# THEORETICAL COMPUTER SCIENCE: A COMPLETE GUIDE

COMPUTER SCIENCE

AUGUST 30, 2023

E/EA THOMPSON(THEY/THEM),
PHYSICS AND MATH HONORS

*Solo Pursuit of Learning*

# Contents

# Part I

# Comp Phys 381

# Chapter 1

# Root Finding Methods

## 1.1.0 Bisection Method

**Process 1.1.1.** *To find the roots of a function $f$ we first choose a pair of initial values $x_B$ and $x_U$ such that $f(x_B) < 0$ and $f(x_U) > 0$. Then we perform the following iterative procedure:*

1. *Define $x_C = \frac{x_B + x_U}{2}$, and evaluate $f(x_C)$.*
2. *If $f(x_C) > 0$ set $x_U = x_C$*
3. *If $f(x_C) < 0$ set $x_B = x_C$*
4. *Repeat until $|f(x_C)|$ is less than some chosen tolerance, $\epsilon$.*

## 1.2.0 Newton-Raphson Method

**Process 1.2.1.** *Consider a real valued function $f$. Note that the Taylor expansion of $f$ centered at a point $x_0$ and evaluated at $x_0 + \epsilon$ is*

$$f(x_0 + \epsilon) = f(x_0) + f'(x_0)\epsilon + \frac{1}{2}f''(x_0)\epsilon^2 + ... = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)\epsilon^n}{n!} \tag{1.2.1}$$

*With the NR Method $x_0$ is the current estimate for the root of our function. We now truncate the series to terms linear in $\epsilon$:*

$$f(x_0 + \epsilon) = f(x_0) + f'(x_0)\epsilon + O(\epsilon^2) \tag{1.2.2}$$

*where $O(\epsilon^2)$ indicates that the terms of order $\epsilon^2$ and higher are omitted. We set $f(x_0 + \epsilon) = 0$. This gives*

$$f(x_0) + f'(x_0)\epsilon = 0 \tag{1.2.3}$$

*from which we obtain*

$$\epsilon_0 = -\frac{f(x_0)}{f'(x_0)} \tag{1.2.4}$$

*setting $\epsilon = \epsilon_0$. We then let $x_1 = x_0 + \epsilon_0$ and calculate a new $\epsilon_1$. We extend this process and define it recursively as*

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.2.5}$$

*for an indexing ineger $n \geqslant 0$.*

# Chapter 2

# Non-Linear ODEs

**Remark 2.0.1.** In this chapter considered the differential equation described by

$$\frac{dx}{dt} = y, \quad \frac{dy}{dt} = f(x, y, t) \tag{2.0.1}$$

## 2.1.0 Euler Method

**Process 2.1.1.** *We first choose initial conditions $x(0) = x_0$ and $y(0) = y_0$. Consider the Taylor series expansion of a function $g$ about $a + h$ centered at $a$:*

$$f(a + h) = f(a) + hf'(a) + \frac{h^2}{2!}f''(a) + O(h^3) \tag{2.1.1}$$

*We apply this to $x(t)$ and $y(t)$ to obtain*

$$x(t + \Delta t) = x(t) + \Delta t\frac{dx(t)}{dt} + O(\Delta t^2) \tag{2.1.2}$$

$$y(t + \Delta t) = y(t) + \Delta t\frac{dy(t)}{dt} + O(\Delta t^2) \tag{2.1.3}$$

*where we omit terms of order $\Delta t^2$. We shall vary $t$ steps discretely and label $\Delta t = t_{n-1} - t_n$, so our equations of motion become*

$$x_{n+1} = x_n + y_n\Delta t \tag{2.1.4}$$

$$y_{n+1} = y_n + f(x_n, y_n, t_n)\Delta t \tag{2.1.5}$$

*using Euler's Method for approximating integrals. In particular, for Euler's method we take*

$$g(b) - g(a) = \int_a^b \frac{dg}{dt}dt \approx \sum_{i=1}^{n} \frac{dg(t_i)}{dt}\Delta t, t_1 = a, t_n = b - \Delta t \tag{2.1.6}$$

## 2.2.0 Trapezoid Rule

**Process 2.2.1.** *For the trapezoid rule we approximate an integral as follows*

$$(b-a)\left[\frac{\frac{dg(a)}{dt}+\frac{dg(b)}{dt}}{2}\right] \approx \int_a^b \frac{dg}{dt}dt = g(b)-g(a) \tag{2.2.1}$$

*Then, applying this to our DE we obtain the iterative equation*

$$x_{n+1} \approx x_n + \frac{\Delta t}{2}\left(y_n + y_{n+1}\right) \tag{2.2.2}$$

*where $\Delta t = t_{n+1} - t_n$. We then approximate $y_{n+1}$ using Euler's Method*

$$y_{n+1} \approx y_n + \Delta t f(x_n, y_n, t_n) \tag{2.2.3}$$

*Substituting this back into our previous approximation for $x_{n+1}$ we have*

$$x_{n+1} \approx x_n + \frac{\Delta t}{2}\left[y_n + \left(y_n + \Delta t f(x_n, y_n, t_n)\right)\right] \tag{2.2.4}$$

*We apply this again to $y_{n+1}$ to obtain the approximation*

$$y_{n+1} \approx y_n + \frac{\Delta t}{2}\left[f(x_n, y_n, t_n) + \delta t f(x_{n+1}, y_n + \Delta t f(x_n, y_n, t_n), t_{n+1})\right] \tag{2.2.5}$$

*This pair of iterative equations constitute the application of the trapezoid rule to solving our DE.*

## 2.3.0 Runge-Kutta

**Process 2.3.1 (Second Order).** *We first carry out a Taylor expansion of $\frac{dx(t)}{dt}$ about the mid-points of our interval, $\Delta t/2$:*

$$\frac{dx(t)}{dt} = \frac{dx(\Delta t/2)}{dt} + \frac{d^2x(\Delta t/2)}{dt^2}(t-\Delta t/2) + O\left(\frac{\Delta t^2}{2}\right) \tag{2.3.1}$$

*We use this expansion to approximate the following integral:*

$$\int_0^{\Delta t} \frac{x(t)}{dt}dt \approx \frac{dx(\Delta t/2)}{dt}\Delta t + \frac{d^2x(\Delta t/2)}{dt^2}\int_0^{\Delta t}(t-\Delta t/2)dt = \frac{dx(\Delta t/2)}{dt}\Delta t \tag{2.3.2}$$

*Then, the rule to update $x$ in an algorithmic notation is given by*

$$x_{n+1} \approx x_n + y_{n+1/2}\Delta t + O\left(\frac{\Delta t^2}{2}\right) \tag{2.3.3}$$

*and applying the Taylor expansion to $y_{n+1/2}$ we have*

$$y_{n+1/2} \approx= y_n + f(x_n, y_n, t_n)\frac{\Delta t}{2} + O\left(\frac{\Delta t^2}{2}\right) \tag{2.3.4}$$

*Substituing into our $x_{n+1}$ rule we have*

$$x_{n+1} \approx x_n + \left(y_n + f(x_n, y_n, t_n)\frac{\Delta t}{2}\right)\Delta t \tag{2.3.5}$$

*The rule for updating $y_n$ is given by*

$$y_{n+1} = y_n + f(x_{n+1/2}, y_{n+1/2}, t_{n+1/2})\Delta t \tag{2.3.6}$$
$$y_{n+1/2} = y_n + f(x_n, y_n, t_n)\frac{\Delta t}{2} \tag{2.3.7}$$
$$x_{n+1/2} = x_n + y_n\frac{\Delta t}{2} \tag{2.3.8}$$

**Process 2.3.2 (Fourth Order).** *In implementing we take a series of approximations for $x_n$ and $y_n$ then take a weighted average of the result. In particular, we take the approximations*

$$k_{1x,n} = \Delta t y_n \tag{2.3.9}$$
$$k_{1y,n} = \Delta t f(x_n, y_n, t_n) \tag{2.3.10}$$
$$k_{2x,n} = \Delta t\left(y_n + \frac{k_{1y,n}}{2}\right) \tag{2.3.11}$$
$$k_{2y,n} = \Delta t f\left(x_n + \frac{k_{1x,n}}{2}, y_n + \frac{k_{1y,n}}{2}, t_n + \Delta t/2\right) \tag{2.3.12}$$
$$k_{3x,n} = \Delta t\left(y_n + \frac{k_{2y,n}}{2}\right) \tag{2.3.13}$$
$$k_{3y,n} = \Delta t f\left(x_n + \frac{k_{2x,n}}{2}, y_n + \frac{k_{2y,n}}{2}, t_n + \Delta t/2\right) \tag{2.3.14}$$
$$k_{4x,n} = \Delta t\left(y_n + k_{3y,n}\right) \tag{2.3.15}$$
$$k_{4y,n} = \Delta t f\left(x_n + k_{3x,n}, y_n + k_{3y,n}, t_n + \Delta t\right) \tag{2.3.16}$$
$$x_{n+1} = x_n + \frac{k_{1x,n} + 2k_{2x,n} + 2k_{3x,n} + k_{4x,n}}{6} \tag{2.3.17}$$
$$y_{n+1} = y_n + \frac{k_{1y,n} + 2k_{2y,n} + 2k_{3y,n} + k_{4y,n}}{6} \tag{2.3.18}$$

# Chapter 3

# Numerical Fourier Analysis

## 3.1.0 Fourier Series

**Definition 3.1.1 (Fourier Series).** *Any periodic function $f(t)$ with period $T = 2\pi/\omega$ can be represented as a* **Fourier Series**

$$f(t) = a_0 + \sum_{n=1}^{\infty} [a_n \cos(n\omega t) + b_n \sin(n\omega t)] \tag{3.1.1}$$

*The frequency $\omega$ is known as the* **fundamental frequency** *and $\omega_n = n\omega$ for $n > 1$ are the* **harmonics**. *The result of Fourier-analysing a signal is a set of values for these coefficients for all $n$.*

**Process 3.1.2.** *The Fourier coefficients for a periodic function $f$ are evaluated using the orthogonality properties of sines and cosines:*

$$\frac{2}{T}\int_0^T \sin(n\omega t)\sin(k\omega t)dt = \delta_{nk} \tag{3.1.2}$$

$$\frac{2}{T}\int_0^T \cos(n\omega t)\sin(k\omega t)dt = 0 \tag{3.1.3}$$

$$\frac{2}{T}\int_0^T \cos(n\omega t)\cos(k\omega t)dt = \delta_{nk} \tag{3.1.4}$$

*Applying these orthogonality properties we obtain the following equations for the coefficients:*

$$a_0 = \frac{1}{T}\int_0^T f(t)dt \tag{3.1.5}$$

$$a_k = \frac{2}{T}\int_0^T f(t)\cos(k\omega t)dt, k \in \{1, 2, 3, ...\} \tag{3.1.6}$$

$$b_k = \frac{2}{T}\int_0^T f(t)\sin(k\omega t)dt, k \in \{1, 2, 3, ...\} \tag{3.1.7}$$

## 3.2.0 Simpson's Rule

**Process 3.2.1.** *Simpson's rule is a method of numerical integration. In particular, to approximate the integral $\int_a^b f(x)dx$ we split the interval $[a,b]$ into $n$ steps of length $h = (b-a)/n$, where $n \in 2\mathbb{Z}$. The approximation is taken as*

$$\int_a^b f(t)dt \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right] \tag{3.2.1}$$

*where $x_j = a + jh$ for $j \in \{0,1,2,...,n-1,n\}$. In particular $x_0 = a$ and $x_n = b$.*

## 3.3.0 Fourier Integral

**Remark 3.3.1.** For a non-periodic function $f(t)$ we require a ***Fourier integral*** over a continuous range of frequencies. The Fourier integral may be viewed as a limit of a Fourier series in the limit $T \to \infty$.

**Process 3.3.1.** *For a non periodic function $f(t)$ its Fourier integral is given by*

$$f(t) = \int_0^\infty [a(\omega)\cos(\omega t) + b(\omega)\sin(\omega t)]\, d\omega \tag{3.3.1}$$

*and the coefficient equations become functions of $\omega$:*

$$a(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(t)\cos(\omega t)dt \tag{3.3.2}$$

$$b(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(t)\sin(\omega t)dt \tag{3.3.3}$$

*Using Euler's identity $e^{i\omega t} = \cos(\omega t) + i\sin(\omega t)$ we can rewrite this as*

$$f(t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} F(\omega)e^{i\omega t}d\omega \tag{3.3.4}$$

$$F(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt \tag{3.3.5}$$

*where $F(\omega)$ is the* **Fourier Transform** *of $f(t)$. If the signal has dimensions of energy, then its Fourier Transform has units of Power, and its magnitude $|F(\omega)|$ is a measure of the total power in the signal at frequency $\omega$:*

$$|F(\omega)| = \sqrt{\mathbb{R}e\{F(\omega)\}^2 + \mathbb{I}m\{F(\omega)\}^2} = \sqrt{\pi}2\sqrt{a^2(\omega) + b^2(\omega)} \tag{3.3.6}$$

## 3.4.0 Discrete Fourier Transform

**Definition 3.4.1.** *In practice we approximate the Fourier integral and its other corresponding forms using finite summations, known as the* **Discrete Fourier Transform**. *Let $f(t)$ be a non-periodic function that we have $N$ samples of at intervals $h$ going from $t = 0$ to $t = (N-1)h$.*

*We define a discrete timeline by $t_m = mh$ for $m \in \{0, 1, 2, ..., N-1\}$. The time $\tau = Nh$ will become the period of our approximated function under reconstruction, and we need $\tau$ to be the longest time over which we are interested in the behaviour of $f(t)$. We also assume*

$$f(t) = f(t+\tau) \iff f(t_m) = f(t_{m+N}) \iff f_m = f_{m+N} \tag{3.4.1}$$

*The lowest frequency in the DFT will be $\nu_1 = 1/\tau = 1/(Nh)$, and this will be the fundamental frequency of our reconstructed function. The frequency spectrum is given by*

$$\Lambda := \left\{\nu_n = \frac{n}{Nh} = n\nu_1 | n \in \mathbb{N}\right\} \tag{3.4.2}$$

*We then discretize the integrals for the function and its Fourier transform as*

$$f_m = frac1N \sum_{n=0}^{N-1} F_n e^{i2\pi\nu_n t_m} = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{i2\pi mn/N} \tag{3.4.3}$$

$$F_n = \sum_{m=0}^{N-1} f_m e^{-i2\pi\nu_n t_m} = \sum_{m=0}^{N-1} f_m e^{-2\pi mn/N} \tag{3.4.4}$$

*Note that it can be shown that $F_{N/2-n} = \overline{F}_{N/2+n}$ for $n \in \{0, 1, ..., N/2\}$. The highest frequency component is thus $F_{N/2-1}$, corresponding to a frequency of*

$$\nu_{max} = (N/2-1)/Nh = 1/(2h) - 1/(Nh) \approx 1/(2h), \text{ if } N \text{ large} \tag{3.4.5}$$

*This is also known as the* **nyquist frequency** *$\nu_{Nyquist}$.*

*If the function has a component with frequency $\nu > \nu_{Nyquist}$ there are less than two sample points per period. This implies that there will be one or more frequencies less than $\nu_{Nyquist}$ for which the amplitude equals the true amplitude at the sample points, but these lower frequencies are not in the signal although they will appear in the frequency spectrum - this is phenomenon known as* **aliasing**. *The power spectrum of the DFT is often plotted as all values*

$$P_n = |F_n|^2 = \mathbb{R}e\{F_n\}^2 + \mathbb{I}m\{F_n\}^2 \tag{3.4.6}$$

**Process 3.4.2.** *In application we use the following summations for the components of $F_n$*

$$\mathbb{R}e\{F_n\} = \sum_{m=0}^{N-1} f_m \cos\left(\frac{2\pi mn}{N}\right) \tag{3.4.7}$$

$$\mathbb{I}m\{F_n\} = \sum_{m=0}^{N-1} f_m \sin\left(\frac{2\pi mn}{N}\right) \tag{3.4.8}$$

*We then reconstruct the original signal as*

$$f_m = \frac{1}{N}\sum_{n=0}^{N-1}\left\{\mathbb{R}e\{F_n\}\cos\left(\frac{2\pi mn}{N}\right) + \mathbb{I}m\{F_n\}\sin\left(\frac{2\pi mn}{N}\right)\right\} \tag{3.4.9}$$

# Chapter 4

# Curve-fitting and Optimization

## 4.1.0 Least Squares

**Process 4.1.1.** *Assume we have some sequence of measurements at times $t_i$*

$$y_i = y(t_i) \tag{4.1.1}$$

*and for some presumed model of the relationship*

$$y = f(t; p) \tag{4.1.2}$$

*expressed in terms of the independent variable $t$ and the model parameters $p$. We define the distance between a data point and our model by*

$$\delta_i = y_i - y_i \tag{4.1.3}$$

*A general measure of the distance is*

$$\sum_i |\delta_i|^d \tag{4.1.4}$$

*For $d = 2$ we obtain the* **chi-squared** *measure*

$$\chi^2 = \sum_i |y_i - y_i(p_1, ..., p_K)|^2 \tag{4.1.5}$$

*The best fit is assumed to minimize $\chi^2$ with respect to the model parameters*

$$\frac{\partial \chi^2}{\partial p_l} = \sum_i 2|y_i - y_i(p_1, ..., p_K)| \frac{\partial\, y_i}{\partial p_l} \tag{4.1.6}$$

*We also usually use the* **reduce chi-squared** *value*

$$\chi_N^2 = \frac{\chi^2}{N} \tag{4.1.7}$$

*In general we want $\chi^2$ to scale with the uncertainty in our measurements, so we wish to minimize*

$$\sum \left( \frac{expected - observed}{uncertainty} \right)^2 \tag{4.1.8}$$

**Remark 4.1.1.** This minimization can be done numerically using ***scipy.optimize*** package's ***minimize*** method. This package also has a ***curve_fit*** method for non-linear data sets.

## 4.2.0 Finite Differences

**Process 4.2.1.** *Consider a real-valued function $f(x)$ and its Taylor series about some point $x = a$:*

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{4.2.1}$$

*Consider a set of points, $x_i$, such that $x_{i+1} = x_i + \Delta$. Then we have that*

$$f(x_{i+n}) = f(x_i) + f'(x_i)n\Delta + \frac{f''(x_i)}{2!}(n\Delta)^2 + \frac{f'''(x_i)}{3!}(n\Delta)^3 + O(\Delta^4) \tag{4.2.2}$$

*Define $f(x_{i+n}) =: f_{i+n}$. Then for neighboring points we have*

$$f_{i+1} = f_i + f_i'\Delta + \frac{f_i''}{2!}\Delta^2 + \frac{f_i'''}{3!}\Delta^3 + O(\Delta^4) \tag{4.2.3}$$

$$f_i = f_i \tag{4.2.4}$$

$$f_{i-1} = f_i - f_i'\Delta + \frac{f_i''}{2!}\Delta^2 - \frac{f_i'''}{3!}\Delta^3 + O(\Delta^4) \tag{4.2.5}$$

*Subtracting the expression for two neighboring points we have*

$$f_{i+1} - f_i = f_i'\Delta + \frac{f_i''}{2!}\Delta^2 + \frac{f_i'''}{3!}\Delta^3 + O(\Delta^4) \tag{4.2.6}$$

*Dividing by $\Delta$ we obtain the following* **forward difference** *estimate of the first derivative*

$$\frac{f_{i+1} - f_i}{\Delta} = f_i' + \frac{f_i''}{2!}\Delta + \frac{f_i'''}{3!}\Delta^2 + O(\Delta^3) \approx f_i' + O(\Delta) \tag{4.2.7}$$

*Similarly we obtain the* **backward difference** *estimate*

$$f_i' \approx \frac{f_i - f_{i-1}}{\Delta} + O(\Delta) \tag{4.2.8}$$

*We can also cancel all even terms in the expansion by*

$$f_{i+1} - f_{i-1} = 2f_i'\Delta + 2\frac{f_i'''}{3!}\Delta^3 + O(\Delta^5) \tag{4.2.9}$$

*which gives us the* **centered difference** *estimate*

$$f_i' \approx \frac{f_{i+1} - f_{i-1}}{2\Delta} + O(\Delta^2) \tag{4.2.10}$$

*By adding terms we can cancel all odd terms and obtain*

$$f_{i+1} + f_{i-1} = 2f_i + 2\frac{f_i''}{2!}\Delta^2 + O(\Delta^4) \tag{4.2.11}$$

*We then obtain an expression for the* **second difference** *estimate*

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta^2} + O(\Delta^2) \tag{4.2.12}$$

# Part II

# Comp Phys 481

# Part III

# Computability

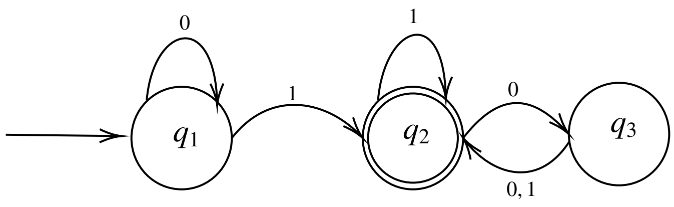
# Chapter 5

# Automata and Languages

## 5.1.0 Finite Automata

In general, finite automata are good models for computers with an extremely limited amount of memory. Finite automata and their probabilistic counterpart ***Markov chains*** are useful tools when we are attempting to recognize patterns in data.

A ***state diagram*** of a finite automaton $M_1$ is a diagram with several states and labeled arrows depicting transitions between states.



The ***start state*** is indicated by an arrow pointing at it from nowhere. The ***accept state*** is the one with a double circle, and the arrows going from one state to another are called ***transitions***.

A finite automaton consists of a set of states and rules for going between states depending on the input symbol; an input alphabet that indicates the allowed input symbols; a start state and a set of accept states; and a ***transition function***, $\delta$, which takes in a state and an input symbol and outputs another state: $\delta(state_1, input) = state_2$. Formally, we have

**Definition 5.1.1.** *A* **finite automaton** *is a 5-tuple* $(Q, \sum, \delta, q_0, F)$*, where*

1. *$Q$ is a finite set called the* **states**
2. *$\sum$ is a finite set called the* **alphabet**
3. *$\delta : Q \times \sum \to Q$ is the* **transition function**
4. *$q_0 \in Q$ is the* **start state***, and*

5. *$F \subseteq Q$ is the* **set of accept states**

If $A$ is the set of all strings that machine $M$ accepts, we say that $A$ is the ***language of machine M*** and write $L(M) = A$. We say that ***M recognizes A*** or that ***M accepts A***.

A machine may accept several strings, but it always recognizes only one language.

Let $M = (Q, \sum, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 ... w_n$ be a string where each $w_i$ is a member of the alphabet $\sum$. Then $M$ ***accepts*** $w$ if a sequence of states $r_0, r_1, ..., r_n$ in $Q$ exists satisfying three conditions:

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, ..., n-1$
3. $r_n \in F$.

We say that ***M recognizes language A*** if $A = \{w | M \text{ accepts } w\}$.

**Definition 5.1.2.** *A language is called a* **regular language** *if some finite automaton recognizes it.*

When designing an automaton it is best to put yourself in the place of the machine, seeing how you would perform the tasks you need the machine to. Once you have determined the necessary information to remember about the string as it is being read, you represent this information as a finite list of possibilities, and assign a state to each of the possibilites.

In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the ***regular operations***, and use them to study properties of regular languages:

**Definition 5.1.3.** *Let A and B be languages. We define the regular operations* **union**, **concatenation**, *and* **star** *as follows:*

- **Union:** $A \cup B = \{x | x \in A \vee x \in B\}$
- **Concatenation:** $A \circ B = \{xy | x \in A \wedge y \in B\}$
- **Star:** $A^* = \{x_1 x_2 ... x_k | k \geqslant 0 \text{ and each } x_i \in A\}$.

**Theorem 5.1.1.** *The class of regular languages is closed under the union operation. In other words, if $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$.*

*Sketch.* We have regular languages $A_1$ and $A_2$ and want to show $A_1 \cup A_2$ is regular. Let $M_1 = (Q_1, \sum, \delta_1, q_1, F_1)$ be the finite automata that recognizes $A_1$, and let $M_2 = (Q_2, \sum, \delta_2, q_2, F_2)$ be the finite automata that recognizes $A_2$. Construct $M = (Q, \sum, \delta, q_0, F)$ as follows:

1. Let $Q = \{(r_1, r_2) | r_1 \in Q_1 \wedge r_2 \in Q_2\} = Q_1 \times Q_2$.
2. $\sum$, the alphabet, is the same as in both automatas. Even if the automata had different alphabets $\sum_1$ and $\sum_2$, we would simply take $\sum = \sum_1 \cup \sum_2$.
3. $\delta$, the transition function, is defined for all $(r_1, r_2) \in Q$ and $a \in \sum$ by
$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$$
4. $q_0$ is the pair $(q_1, q_2)$
5. $F$ is the set of pairs in which either member is an accept state of $M_1$ or $M_2$. We can write it as
$$F = \{(r_1, r_2) | r_1 \in F_1 \vee r_2 \in F_2\}$$
This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

By construction it follows that $M$ is a finite automaton which recognizes the union of $A_1$ and $A_2$. ■

**Theorem 5.1.2.** *The class of regular languages is closed under the concatenation operation. In other words, if $A_1$ and $A_2$ are regular languages then so is $A_1 \circ A_2$.*

To prove this theorem we must learn some more techniques.

## 5.2.0 Nondeterminism

So far in our discussion, every step of a computation follows in a unique way from the preceding step.

**Definition 5.2.1.** *If when the machine is in a given state and reads the next input symbol the next state is given uniquely by the input, then this is* **deterministic** *computation. In a* **nondeterministic** *machine, several choices may exist for the next state at any point.*

We define nondeterminism to be a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. However, an NFA can have multiple exiting transition arrow for one symbol in the alphabet, or even no arrows for a symbol in the alphabet. Second, in a DFA, labels on the transition arrows are symbols from the alphabet. In general, an NFA may have arrows labeled with members of the alphabet or $\varepsilon$, and as with the alphabet symbols in NFA's, zero, one, or many arrows may exit from each state with the label $\varepsilon$.

After reading a symbol associated with multiple transition arrows at a given state of a NFA, the machine splits into multiple copies of itself and follows all the possibilities in parallel. If an input symbol doesnâppear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of computation associated with it. Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an $\varepsilon$ symbol on an exiting arrow is encountered, without reading any input, the machine splits into multiple copies, one following each of the exiting $\varepsilon$-labeled arrows and one staying at the current state. Then the machine proceed nondeterministically as before.

Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent processes or threads can be running concurrently, or it can also be thought of as a tree of possibilities; the root of the tree corresponds to the start of the computation, every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices, and the machine accepts if at least one of the computation branches ends in an accept state.

In an NFa, the transition function takes a state and an input symbol, or the empty string, and produces the set of possible next states. Formulla the definition of an NFA is:

**Definition 5.2.2.** *A* **nondeterministic finite automaton** *is a* 5*-tuple* $(Q, \sum, \delta, q_0, F)$*, where*

1. *$Q$ is a finite set of states*
2. *$\sum$ is a finite alphabet*
3. *$\delta : Q \times (\sum \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition function,*
4. *$q_0 \in Q$ is the start state, and*
5. *$F \subseteq Q$ is the set of accept states.*

The formal definition of computation for an NFA is as follows: let $N = (Q, \sum, \delta, q_0, F)$ be an NFA nad $w$ a string over the alphabet $\sum$. Then we say that $N$ ***accepts*** $w$ if we can write $w$ as $w = y_1 y_2 ... y_m$, where each $y_i$ is a member of $\sum \cup \{\varepsilon\}$ and a sequence of states $r_0, r_1, ..., r_m$ exists in $Q$ with three conditions:

1. $r_0 = q_0$
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, ..., m-1$, and
3. $r_m \in F$

Condition 1 states that the machine starts in the start state, condition 2 says that state $r_{i+1}$ is one of the allowable next states when $N$ is in state $r_i$ and reading $y_{i+1}$, and condition 3 says that the machine accepts its input if the last state is an accept state. Observe that $\delta(r_i, y_{i+1})$ is the set of allowable next states.

## 5.3.0 Cellular Automata

**Definition 5.3.1.** *A* **cellular automaton** *is a discrete model of computation.*

A cellular automaton classically consists of a regular grid of cells, each in one of a finite number of states (for example on and off). For each cell, a set of cells called its neighborhood is defined relative to the specified cell. We define an initial state by assigning a state for each cell. We advance our state, creating a new generation, according to some fixed rule (usually a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Usually the rule updates the state cells in the same manner and does not change over time, while being applied to the whole grid simultaneously.

The primary classifications of cellular automata, as outlined by Wolfram, are

1. Automata in which patterns generally stabilize into homogeneity (uniform in composition or character)
2. Automata in which patterns evolve into mostly stable or oscillating structures
3. Automata in which patterns evolve in a seemingly chaotic fashion
4. Automata in which patterns become extremely complex and may last for a long time, with stable local structures.

This last class is thought to be "computationally universal" or capable of simulating a "Turing machine."

Common types of neighborhoods are the ***von Neumann neighborhood*** which consists of four orthogonally adjacent cells (adjacent cells, but not the corners), and the ***Moore neighborhood*** which consists adjacent cells (creating a 3 by 3 square). For a cell and its Moore neighborhood there are $2^9 = 512$ possible patterns, each of which would determine whether the cell will be black or white on the next interval based on the rule table. The extended von Neumann neighborhood includes two cells in each orthogonal direction, for a total of eight.

**Proposition 5.3.1.** *The general equation for a system of rules is $k^{k^s}$ where $k$ is the number of possible states for a cell, and $s$ is the number of neighboring cells (including the cell to be calculated itself) used to determine the cell's next state.*

For infinite grids we either assume all cells are of the same state, except for a finite number, or all cells follow some pattern, except for a finite number. In a finite grid we must determine ways to deal with edge cells. One way of dealing with the finite cells is to employ a toroidal structure.

# Appendices

## .1.0 Lambda Functions

**Definition .1.1.** *Lambda functions are in practice one-line functions which cannot contain commands or more than one expression. In particular, a Lambda function can be created to and assigned to a variable in Python by*

$$g = \textbf{lambda } args_{array} : function\ rule \tag{.1.1}$$

## .2.0 List Comprehension

**Definition .2.1.** *List comprehension is a method of defining and filling a list all in one step. In general, list comprehension can be implemented in Python by*

$$list = [item\ for\ item\ in\ old\ list\ if\ P(item) == True] \tag{.2.1}$$

## .3.0 ODE-int Solve

**Definition .3.1.** *The* **scipy.integrate.odeint** *method can be used to numerically solve a system of differential equations. Define a method which takes the input vector of the system, a timeline, as well as any other needed parameters. Then, implement odeint by*

$$result = odeint(system\_function, y0, t, args = (arg\_tuple)) \tag{.3.1}$$

*where $y0$ is an initial state vector.*

## .4.0 Big-O Notation

The number of steps that an algorithm uses on a particular input may depend on several parameters. For simplicity, we compute the running time of an algorithm purely as a function of the lenggth of the string representing the input and don't consider any other parameters. In ***worst-case analysis***, the form we consider here, we consider the longest running time of all inputs of a particular length. In ***average-caes analysis***, we consider the average of all the running times of inputs of a particular length.

**Definition .4.1.** *Let $M$ be a deterministic Turing machine that halts on all inputs. The* **running time** *or* **time complexity** *of $M$ is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that $M$ uses on any input of length $n$. If $f(n)$ is the running time of $M$, we say that $M$ runs in time $f(n)$ and that $M$ is an $f(n)$ time Turing machine. Customarily we use $n$ to represent the length of the input.*

Because the exact running time of an algorithm often is a complex expression, we usually estimate it. In ***asymptotic analysis***, we consider only the highest order term of the expression for the running time of the algorithm to understand the running time when the algorithm is run on large inputs. This is commonly known as ***big-O notation***.

**Example .4.1.** Suppose our computational time is given by $f(n) = 6n^3 + 2n^2 + 20n + 45$. Then we describe the asymptotic analysis of this relationship by $f(n) = O(n^3)$.

**Definition .4.2.** *Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions. Say that $f(n) = O(g(n))$ if positive integers $c$ and $n_0$ exist such that for every integer $n \geqslant n_0$,*

$$f(n) \leqslant cg(n)$$

*When $f(n) = O(g(n))$, we say that $g(n)$ is an* **upper bound** *for $f(n)$, or more precisely, that $g(n)$ is an* **asymptotic upper bound** *for $f(n)$.*

$O$ can be thought of as representing a suppressed constant. big-$O$ interacts with logarithms in a particular way. Note that we have the relation $\log_b n = \log_c n / \log_c b$, where $\log_c b$ is a constant factor, so we write $f(n) = O(\log n)$ since the specifying the base amounts to choosing a constant factor which is hence suppressed in big-$O$ notation.

The expression $f(n) = 2^{O(n)}$ represents an upper bound of $2^{cn}$ for some constant $c$. The expression $f(n) = 2^{O(\log n)}$, with the identity $n = 2^{\log_2 n}$, and thus $n^c = 2^{c \log_2 n}$, represents an upper bound of $n^c$ for some constant $c$. The expression $n^{O(1)}$ represents the same bound because the expression $O(1)$ represents a value that is never more than a fixed constant.

Bounds of the form $n^c$ for $c > 0$ are called ***polynomial bounds***. Bounds of the form $2^{(n^{\delta})}$ are called ***exponential bounds*** when $\delta > 0$.

Big-$O$ notation says that one function is asymptotically no more than another. To say that one function is asymptotically less than another we use small-$o$ notation ($<$ instead of $\leqslant$).

**Definition .4.3.** *Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions. Say that $f(n) = o(g(n))$ if*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

*In other words, $f(n) = o(g(n))$ means that for any real numebr $c > 0$, a number $n_0$ exists where $f(n) < cg(n)$ for all $n \geqslant n_0$.*

## .5.0 Graphs Basics

**Definition .5.1.** *An* **undirected graph***, or simply a graph, is a set of points with lines connecting some of the points. The points are called* **nodes** *or* **vertices***, and the lines are called* **edges***.*

**Definition .5.2.** *The number of edges at a particular node of a graph is the* **degree** *of that node.*

No more than one edge is allowed between any two nodes, but we may allow an edge from a node to itself, called a ***self-loop***.

When we label the nodes and/or edges of a graph, we often call the result a ***labeled graph***.

**Definition .5.3.** *We say that a graph $G$ is a* **subgraph** *of a graph $H$ if the nodes of $G$ are a subset of the nodes of $H$, and the edges of $G$ are the edges of $H$ on the corresponding nodes.*

**Definition .5.4.** *A* **path** *in a graph is a sequence of nodes connected by edges. A* **simple path** *is a path that doesn't repeat any nodes.*

**Definition .5.5.** *A graph is* **connected** *if every two nodes have a path between them.*

**Definition .5.6.** *A path is a* **cycle** *if it starts and ends in the same node. A* **simple cycle** *is one that contains at least three nodes and repeats only the first and last nodes.*

**Definition .5.7.** *A graph is a* **tree** *if it is connected has no simple cycles. A tree may contain a specially designated node called the* **root**. *The nodes of degree 1 in a tree, other than the root, are called the* **leaves** *of the tree.*

**Definition .5.8.** *A* **directed graph** *has arrows instead of lines. That is, instead of unordered pairs $\{i, j\}$ to represent edges, we use ordered pairs $(i, j)$ to represent the arrow from $i$ to $j$.*

**Definition .5.9.** *The number of arrows pointing from a particular node in a directed graph is the* **outdegree** *of that node, and the number of arrows pointing to a particular node is the* **indegree**.

**Definition .5.10.** *A path in which all the arrows point in the same direction as its steps is called a* **directed path**.

**Definition .5.11.** *A directed graph is* **strongly connected** *if a directed path connects every two nodes.*

## .6.0 Strings and Languages Basics

We define an ***alphabet*** to be any nonempty finite set. The members of the akphabet are the ***symbols*** of the alphabet. A ***string over an alphabet*** is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas.

If $w$ is a string over an alphabet $\sum$, the ***length*** of $w$. $|w|$, is the number of symbols that it contains. The string of length zero is called the ***empty string*** and is denoted by $\varepsilon$. If $w$ is of length $n$, we can write $w = w_1w_2...w_n$ for $w_i \in \sum$. The ***reverse*** of $w$, written $w^{\mathcal{R}}$, is the string obtained by writing $w$ in the opposite order (i.e. $w_nw_{n-1}...w_1$). String $z$ is a ***substring*** of $w$ if $z$ appears consecutively within $w$.

If we have a string $x$ of length $m$ and a string $y$ of length $n$, the ***concatenation*** of $x$ and $y$, written $xy$, is the string obtained by appending $y$ to the end of $x$, as in $x_1...x_my_1...y_n$. We write $x^k$ to denote the concatenation of a string with itself $k$ times.

The ***lexicographic order*** of strings is the usual dictionary order. The ***shortlex*** or ***string order*** is identical to the lexicographic order, except that shorter strings precede longer strings.

We say that string $x$ is a ***prefix*** of a string $y$ if a string $z$ exists where $y = xz$, and that $x$ is a ***proper prefix*** of $y$ if in addition $x \neq y$. A ***language*** is a set of strings. A language is ***prefix-free*** if no member is a proper prefix of another member.

## .7.0 Recursive Transition Networks

Two common models for generating random grammatically correct text is through ***Markov models*** and ***recursive transition networks***. Markov models consist of analysing sample text in a genre by breaking it up into units (words or characters) and building up tables of the probabilities of units following other units. To generate text. one performs a ***random walk*** through these tables, starting at a random unit, and selecting the following unit randomly with respect to the probabilities discovered, and repeating the process. However, this does not model the grammatical structure of the text.

For Recursive transition networks (RTNs), one has to explicitly provide a specification for the domain of text to be modelled. This specification is in the form of a grammar, which defines the different forms and alternatives which can form a valid text fragment in the domain. An RTN can be conceptualized as a directed acyclic graph consisting of nodes representing subtasks which are performed in sequences and paths linking these nodes.

### .7.1 The Dada Engine

Scripts contain RTN definitions in the form of a "grammar." Literal quote, the output, is enclosed in double quotes. Text which is not quoted is assumed to the names of rules or other objects which are evaluated by the interpreter.

We define parametric rules, such as

A(name, pronoun): name " shook " pronoun " head " ;

We also define mappings between strings, for example to match names to pronouns one could define a mapping from each name to the respective pronoun.