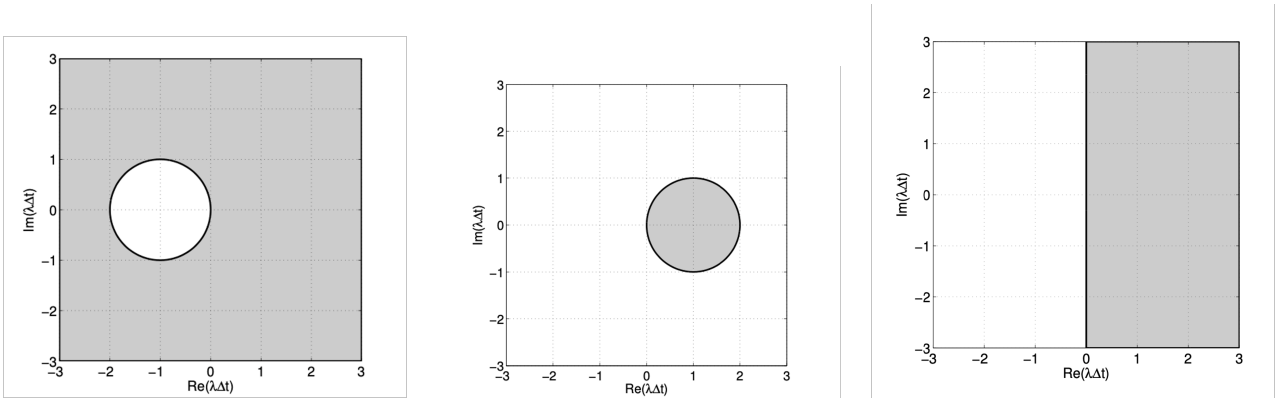


Figure 3: A comparison of the stability regions for 1) Forward Euler, 2) Backward Euler and 3) Trapezoidal where the white area corresponds to stability (the absolute stability region) and the gray area to instability [**?**]

Section 3: Optimization

3.0.1 Introduction

Although colloquially used in everyday language, optimization (or commonly spelled by the intelligentsia as *optimisation*) has a rich and deep philosophical meaning in mathematics and physics. The entire field can arguably be considered the core idea of any dynamical system that seeks equilibrium. In physics and chemistry, these dynamical systems seek paths that will reduce its energy state to some equilibrium point. Similar problems in finance, engineering, operations research, and/or parameter *learning* can also be cast following the same metaphor by maximizing or minimizing a desired quantity by varying the *decision variables* (also referred to as *optimization variables* or *manipulated variables*). In some applications, the system is driven to a certain state following a local descent path to the nearest minimum energy state, as does a ball rolling down a hill to find peace in a valley. Other examples of optimization appear out of randomness, such as life in Darwin's game of survival of the fittest.

In general, we can approximate these problem statements mathematically as a set of inputs $\boldsymbol{x} \in \mathcal{X}$ and outputs $\boldsymbol{y} \in \mathcal{Y}$, via some mapping of the form $f \ : \ \mathcal{R}^n \rightarrow \mathcal{R}$. For mathematical optimization, the goal is to find some input $\boldsymbol{x} \in \mathcal{R}^n$ that minimizes or maximizes the output $y \in \mathcal{R}$. The canonical, or standard form of such an *optimisation* problem can be read as the following:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f_0(\boldsymbol{x}) \\ \text{subject to} \quad & f_i(\boldsymbol{x}) \leq b_i, \quad i = 1, ..., m. \end{aligned}$$

which reads as "minimize $f_0(\boldsymbol{x})$ subject to $f_i(\boldsymbol{x}) \leq b_i$ for all $i = 1, ..., m$", where $f_0(x)$ is the objective function $f_0 \ : \ \mathcal{R}^n \rightarrow \mathcal{R}$ (also referred to as the cost or evaluation function), the vector $\boldsymbol{x} \in \mathcal{R}^n$, the functions $f_i \ : \ \mathcal{R}^n \rightarrow \mathcal{R}, \ i = 1, ..., m$ are the inequality *constraints*, and the constants $b_1, \ ..., b_m$ are the bounds for the constraints. We call the best choice $\boldsymbol{x}^*$ the *optimal solution*. For more details on notation, see Boyd.

As one might expect, it is only natural for individuals pursuing the topic of *optimisation* to cleverly conclude (hopefully prior to beginning graduate school, and god forbid, before middle life) that *"oh my god, everything is an optimisation problem"*, thus leading the individual to rich and deep thoughts of the nature of reality, philosophy, and an optimal utopia. But alas, remember the words of Steven Boyd echoing through the hallowed convex halls of Stanford University, "to say everything is an optimisation problem is a stupid tautology". In fact, it does not mean anything to shout such a revelation as nothing can be concluded from such a whimsical statement. What matters is what the *optimisation* problem is, since most optimisation problems *you can not solve*. So oh dear reader, come to this clever conclusion as rapidly as possible, and the wipe your shoes of such dirt before entering mathematical wonders of *optimisation*.

But oh, what *optimisation* problems can we solve, our dear reader (who may or may not exist) must be wondering. Here, I seek to describe just that. Let us first consider the set of all tractable *optimisation* problems $\min_{\boldsymbol{x}} \ f_0(\boldsymbol{x})$. The tractability of such a problem first begins with the the structure of $f_0(\boldsymbol{x})$. Under some circumstances, $f_0(x)$ is an explicit function in which we can analyze or operate on. For example, the univariate objective function $f_0(\boldsymbol{x}) = (x+2)^2 + 1$ is a familiar one from basic maths courses from early secondary school. The minimization of such a function is as easy whipping out our $\frac{d}{dx}$ operator, setting its derivative equal to zero, and solving for x. In this simplified case, the solution to the *optimisation* problem is written as $x^* = -2$. In this case, the *optimisation* problem is considered to have an *analytical solution*. And thus we have our first approach to solving an *optimisation* problem. Unfortunately, analytical solutions are far from achievable for most tractable problems. *Numerical solutions* shall be our toolset for solving the nastier *optimisation* problems. It is in this realm we seek to develop and compare algorithms for solving a tractable $\min_x \ f_0(\boldsymbol{x})$.

The following sections will describe numerical techniques for solving tractable optimization problems. These problems can generally be subdivided into two categories: convex optimization and non-convex (blackbox) optimization.

3.0.2 Gradient-based methods

Newton's method:

Let us first consider an convex objective $f_0(x)$ and method in which we can approach solving the minimization. Convexity enables us to makes certain assumptions about optimality conditions. One optimality condition being

$$\nabla_{\boldsymbol{x}} f_0(\boldsymbol{x}) = 0,$$

or more simply, we would like to find the root of the gradient of $f_0(\boldsymbol{x})$.

Newton's method is a simple numerical approach used for find roots of an equation if an initial candidate solution is close the exact solution. Here I derive Newton's method using a Taylor Series expansion.

Suppose $f_0(\boldsymbol{x})$ is a continuous function on a closed interval $[a, b]$ and has $n+1$ continuous derivatives on the open interval (a, b). If $x, c \in (a, b)$, then the expansion of $f_0(x)$ about c is:

$$f(c) + f'(c)(x-c) + \frac{1}{2!}f''(c)(x-c)^2 + \frac{1}{3!}f'''(c)(x-c)^3 + ... + \frac{1}{n!}f^n(c)(x-c)^n$$

or more concisely:

$$f(x) \approx \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(c)(x-c)^k$$

Provided an initial guess value $\boldsymbol{x}^j$,

By approximating $f_0(\boldsymbol{x})$ using this expansion, we can find the optimal x^* by expanding $\nabla_x f_0(x)$ around a initial guess value x^j, which at optimum should be equal to zero:

$$\nabla_x f_0(x^{j+1}) = \nabla_x f_0(x^j) + \nabla_x(\nabla_x f_0(x^j))(x^{j+1} - x^j) + \mathcal{O}(x^{j+1} - x^j)^2 = 0|_{x^{j+1}=x^*}.$$

The resulting error $\mathcal{O}(x^{j+1} - x^j)^2$ indicates that this optimization method is a second-order gradient-based approach. Rearranging the above equation yields:

$$-\nabla_x f_0(x^j) = \nabla_x(\nabla_x f_0(x^j))(x^{j+1} - x^i)$$

where we seek to solve for x^{j+1}, where $\nabla_x(\nabla_x f_0(x^j))$ is the Hessian. If x is a vector, than a matrix inversion is required to solve for x^{j+1}.

$$\boldsymbol{x}^{j+1} = \boldsymbol{x}^j - [\nabla_x(\nabla_x f_0(\boldsymbol{x}^j)]^{-1} \nabla_x f_0(\boldsymbol{x}^j).$$

Explicitly, the Hessian is defined as:

$$H = \nabla_x(\nabla_x f_0(\boldsymbol{x}_i)) = \begin{bmatrix} \frac{\partial f_0^2}{\partial x_1^2} & \frac{\partial f_0^2}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f_0^2}{\partial x_1 \partial x_n} \\ \frac{\partial f_0^2}{\partial x_2 \partial x_1} & \frac{\partial f_0^2}{\partial x_2^2} & \cdots & \frac{\partial f_0^2}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_0^2}{\partial x_n \partial x_1} & \cdots & \cdots & \frac{\partial f_0^2}{\partial x_n^2} \end{bmatrix}$$

This means that use of Newton's methods for gradient descent requires an inversion of a $n \times n$ matrix to compute the next candidate solution. The update equation can be explicitly referred to as:

$$\begin{bmatrix} x_1^{j+1} \\ \vdots \\ x_n^{j+1} \end{bmatrix} = \begin{bmatrix} x_1^j \\ \vdots \\ x_n^j \end{bmatrix} - \begin{bmatrix} \frac{\partial f_0^2}{\partial x_1^2} & \cdots & \frac{\partial f_0^2}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_0^2}{\partial x_n \partial x_1} & \cdots & \frac{\partial f_0^2}{\partial x_n^2} \end{bmatrix} \begin{bmatrix} \frac{\partial f_0}{\partial x_1} \\ \vdots \\ \frac{\partial f_0}{\partial x_n} \end{bmatrix}.$$

Sometime, we do not have an explicit form of $f_0(\boldsymbol{x})$ and therefore do not have access to the gradients directly. We can approximate these gradients numerically (see Section on Numerical Methods), however Newton's method quickly deteriorates as a viable option as $n \to \infty$ since matrix inversion has computational complexity on the order of $\mathcal{O}(n^3)$.

First-order methods

Gradient descent optimizes a function by using first-order gradient information, while Newton's method optimizes a function utilizing second-order gradient information. The most basic implementation of gradient descent computes the gradient of the objective function and slightly perturbs the current position. Although first-order methods are considerably faster, these algorithms can be unstable and perform poorly on non-convex functions. Consider the example in which parameters of a model are being tuned to fit a data set:

Let θ be the parameters for a function approximator $f_0(x)$. Gradient descent can computed iteratively as

$$\theta^{i+1} = \theta^i - \gamma \nabla_\theta f_0(x), \quad x \in \mathcal{D}$$

where γ is a hyper parameter chosen to determine the incremental step size performed by optimizer.

An example implementation in Python is shown below while attempting to optimize the non-convex Beale Function:

```
 # manual gradients and hessians for each funciton
def grad_beale(curr_loc, ord=1, method=1):
    """ computes gradient and hessian
        inputs:
            - {x, y} coords: ndarray
            - ord: 1 or 2 for gradient or hessian
            - method: 1 or 2 for analtyical or numerical
        outputs:
            - gradient
            - hessian """
    x = curr_loc[0]
    y = curr_loc[1]

    # compute gradient
    dfdx = 2 * (1.5 - x + x * y) * (-1 + y) + \
           2 * (2.25 - x + x * y ** 2) * (-1 + y ** 2) + \
           2 * (2.6250 - x + x * y ** 3) * (-1 + y ** 3)
    dfdy = 2 * (1.5 - x + x * y) * x + \
           2 * (2.25 - x + x * y ** 2) * (2 * x * y) + \
           2 * (2.6250 - x + x * y **3) * (3 * x * y ** 2)

    gradient = np.array([dfdx, dfdy])

    if ord == 1:
        return gradient

    elif ord == 2:
        # compute hessian
        ddfddx = 2 * (y - 1) * (y - 1) + \
                 2 * (y * y - 1) * (y * y - 1) + \
                 2 * (y * y * y - 1) * (y * y * y - 1)
        ddfdydx = 4 * x * (y ** 2 - 1) + \
                  4 * y * (x * y * y - x + 2.25) + \
                  6 * x * (y * y * y - 1) * y * y + \
                  y * y ** 2 * (x * y * y * y - x + 2.625) + \
                  2 * x * (y - 1) + \
                  2 * (x * y - x + 1.5)
        ddfddy = 18 * x * x * y * y * y * y + \
                 8 * x * x * y * y + \
                 4 * x * (x * y * y - x + 2.25)

        hessian = np.array([[ddfddx, ddfdydx], [ddfdydx, ddfddy]])
        return gradient, hessian
```

```
def grad_descent(grad_fn, init_loc=[0, 0], n_steps=10000, f=None):
gamma = 1e-6

# initialize trajectories
traj = np.zeros((n_steps, len(init_loc)))
traj[0, :] = np.array(init_loc)

for i in range(1, n_steps):
    if f is not None:
        traj[i, :] = traj[i - 1, :] \
                   - gamma * grad_fn(f=f,
                                     curr_loc=traj[i - 1, :],
                                     ord=1)
    else:
        traj[i, :] = traj[i - 1, :] \
                   - gamma * grad_fn(traj[i - 1, :],
                                     ord=1)

return traj

for i in range(len(initial_points)):
trajectories_gd = grad_descent(grad_beale, initial_points[i])
output, fig, ax = beale([], [])
ax.scatter(trajectories_gd[:-1, 0], trajectories_gd[:-1, 1],
           s=25, facecolor='orange')
ax.scatter(trajectories_gd[-1, 0], trajectories_gd[-1, 1],
           s=150, facecolor='m', marker='*', label='gd opt')
```

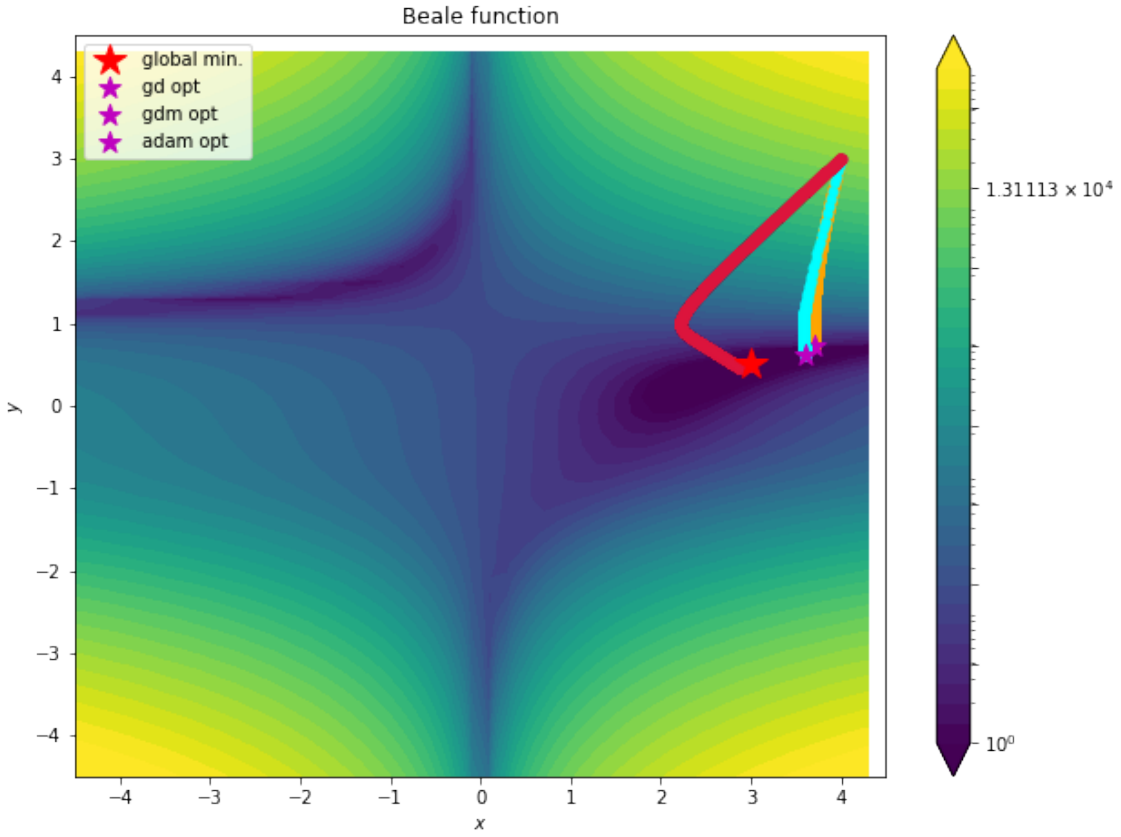


Figure 4: A benchmark comparison between gradient descent, gradient descent + momentum, and ADAptive Momentum (ADAM) gradient descent on the non-convex Beale Function

3.0.3 Blackbox optimization + non-gradient-based methods

Genetic algorithms Simulated annealing Bayesian optimization

Question 1

Let's say you are trying to learn from a distribution of data. What do you do?

(Answer on page 21)

Solution:-

You simply apply RL until you achieve an h-index of 130.

(Question on page 21)

Section 4: Machine Learning

5 Deep Neural Networks

4.1 Multi-Layer Perceptron

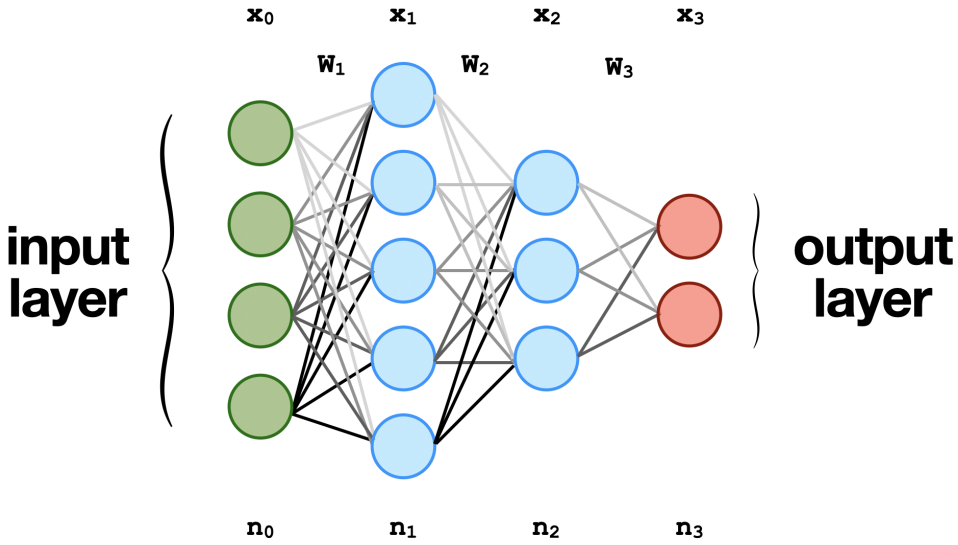


Figure 5: A typical visual representation of fully-connected 4-layer neural network or multi-layer perception (MLP) where $\boldsymbol{x}_i$ is input vector to the next layer, $\boldsymbol{W}_i$ are te matrix weights, and n_i are the number of neurons in a layer.

Neural networks are a subset of machine learning algorithms that are designed to model the behavior of the human brain. They are composed of interconnected nodes or neurons, which are organized into layers and process information in a parallel and distributed manner. Neural networks have gained significant attention in recent years due to their ability to solve complex problems in various domains, including image and speech recognition, natural language processing, and predictive analytics. One of the key strengths of neural networks is their ability to approximate any function to arbitrary accuracy, given a sufficiently large number of neurons in the hidden layers. This property, known as universal approximation, has made neural networks an essential tool for data scientists and researchers seeking to develop intelligent systems capable of learning and adapting to changing environments. In this background section, we will provide a comprehensive overview of the fundamentals of neural networks, including their architecture, training algorithms, and applications.

Multi-layer perceptions (MLPs) are a class of feed-forward neural networks that consist of one or more hidden layers of fully connected neurons (see Figure 5). In an MLP, each neuron receives inputs from all the neurons in the previous layer and computes a weighted sum of these inputs with additional bias term b. The weighted sum is then passed through an activation function, which introduces non-linearity into the model. The most commonly used activation functions include `sigmoid`, `ReLU`, and `tanh`. The output of each neuron is then passed on to the neurons in the next layer until the output layer is reached, which produces the final output of the network. This sequence of operations is typically referred to as the *forward pass*. The neural network *learns* by iteratively updating itself using optimization methods that minimizes the error between output of the neural network and the output from the training data, also known as the *backward pass* or *backward propagation*.

Forward Pass

The computations performed by the neurons in an MLP can be mathematically represented as matrix-vector multiplication, where the weights of each neuron in a layer are represented as a row in a weight matrix. Consider the MLP represented in Figure 5. The first layer contains four input nodes, which can be represented as an input vector $\boldsymbol{x}_0 \in \mathcal{R}^{n_0+1}$. The additional dimension is a place holder for the bias term b. The non-linear mapping between each layer begins with a linear matrix-vector operation between the preceding layer and its corresponding weight matrix $W_1 \in \mathcal{R}^{n_1 \times n_0+1}$. The output of this matrix is passed element-wise through a non-linear activation function of the user's choice. The output of this operation has a dimension equal to the size of the subsequent layer. Explicitly, this operation is

$$\phi(\boldsymbol{W}_1\boldsymbol{x}_0) = \phi\Bigg(\begin{bmatrix} W^1_{0,0} & W^1_{0,1} & W^1_{0,2} & W^1_{0,3} & b_0 \\ W^1_{1,0} & W^1_{1,1} & W^1_{1,2} & W^1_{1,3} & b_1 \\ W^1_{2,0} & W^1_{2,1} & W^1_{2,2} & W^1_{2,3} & b_2 \\ W^1_{3,0} & W^1_{3,1} & W^1_{3,2} & W^1_{3,3} & b_3 \\ W^1_{4,0} & W^1_{4,1} & W^1_{4,2} & W^1_{4,3} & b_4 \end{bmatrix} \begin{bmatrix} x^0_0 \\ x^0_1 \\ x^0_2 \\ x^0_3 \\ 1 \end{bmatrix} \Bigg) = \boldsymbol{x}_1 \tag{69}$$

where $W^1_{i,j}$ represents the element in the $(i+1)$-th row and $(j+1)$-th column of the weight matrix W_1, using zero-indexing, the superscript 1 indicates that $\boldsymbol{W}$ is weight matrix mapping to the first layer, and ϕ represents the activation function of choice.

Iterating this non-linear mapping procedure through each subsequent layer in Figure 5 yields the following result

$$\psi\Bigg(\underbrace{\boldsymbol{W}_3\, \phi\big(\boldsymbol{W}_2 \underbrace{\phi(W_1\boldsymbol{x}_0)}_{\boldsymbol{x}_1}\big)}_{\boldsymbol{x}_2} \Bigg) = \boldsymbol{x}_3. \tag{70}$$

where the outer activation function ψ is the soft max function used to produce a probability distribution over the possible output classes.

Backward Pass

When training a model, inputs from a training data set are passed through a neural network following the matrix-vector multiplications provided above and the error between the outputs of the neural network and the training data can mathematically defined as the norm of the differences between both outputs:

$$\mathcal{L} = \frac{1}{2}||\boldsymbol{x}_3 - \boldsymbol{t}||_2 \tag{71}$$

where $\boldsymbol{x}_3$ is the output of the neural network, $\boldsymbol{t}$ is the ground truth from the training data, and $\mathcal{L}$ is the *loss*. Although there are a variety of optimization methods (more methods are discussed in later in this chapter) for determining the weights, the widely accepted approach is a first order gradient descent method [**?**]:

$$\boldsymbol{W}_i^{n+1} = \underbrace{\boldsymbol{W}_i^n}_{\in \mathcal{R}^{n_i \times n_{i-1}+1}} - \underbrace{\alpha \nabla_{\boldsymbol{W}_i} \mathcal{L}}_{\in \mathcal{R}^{n_i \times n_{i-1}+1}} \tag{72}$$

where $i = 1, \ldots, m$ represents the total number of transitions between layers. For the four layer network in Figure 5, there are three transitions meaning back propagation of this network will required differentiating $\mathcal{L}$ with respect to three weight matrices, $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, and $\boldsymbol{W}_3$.

Back propagation of a four-layer fully connected neural network

The loss function we want to minimize is the L_2 norm of the difference between the output of the network and the training data:

$$\mathcal{L} = \frac{1}{2}||\boldsymbol{x}_3 - \boldsymbol{t}||_2^2 = \frac{1}{2}(\boldsymbol{x}_3 - \boldsymbol{t})^\top(\boldsymbol{x}_3 - \boldsymbol{t}) \tag{73}$$

where the explicit equation for the neural network is:

$$\boldsymbol{x}_3 = \psi\Bigg(\boldsymbol{W}_3\phi\Big(\boldsymbol{W}_2\phi\big(\boldsymbol{W}_1\boldsymbol{x}_0\big)\Big)\Bigg). \tag{74}$$

To update the next guess matrix for $\boldsymbol{W}_i$, we need to compute the gradients $\nabla\mathcal{L} = [\frac{\partial\mathcal{L}}{\partial\boldsymbol{W}_1}, \frac{\partial\mathcal{L}}{\partial\boldsymbol{W}_2}, \frac{\partial\mathcal{L}}{\partial\boldsymbol{W}_3}]$. It is important to note that since we are dealing partial derivatives with respect to matrices and vectors, the gradient of the loss function is actually a vector of matrices. The derivatives are obtained by using the chain rule:

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{W}_1} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{x}_3} \circ \frac{\partial\boldsymbol{x}_3}{\partial\boldsymbol{x}_2} \circ \frac{\partial\boldsymbol{x}_2}{\partial\boldsymbol{x}_1} \circ \frac{\partial\boldsymbol{x}_1}{\partial\boldsymbol{W}_1} \circ \boldsymbol{x}_0^\top \tag{75}$$

An explicit computation of backpropagation is shown below.

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_1} = (\boldsymbol{x}_3 - \boldsymbol{t}) \circ \psi'(\boldsymbol{W}_3 \boldsymbol{x}_2) \circ \boldsymbol{W}_3 \phi'(\boldsymbol{W}_2 \boldsymbol{x}_1) \circ \boldsymbol{W}_2 \phi'(\boldsymbol{W}_1 \boldsymbol{x}_0) \circ \boldsymbol{x}_0^\top \tag{76}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_1} = (\boldsymbol{x}_3 - \boldsymbol{t}) \circ \psi'(\boldsymbol{W}_3 \boldsymbol{x}_2) \circ \boldsymbol{W}_3 \phi'(\boldsymbol{W}_2 \boldsymbol{x}_1) \boldsymbol{x}_1^\top \tag{77}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_3} = \underbrace{\underbrace{(\boldsymbol{x}_3 - \boldsymbol{t})}_{\in \mathcal{R}^{n_3}} \circ \underbrace{\psi'(\boldsymbol{W}_3 \boldsymbol{x}_2)}_{\in \mathcal{R}^{n_3}}}_{\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}} \underbrace{\boldsymbol{x}_2^\top}_{\in \mathcal{R}^{1 \times n_2 + 1}} \tag{78}$$

$$\begin{bmatrix} x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & 1 \end{bmatrix}$$

where $\circ$ is the Hadamard product, and the final operation for each partial derivative is an outer product.

Inference from neural networks refers to the process of using a trained model to make predictions or decisions on new input data. During inference, the input data is passed through the network, which performs the sequential matrix-vector operations with the weights $\boldsymbol{W}_i$ obtained from training on the input to generate an output. The output is then used to make a prediction or decision based on the task the network was trained to perform, such as image classification or natural language processing. Inference is an important step in using neural networks for real-world applications, as it allows the model to be applied to new data and provide useful insights or actions.

4.2 Automatic Differentiation

In machine learning, the successful training of complex neural networks and mathematical models hinges upon the seamless interplay between two fundamental processes: optimization, which fine-tunes model parameters to minimize or maximize a specific objective, and backpropagation, which efficiently computes gradients essential for guiding the optimization process. Central to the efficacy of these processes are automatic differentiation tools, which provide a systematic and computationally efficient way to compute gradients of complex functions. Automatic differentiation techniques have emerged as a cornerstone technology, enabling the training of deep learning architectures and other optimization-driven algorithms. This methods-focused exposition delves into the landscape of automatic differentiation, elucidating its pivotal role in the domains of optimization and backpropagation. We explore the underlying principles of automatic differentiation, encompassing both forward and reverse modes, while highlighting prominent automatic differentiation frameworks and their integration within optimization algorithms. By elucidating the symbiotic relationship between automatic differentiation, optimization, and backpropagation, this section aims to equip practitioners with a comprehensive understanding of the techniques that underpin the training of modern machine learning models.

As indicated in the preceding section, deep learning is predicated on the successive matrix-vector operations followed by a non-linear activation function. Back-propagation requires differentiating through the entire network and requires a computational approach to keep track of an arbitrary number of mathematical operations. The fundamental representation in automatic differentiation is the computation graph which serves as a structured visualization of the transformations and computations that occur within a model during its forward pass.

By analyzing the chain of operations encoded within the computational graph, automatic differentiation enables the extraction of derivatives, facilitating efficient gradient computation for optimization algorithms like stochastic gradient descent. This dynamic interplay between computational graphs and automatic differentiation forms the foundation upon which modern optimization-driven machine learning thrives, engendering the ability to train intricate models while efficiently updating their parameters through gradient-based techniques.

6 Deep Reinforcement Learning

4.1 Policy Gradients Introduction

The goal of this assignment is to experiment with policy gradient and its variants, including variance reduction tricks such as implementing reward-to-go and neural network baselines. From a high level, the basic policy gradient algorithm is

1. generate samples (i.e., run the policy)
2. fit a model/estimate the return $J(\theta) = \mathbb{E}[\sum_t r_t] \approx \frac{1}{N} \sum_{i=1}^{N} \sum_t r_{t,i}$
   $\rightarrow$ found in `calculate_q_values`
3. improve the policy (backprop) $\theta \leftarrow \theta + \alpha \nabla \theta J(\theta)$
   $\rightarrow$ found in `MLPPolicyPG.update()`

4.2 Foundations of the Policy Gradient

At the core of any reinforcement learning algorithm is the policy, or internal model of the environment. For deep RL, this policy is represented as a neural network model that maps states $\boldsymbol{s}_t$ and/or observations $\boldsymbol{o}_t$ to actions $\boldsymbol{a}_t$, where t is a particular time step in the trajectory. The policy $\pi_\theta(\boldsymbol{a}_t|\boldsymbol{o}_t)$ is represented as a distribution of possible actions, given a set of observations. This can easily be encoded in PyTorch as a neural network that returns a distribution, as shown in the example code below:

```
class GaussianPolicy(nn.Module):
    def __init__(self, input_size, output_size):
        super(GaussianPolicy, self).__init__()

        # Mean will be a function of the input; std will be fixed (learned) value
        self.mean_fc1 = nn.Linear(input_size, 32)
        self.mean_fc2 = nn.Linear(32, 32)
        self.mean_fc3 = nn.Linear(32, output_size)
        self.log_std =  nn.Parameter(torch.randn(output_size))

    def forward(self, x):
        mean = F.relu(self.mean_fc1(x))
        mean = F.relu(self.mean_fc2(mean))
        mean = self.mean_fc3(mean)

        return distributions.Normal(mean, self.log_std.exp())
```

The next core concept in Deep RL, is the objective function that trains this neural network. As the agent interacts with its environment, we seek to learn about how the agent is rewarded when performing certain actions while being driven to a reward. For now, let us assume that we have a solid objective function that will allow us to adequately quantify the performance of the agent, and that the agent receives a reward at every time step. We now need to incorporate this reward into our policy so that the action distribution of the agent is shifted to those that will return the greatest reward. Here we introduce the Deep RL learning objective function:

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}[r(\tau)] \approx \frac{1}{N}\sum_i \sum_t r(s_{i,t}, a_{i,t}). \tag{79}$$

More specifically, we seek a set of neural weight parameters θ that minimizes Equation 79. The inner term in this equation $r(\tau)$ represents summation of rewards throughout an an entire trajectory, other was known as a rollout in Deep RL.

$$r(\tau) = r(\{s_0, a_0\}, ..., \{s_{T-1}, a_{T-1}\}) = \sum_{t=0}^{T-1} r(s_t, a_t) \tag{80}$$

The policy $\pi_\theta(\tau)$ is trained over an entire rollout τ of length T and attempts to map the as trajectory distribution $p_\theta(\tau)$. The trajectory distribution is a probability distribution over a sequence of states and actions:

$$p_\theta(\tau) = p(\{s_0, a_0\}, ..., \{s_{T-1}, a_{T-1}\}) = p(s_0)\pi_\theta(a_0|s_0)\prod_{t=1}^{T-1} p(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t). \tag{81}$$

In this representation, it expanded as the chain rule of probability as the product of the initial state distribution with the product the probability distributions overall time steps of the policy probability $\pi_t heta(a_t|s_t)$, times the transition probability $p(s_{t+1}|s_t, a_t)$. The core assumption behind this equation is that the process is Markovian (see lec-4 slide 12). In model-free RL, like policy gradients, we do not know the transition probabilities. However, we can effectively sample from the transition probabilities by interacting with the environment.

In training our policy, we would like to predict how well the actions of the agent will perform in the environment by sampling from our trajectory distribution $p_\theta(\tau)$, and estimating the potential reward. More explicitly, we would like to find a set of neural network weight parameters θ that maximizes the total summation of expected rewards over the entire trajectory:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\theta\sim p_\theta(\tau)}\left[\sum_t r(s_t, a_t)\right]. \tag{82}$$

As stated in the title of this section, the purpose of the policy gradient approach is to directly take the gradient of Equation 79:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] \tag{83}$$

$$= \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau \tag{84}$$

Evaluating gradient of the objective in this form requires some manipulation, and will require the use of a "convenient identity".

A convenient identity
This directly follow from the equation of the derivative of the logarithm:

$$p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) = p_\theta(\tau)\frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau)$$

The gradient of our objective can now be expressed as:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[r(\tau)] && (85)\\ &= \int p_\theta \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau && (86)\\ &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\Big[\nabla_\theta \log p_\theta(\tau) r(\tau)\Big] && (87)\end{aligned}$$

Since we still do not know explicitly the trajectory distribution $p_\theta(\tau)$, we need further simplify this form. Here we refer back to Equation 81 and take the log of both sides. This yields:

$$\log p_\theta(\tau) = \log p_\theta(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t). \quad (88)$$

Since the first and third terms do not depend on the θ, the gradient of this function only yields the second term. This means we can re-write our policy gradient in terms of the neural network, independent of the transition policies:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\Big[\nabla_\theta \log p_\theta(\tau) r(\tau)\Big] && (89)\\ &= \mathbb{E}_{\tau\sim \pi_\theta(\tau)}\Big[\big(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\big)\big(\sum_{t=1}^{T} r(s_t, a_t)\big)\Big] && (90)\end{aligned}$$

We need to make one more modification to get the basic form of our policy gradient. Recall from Equation 79 that the objective can be approximated as a double summation over all rollouts, and their subsequent time steps. This means that we can also approximate the gradient of the objective as well:

Policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\Big(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\Big)\Big(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\Big) \quad (91)$$

Using this form of the policy gradient, we can now implement what is referred to as the REINFORCE algorithm.

while True:

1. sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (i.e., run the policy)
2. $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\Big(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\Big)\Big(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\Big)$
3. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$.

The summation of the rewards in Equation 91 is part of the green box in Figure 6 and is where we fit a model to estimate a return. The outer summation overall trajectories is processed in the orange box in Figure 6 and is where sample generation occurs. To improve the policy (blue box), θ is updated according to step 3 above.

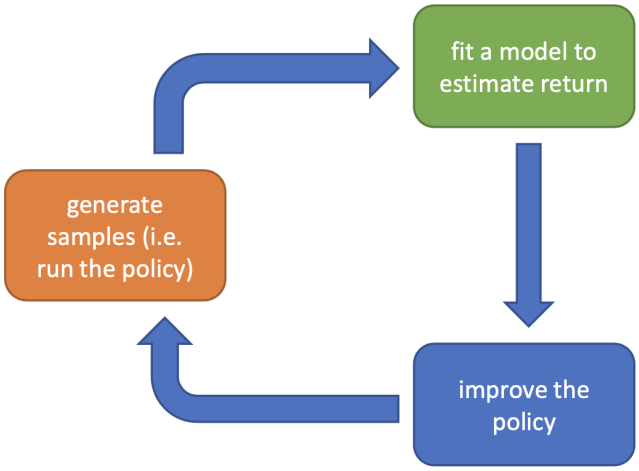


Figure 6: REINFORCE algorithm

The basic idea behind this algorithm is that we want the objective function to capture the probability of reward scaled by the reward from the last action. By iterating over and over within an environment, we begin to bias the policy to produce an action distribution that prioritizes distributions that yield the highest reward (see Figure 7).

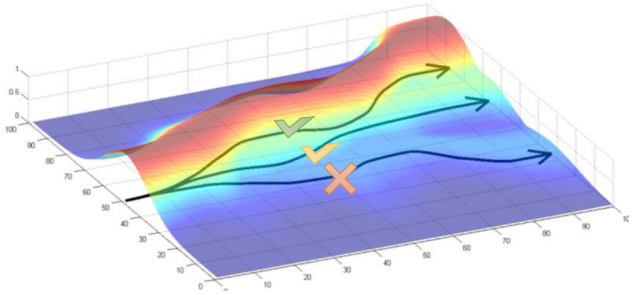


Figure 7: Learning distribution

4.3 Implementation of vanilla policy gradient with PyTorch

On assignment 2, we were required to implement several variations of a policy gradient. This required coding Equation 91 in several ways. Each method will be described below, but here I will describe at a high level how each term of the policy gradient in the current form can be computed with PyTorch.

We seek to encode a PyTorch loss function that encapsulates all of terms in Equation 91. We will first examine the first inner summation over the log probabilities:

$$\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}).$$

Since we have a neural network (policy $\pi_\theta(a_t|s_t)$) that outputs a distribution of potential actions from inputted observations (see https://pytorch.org/docs/stable/distributions.html):

```
loss = -action_distribution.log_prob(action)
```

The second summation of the rewards over an entire rollout is

$$\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \tag{92}$$

and is computed in the assignment as

```
# a function that returns the summation of rewards over an entire rollout
return = lambda rewards: (np.ones(len(rewards)) * sum(rewards)).tolist()
```

Assume for now that the advantage is equivalent to the reward. We can add this to the loss function and implement the REINFORCE algorithm as:

```
# step 1: generate samples
action_distribution = self.forward(observations)
action = action_distribution.sample()  # in hw performed upstream from MLP_policy
next_state, reward = env.step(action)  # in hw performed upstream from MLP_policy

# step 2: fit a model to estimate the return
loss = -action_distribution.log_prob(action) * reward

# step 3: improve policy (backprop)
loss.backward()
```

4.4 Variance reduction: reward-to-go

In this assignment, we implemented the policy gradient (Equation 91) in a variety of ways. We discussed a number of issues that affected the vanilla implementation, and specifically the variance of the action distribution provided by the policy (see Figure 7 for a visual understanding). The additional methods that we sought to implement mainly focused on reducing this variance.

One way to reduce the variance of the policy gradient is to exploit **causality**: the notion that the policy cannot affect rewards in the past. This yields the following modified objective, where the sum of rewards here does not include the rewards achieved prior to the time step at which the policy is being queried. This sum of rewards is a sample estimate of the Q function, and is referred to as the "reward-to-go."

Policy gradient with reward-to-go:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})\right) \tag{93}$$

Since the only thing that has changed between Equations 91 and 93 is the last term, the only thing that is changed in the PyTorch implementations is the computation of the reward:

Computing reward-to-go:

```
# computing the summation of rewards over an entire rollout, with reward-to-go
def return(rewards):
    cumsum_return = np.ones(len(rewards)) * sum(rewards)
    for i in range(1, len(rewards)):
        # subtract off the previous time step's reward
        cumsum_return[i] -= rewards[i - 1]
    return cumsum_return.tolist()
```

4.5 Variance reduction: discounting

Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus more on the rewards that are closer in time, and less on the rewards that are further in the future. This can also be thought of as a means for reducing

variance (because there is more variance possible when considering futures that are further into the future). We saw in lecture that the discount factor can be incorporated in two ways, as shown below.

The first way applies the discount on the rewards from full trajectory:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t'=1}^{T}\gamma^{t'-1}r(s_{i,t'},a_{i,t'})\right) \tag{94}$$

which can be applied in PyTorch by modifying the reward implementations from above:

Computing the discounted reward:

```
# a function that returns the summation of discounted rewards over an entire rollout
def _discounted_return(rewards: list) -> list:
    total_discounted_return = sum([self.gamma ** t * r for t, r in enumerate(rewards)])
    list_of_discounted_returns = np.ones(len(rewards)) * total_discounted_return

    return list_of_discounted_returns.tolist()
```

and the second way applies the discount on the "reward-to-go:"

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\left(\sum_{t'=1}^{T}\gamma^{t'-1}r(s_{i,t'},a_{i,t'})\right). \tag{95}$$

Computing the discounted reward-to-go:

```
# a function that returns the summation of discounted reward-to-gos
# over an entire rollout
def _discounted_cumsum(rewards: list) -> list:
        all_discounted_cumsums = []

        # for loop over steps (t) of the given rollout
        for start_time_index in range(len(rewards)):

            # 1) create an array of indices (t'): goes from t to T-1
            indices = np.arange(start_time_index, len(rewards))

            # 2) create an array where the entry at each index (t') is gamma^(t'-t)
            discounts = np.power(self.gamma, indices - start_time_index)

            # 3) create a list where the entry at each index (t')
            #    is gamma^(t'-t) * r_{t'}
            discounted_rtg = rewards[indices] * discounts

            # 4) calculate a scalar: sum_{t'=t}^{T-1} gamma^(t'-t) * r_{t'}
            sum_discounted_rtg = np.sum(discounted_rtg)

            # appending each of these calculated sums into the list to return
            all_discounted_cumsums.append(sum_discounted_rtg)
        list_of_discounted_cumsums = np.array(all_discounted_cumsums)
        return list_of_discounted_cumsums.tolist()
```

4.6 Variance reduction: baseline

Another variance reduction method is to subtract a baseline (utilizing the "convenient identity", and that is a constant with respect to τ) from the sum of rewards:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\big[r(\tau) - b\big] = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta} \nabla_\theta \log p_\theta(\tau)\big[r(\tau) - b\big]. \tag{96}$$

This leaves the policy gradient unbiased (in expectation, not variance) because:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\big[\nabla_\theta \log p_\theta(\tau)\cdot b\big] &= 0 && (97)\\
&= \int p_\theta(\tau)\nabla_\theta p_\theta(\tau) b d\tau && (98)\\
&= \int \nabla_\theta p_\theta(\tau) b d\tau && (99)\\
&= b\nabla_\theta \int p_\theta \int p_\theta(\tau) d\tau && (100)\\
&= b\nabla_\theta 1 && (101)\\
&= 0 && (102)
\end{aligned}$$

In this assignment, we will implement a value function V^π_ϕ which acts as a state-dependent baseline. This value function will be trained to approximate the sum of future rewards starting from a particular state:

$$V^\pi_\phi \approx \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\big[r(s_{t'}, a_{t'})|s_t\big], \tag{103}$$

so the approximate policy gradient now looks like this:

Policy gradient with baseline:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(s_{i,t}|a_{i,t})\Big(\big(\sum t' = t^{T-1}\gamma^{t'-t} r(s_{i,t'}, a_{i,t'})\big) - V^\pi_\phi(s_{i,t})\Big) \tag{104}$$

The right-most hand term $\big(\big(\sum t' = t^{T-1}\gamma^{t'-t} r(s_{i,t'}, a_{i,t'})\big) - V^\pi_\phi(s_{i,t})\big)$ is practically computed separately. First, a discount is applied to reward, and then the reward, or reward-to-go, is passed on as the "Q-value".

Estimating Q values:

```
def calculate_q_vals(self, rewards_list):
    if not self.reward_to_go:
        discounted_return = [self._discounted_return(reward)
                             for reward in rewards_list]
        q_values = np.concatenate(discounted_return)
    else:
        q_values = np.concatenate([self._discounted_cumsum(i) for i in rewards_list])
    return q_values
```

The value function in Equation 103 can be "learned" by mapping the observed states to the rewards, and returning the summation of the expectations. For this estimate, it is important to normalize the observations prior to training the multi-layer perceptron. An implementation is listed below:

Learning the value function:

```
# setting up the MLP in  MLPPolicy
if nn_baseline:
    self.baseline = ptu.build_mlp(
        input_size=self.ob_dim,
        output_size=1,
        n_layers=self.n_layers,
        size=self.size,
    )
    self.baseline.to(ptu.device)
    self.baseline_optimizer = optim.Adam(
        self.baseline.parameters(),
        self.learning_rate,
    )

...

# within the update function
if self.nn_baseline:
    # normalize rewards
    targets = ptu.from_numpy((q_values - np.mean(q_values)) / (np.std(q_values) + 1e-8))
    baseline_prediction = self.baseline(observations).squeeze()
    baseline_loss = self.baseline_loss(baseline_prediction, targets)

    self.baseline_optimizer.zero_grad()
    baseline_loss.backward()
    self.baseline_optimizer.step()

    train_log = {'Baseline Loss': ptu.to_numpy(baseline_loss), }
```

In our implementation, the actual estimation of `values` from `nn_baseline()` is computed in `PGAgent` while estimating the *advantage*. *Recall* that the advantage is defined as the difference between the `q_values - values`.

Estimating advantage:

```
def estimate_advantage(self,
                       obs: np.ndarray,
                       rews_list: np.ndarray,
                       q_values: np.ndarray,
                       terminals: np.ndarray):
    if self.nn_baseline:
        values_unnormalized = self.actor.run_baseline_prediction(obs)
        assert values_unnormalized.ndim == q_values.ndim

        # raise values
        values = values_unnormalized * np.std(q_values) + np.mean(q_values)
        advantages = q_values - values
    else:
        advantages = q_values.copy()

    if self.standardize_advantages:
        advantages = (advantages - np.mean(advantages)) / (np.std(advantages) + 1e-8)

    return advantages
```

4.7 Variance reduction: generalized advantage estimation:

The quantity $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{t'}, a_{t'})\right) - V_\phi^\pi(s_t)$ from the previous policy gradient expression (removing the i index for clarity) can be interpreted as an estimate of the advantage function:

$$A^\pi(s_t, a_t) = \left(\sum t' = t^{T-1} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'})\right) - V_\phi^\pi(s_{i,t}) \tag{105}$$

$$= Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{106}$$

where $Q^\pi(s_t, a_t)$ is estimated using Monte Carlo returns and $V^\pi(s_t)$ is estimating using the learned value function V_ϕ^π. We can further reduce variance by also using V_ϕ^π in place of the Monte Carlo returns to estimate the advantage function as

$$A^\pi(s_t, a_t) \approx \delta_t = r(s_t, a_t) + \gamma V_\phi^\pi(s_{t+1}) - V_\phi^\pi(s_t), \tag{107}$$

with the edge case $\delta_{T-1} = r(s_{T-1}, a_{T-1}) - V_\phi^\pi(s_{T-1})$. However, this comes at the cost of introducing bias to our policy gradient estimate, due to modeling errors in V_ϕ^π. We can instead use a combination of n-step Monte Carlo returns and V_ϕ^π to estimate the advantage function as:

$$A_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) + \gamma^n V_\phi^\pi(s_{t+n+1}) - V_\phi^\pi(s_t). \tag{108}$$

Increasing n incorporates the Monte Carlo returns more heavily in the advantage estimate, which lowers bias and increases variance, while decreasing n does the opposite. Note that $n = T - t - 1$ recovers the unbiased but higher variance Monte Carlo advantage estimate used in Equation 104, while $n = 0$ recovers the lower variance but higher bias advantage estimate δt.

We can combine multiple n-step advantage estimates as an exponentially weighted sum, which is known as the generalized advantage estimator (GAE). Let $\lambda \in [0, 1]$. Then we define:

$$A_{GAE}^\pi(s_t, a_t) = \frac{1 - \lambda^{T-t-1}}{1 - \lambda} \sum_{n=1}^{T-t-1} \lambda^{n-1} A_n^\pi(s_t, a_t), \tag{109}$$

where $\frac{1-\lambda^{T-t-1}}{1-\lambda}$ is a normalizing constant. Note that the higher λ exmphasizes advantage estimates with higher values of n, and a lower λ does the opposite. In the infinite horizon case ($T = \infty$), we can show that

$$A_{GAE}^\pi(s_t, a_t) = \sum_{t'=t}^{T-1} (\gamma\lambda)^{t'-t} \delta_{t'}, \tag{110}$$

which serves as a way we can efficiently implement the generalized advantage estimator since we can recursively compute:

$$A_{GAE}^\pi(s_t, a_t) = \delta_t + \gamma\lambda A_{GAE}^\pi(s_{t+1}, a_{t+1}) \tag{111}$$

Computationally, we can implement this recursion as:

Computing the generalized advantage estimation (GAE) → `estimate_advantage()`:

```
if self.nn_baseline:
        values_unnormalized = self.actor.run_baseline_prediction(obs)
        assert values_unnormalized.ndim == q_values.ndim

        values = values_unnormalized * np.std(q_values) + np.mean(q_values)

        if self.gae_lambda is not None:
            values = np.append(values, [0])

            rews = np.concatenate(rews_list)

            batch_size = obs.shape[0]
            advantages = np.zeros(batch_size + 1)

            for i in reversed(range(batch_size)):
                if terminals[i] == 1:
                    advantages[i] = rews[i] - values[i]
                else:
                    advantages[i] = rews[i] + self.gamma * values[i + 1] - values[i]
                    # equation 33 above
                    advantages[i] += self.gamma * self.gae_lambda * advantages[i + 1]

            advantages = advantages[:-1]

        else:
            advantages = q_values - values

    else:
        advantages = q_values.copy()
```

4.8 Actor-critic algorithms

To improve on this framework, we would like to make some modifications to the format above. Rather than strictly summing up the rewards, we will fit a separate neural network to estimate either Q^π, V^π, A^π. This means that we will have two neural networks within the policy gradient algorithm (Figure 6), specifically in the green box "fit a model to the estimate return":

- a policy network (actor that return actions)
- a value function network (critic that criticizes or baselines)

The reason why we choose to approximate the value function is because we can the value function in the next state can be written as:

$$V^\pi(s_{t+1}) = r(s_t, a_t) + \sum_{t'=t+1}^{T} E_{\pi_\theta}\left[r(s_{t'}|s_t, a_t)\right]$$

additionally, since we know the current reward, we can separate it out from the expectation of the quality function:

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(s_{t'}, a_{t'}|s_t, a_t)\right] \quad \rightarrow \quad r(s_t, a_t) + \sum_{t'=t+1}^{T} E_{\pi_\theta}\left[r(s_{t'}, a_{t'})|s_t, a_t\right]$$

where all $s_{t'}$ after time step t are random variables. Now we can see that the second component of the quality function is just the value function at the next time step. We know that the expectation of all rewards in the future is perfectly summarized with by the expected value of $V^\pi(s_{t+1})$, where $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$:

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + E_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)}\left[V^\pi(s_{t+1})\right]. \tag{112}$$

With this new approximation to the quality function, we can arrive at advantage function for the actor-critic algorithms:

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t) \tag{113}$$

which means we can express the advantage and quality functions in terms of just the value function.

The actor critic algorithm
`while not done do:`

1. sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$ (run the policy in simulation or on a robot)
2. fit $\hat{V}^\pi_\phi$ to the sampled reward sums
3. evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)|s_{i_t})\hat{A}^\pi(s_i, a_i)$
5. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
6. return to 1

4.9 Q-learning

Recall from Equation 103 (also listed below as a reminder) is the expected reward given being in some state s_t provided that the best action is taken $r(s_{t'}, a_{t'})$. In other words, the value function attempts to quantify the value of being in particular state, *assuming that the agent takes the best action.*

$$V^\pi_\phi \approx \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\left[r(s_{t'}, a_{t'})|s_t\right],$$

Q-learning attempts generalize the value function by quantifying the *quality* of being in a particular state s_t, for all possible actions a_t. The quality of being in a state now is my expected current reward in addition to all of the future rewards for being in the next state $s_{t'}$. We can write this as a sum of probabilities (since this is a Markov-Decision Process) over all possible next states $s_{t'}$

$$Q^\pi_\phi \approx \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\left[r(s_{t'}, s_t, a_t) + \gamma V^\pi_\phi(s_{t'})\right] = \sum_{s_{t'}} P(s_{t'}|s_t, a_t) \cdot \left(r(s_{t'}, s_t, a_t) + \gamma V^\pi_\phi(s_{t'})\right) \tag{114}$$

In Equation 114, the value function that is being used can be computed as

$$V^\pi_\phi(s) = \arg\max_{a_t} Q(s_t, a_t) \tag{115}$$

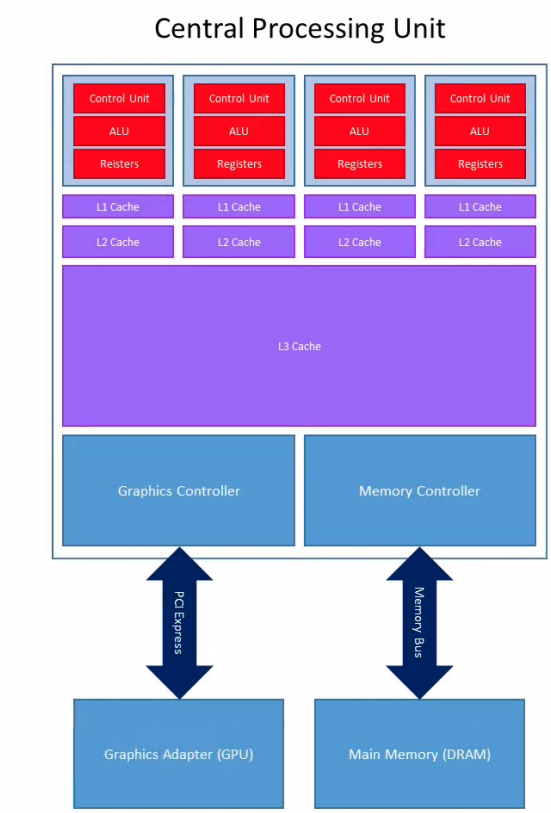
Section 5: Computing

6 GPU hardware

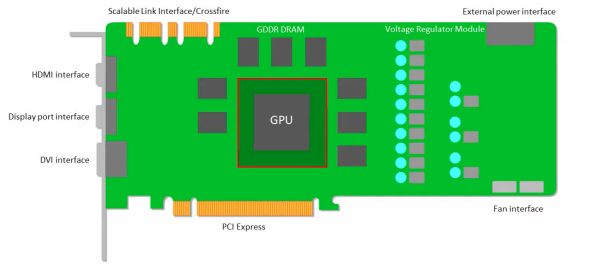
5.1 CPU + graphics card

The CPU is responsible for initiating computation on the GPU and for transferring data to and from the graphics adapter (see Figure 8a), which is a specialized device designed to handle graphics operations that can be massively parallelized (see Figure 8b). This is accomplished via a high-speed bus, such as PCIe. The graphics adapter has its own memory, which acts as a buffer between the CPU and display, and is commonly referred to as VRAM, frame buffer, or GDDR DRAM. The size of the VRAM is typically 8-10GB, but can be much larger. Although the VRAM is similar to the main memory DRAM on the CPU, it is highly optimized for bandwidth, with more channels, lower power consumption, and less heat generation, compared to traditional DRAM that is optimized for low latency.

Additionally, external inputs include a series of voltage regulator modules, which modulates the required power for graphics card from the power supply, thermal fans for cooling, as well as a set of pins that attach the card to the PCI express bus. It may also have pins that will all the connection of multiple GPUs to work together. External outputs of the graphics card may include HDMI, display port, or DVI interfaces.



(a) Typical architecture of a multi-core CPU. Each core contains 1) control unit, 2) arithmetic-logic units (ALU), 3) vector registers. Memory buses transfer of information between the cache (SRAM) and main memory (DRAM). The PCI Express (high speed bus) transfers information between the CPU and graphics card.



(b) A high level representation of a graphics card hardware architecture. The graphics card can contain multiple inputs, such as information flowing through the PICe bus, power flowing through the voltage regulator modules, heat control via the fans, as well as information flowing via scalable link interfaces between connected GPUs.

Figure 8: Relationship between CPU and graphics card.

At center of the graphics card is the graphics processing unit (GPU). A modern GPU is composed of many cores (as shown in Figure 9) called *streaming multi-process* (SM) in Nvidia GPUs and *compute units* (CU) in AMD GPUs. **Each core on the GPU can typically run on the order of a thousand threads** (see [**?**] pg 4) via **single-instruction multiple-thread (SIMT)** corresponding to a the kernel that has been launched to run on the GPU. The threads executing on a single core can communicate through a scratchpad memory and synchronize using fast barrier operations via a controller and L1 cache.

One level below the SM or CU is the processing element (PE). In Nvidia chips this maybe either a CUDA or Tensor cores, each with their own architecture and purpose. On an AMD GPU, this PE is referred to as a **shader**. A detailed explanation of the types of operations that can be performed on a Nvidia PE can be found here. In general, a PE may contain the following components:

- **Registers**: A small amount of on-chip memory that is used to hold data and intermediate results during instruction execution.
- **ALUs**: Specialized units that can perform arithmetic and logical operations on data stored in registers.
- **Load/Store Units**: These are units that handle memory access operations, such as loading data from memory into registers or storing data from registers back to memory.
- **Branching Units**: These are units that handle conditional branching operations, which determine which instruction to execute next based on a specified condition.
- **Control Units**: These are units that manage instruction execution, such as fetching instructions from memory and decoding them into the appropriate operations.

On NERSC Perlmutter machine, the NVIDIA A100 `8.0` GPU contains the following specs. A single SM on a NVIDIA consists of:

- 64 FP32 cores for single-precision arithmetic operations
- 32 FP64 cores for double-precision arithmetic operations
- 64 INT32 cores for integer math
- 4 mixed-precision Third-Generation Tensor Cores supporting half-precision (fp16), __nv_bfloat16, tf32, sub-byte and double precision (fp64) matrix arithmetic
- 16 special function units for single-precision floating-point transcendental functions
- 4 warp schedulers

Nvidia's GPUs operate off of a similar SIMD structure as shown for CPUs above. Modern CPUs typically are built as 64-bit processors, meaning that each register within the vector register are 64 bits wide and are usually capable of holding 8 doubles. (Note when referring to Fig. **??**, column represents a vector register that can hold four double-precision 64-bit floats.) However, since GPUs typically prioritize speed over precision, the register width for floating-point operations (each box) are typically 32 bits wide. As shown above, it is not uncommon for Nvidia's PEs to contain a mix of both 32-bit and 64-bit PEs, allowing the user to specify double-precision floating point operations when precision is prioritized over speed.

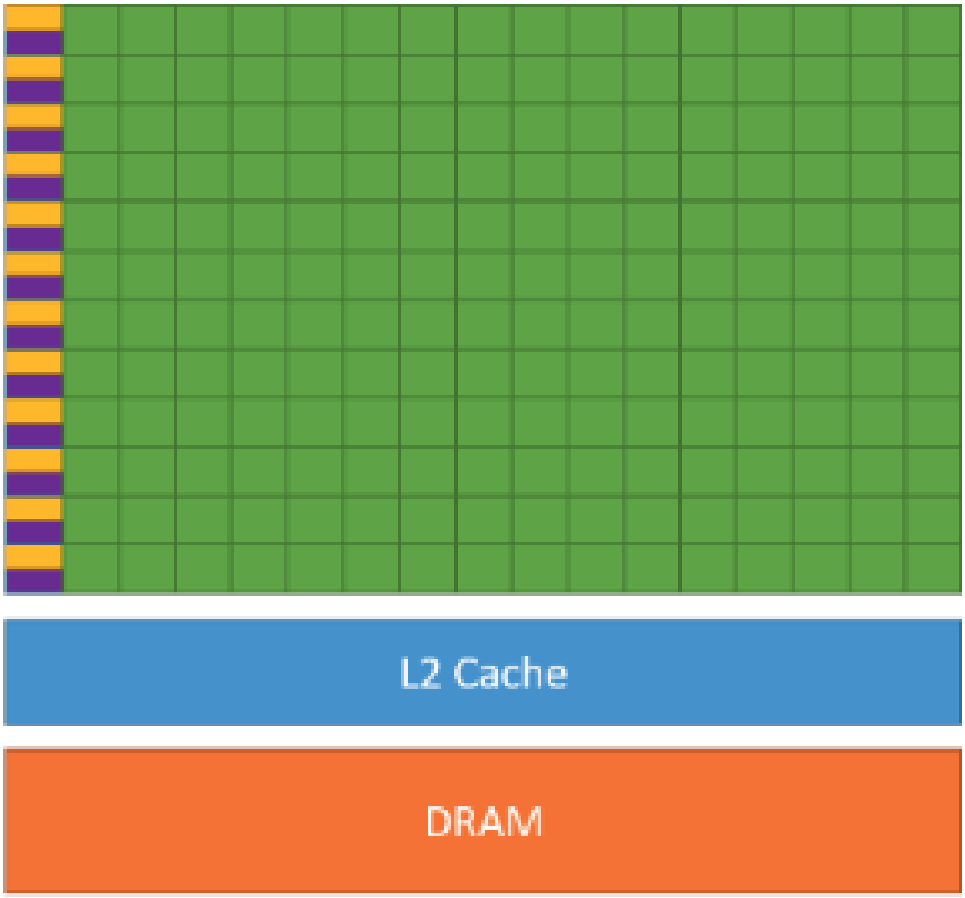


Figure 9: A high-level representation of a Nvidia GPU. The GPU consists of hundreds of light-weight cores (streaming multi-processors), each with its own controller (orange), L1 cache (purple) and processing elements (green) (for Nvidia chips, CUDA or Tensor cores).

NOTE:
The term "core" is used differently depending on the context. At highest level, a Nvidia GPU is typically composed of many streaming multi-processors (SMs), each of which can execute multiple threads in parallel. Each SM can be thought of as a small processor with its own set of execution resources, such as registers, caches, and functional units. In this context, "core" would refer to the entire SM, which can compute multiple threads in parallel.

However, within each SM, there are multiple smaller processing elements, which NVIDIA refers to as "CUDA cores". Each CUDA core is responsible for executing a single thread in the SIMT execution model, as described earlier. Therefore, in this context, "core" would refer to the individual processing elements within the SM that execute the SIMT program.

GPU hardware software abstraction

When executing parallel code, the terminology shifts to software abstraction. At core of a parallizable program is an operation that can be independently performed on different data. A single execution on subset of these data is called a **thread**. Similar to AVX instructions for CPU vectorization, GPUs can compute parallel operations using single-instruction multiple-threads (SIMT) that operates in lockstep with other threads. For Nvidia GPUs, the number of threads is fixed as 32 and is grouped as a **warp**. A single instruction can be loaded into a warp to perform massively parallized computation. Many warps can exist on a single core, which can also simultaneously run the same or different kernels in parallel. The core, referred to by Nvidia as a streaming multi-processor (SM) and by AMD as computing unit, can contain multiple warps. Additional SMs within the GPU can provide even more parallelization, and even more can be added by tethering multiple GPUs in parallel. This hierachy and relationship between hardware and software terminology is shown in Figure 10.

Table 1: Hardware to software abstraction

Hardware	Software abstraction
instruction	kernel
processing element (PE)	thread
32 PEs in lockstep (SIMT)	warp
# group of 32 PEs	threadblocks
	grid
streaming multi-processor (SM)	

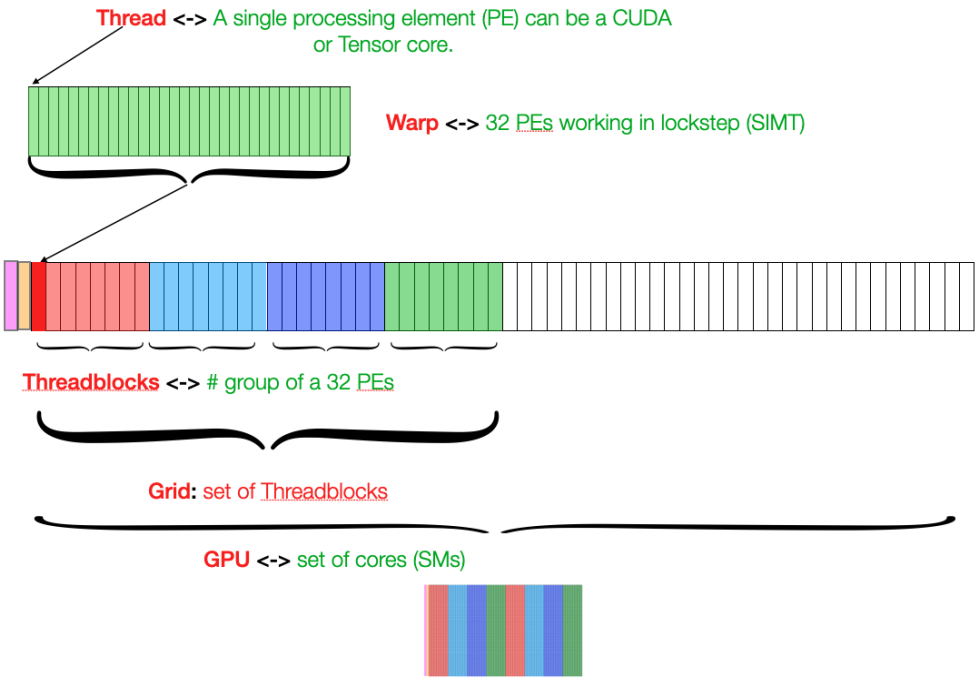


Figure 10: A visual representation of The relationship between hardware and software terminology. The red reprensents the software abstractions and the green represents the hardware terminology.

The number of SMs in a Nvidia GPU and the number of PEs in each SM can vary depending on the specific GPU model and architecture. Nvidia groups 32 PEs as a **warp** of threads or **single-instruction multiple-threads** (SIMT). For Perlmutter, there are 108 SMs each containing 64 warps, meaning that there are about 200,000+ ways of parallelism (108*64*32) per GPU.

A CUDA core differs from a Tensor core in that CUDA cores are general-purpose processing units that can perform arithmetic and logic operations on floating-point and integer data types. CUDA cores are used for a wide range of tasks, including graphics rendering, scientific simulations, and data processing.

Tensor cores, on the other hand, are specialized hardware units that are designed specifically for performing matrix operations, which are a fundamental building block of many machine learning algorithms. Tensor cores are capable of performing mixed-precision matrix multiplication and accumulation operations with high throughput and low latency, which makes them well-suited for accelerating deep learning workloads.

Perlmutter GPU specifications

NERSC's Perlmutter super computer utilizes 4 NVIDIA Ampere A100 GPUs which are powerful computing units designed for massively-parallelized computing and artificial intelligence applications. Here are the specifications of the A100 GPU:

Query	**Result**	**Command**
GPU	NVIDIA A100-SXM4-80GB	`nvidia-smi --query-gpu=name --format=csv`
SM's	108	see number of streaming multiprocessors (SMs)
max threads per SM	2048	see max number of threads per SM
max threads per GPU	221184 (=108×2048)	multiply the two above
maximum x-dim of thread blocks grid	$2^{31}-1$	see Table 15 here
cores	≥6,912	see total number of cores
GPU Memory	81920 MiB (~86GB)	`nvidia-smi --query-gpu=memory.total --format=csv`

Table 2: Specifications of the NVIDIA Ampere A100 GPU and corresponding commands to check.

number of streaming multiprocessors (SMs)

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);

for (int i = 0; i < deviceCount; ++i) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, i);
    std::cout << "Device " << i << " has " << deviceProp.multiProcessorCount << " SMs\n";
}
```

max number of threads per SM

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);

for (int i = 0; i < deviceCount; ++i) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, i);
    std::cout << "Device " << i << " has maximum threads per SM: " << deviceProp.
maxThreadsPerMultiProcessor << "\n";
}
```

total number of cores

The best way to see the cores within a single streaming multiprocessor (SM) on an NVIDIA GPU is to look up the major revision and the minor revision of the GPU with the following code:

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);

for (int i = 0; i < deviceCount; ++i) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, i);
    std::cout << "Device " << i << ":\n";
    std::cout << "  deviceProp.major: " << deviceProp.major << "\n";
    std::cout << "  deviceProp.minor: " << deviceProp.minor << "\n";
}
```

Running this code on an NVIDIA A100 revealed that it has a major revision of `8` and a minor revision of `0`. Now, we just need look up the specs for NVIDIA `8.0` GPUs here and find that a single SM on a NVIDIA `8.0` GPU consists of:

- 64 FP32 cores for single-precision arithmetic operations
- 32 FP64 cores for double-precision arithmetic operations
- 64 INT32 cores for integer math
- 4 mixed-precision Third-Generation Tensor Cores supporting half-precision (fp16), __nv_bfloat16, tf32, sub-byte and double precision (fp64) matrix arithmetic
- 16 special function units for single-precision floating-point transcendental functions
- 4 warp schedulers

We can run performance analysis on any parallelized code by using Nvidia's software Nsight Systems (download Nsight Systems 2023.1.1 (macOS Host)).

nsys profile ./gpu -n 1000000 -s 1