

UNIVERSITY OF BIRMINGHAM

COMPUTER SCIENCE YEAR 2

FAISAL IMH ALRAJHI

---

## Year 2 Study Guide

---



UNIVERSITY OF  
BIRMINGHAM

# Contents

<b>1</b>	<b>Graphics</b>	<b>1</b>
1.1	Surface Geometry . . . . .	1
1.1.1	Notes . . . . .	1
1.1.2	Examples . . . . .	3
1.1.3	Normal Vectors . . . . .	5
1.1.4	Further Sources . . . . .	5
1.2	Transforms . . . . .	6
1.2.1	Notes . . . . .	6
1.2.2	Transformation Matrices . . . . .	6
1.2.3	Examples . . . . .	7
1.2.4	References . . . . .	8
1.3	Lighting . . . . .	9
1.3.1	Notes . . . . .	9
1.3.2	Phong Shading Equation . . . . .	10
1.3.3	Examples . . . . .	11
1.3.4	References . . . . .	12
1.4	Projection . . . . .	13
1.4.1	Notes . . . . .	13
1.4.2	Examples . . . . .	15
1.4.3	References . . . . .	16
1.5	Misc. Definitions . . . . .	17
1.5.1	Rasterisation . . . . .	17
1.5.2	Texture Mapping . . . . .	17
1.5.3	Hidden Surface Removal . . . . .	18
1.5.4	Splines . . . . .	19
<b>2</b>	<b>Computational Vision</b>	<b>20</b>
2.1	Light Capturing Devices . . . . .	20
2.1.1	Notes . . . . .	20
2.2	Human Vision . . . . .	21
2.2.1	Notes . . . . .	21
2.3	Edge Detection . . . . .	23
2.3.1	Convolving a Matrix . . . . .	23
2.3.2	First Order Operators . . . . .	23
2.3.3	Second Order Operators . . . . .	24
2.3.4	Canny Edge Detection . . . . .	25
2.3.5	References . . . . .	25
2.4	Facial Recognition . . . . .	26
2.4.1	Eigenfaces . . . . .	26
2.4.2	References . . . . .	26

2.5	Vision Systems . . . . .	27
2.5.1	Invariants . . . . .	27
2.5.2	Feature Detection . . . . .	27
2.5.3	Object Detection . . . . .	28
<b>3</b>	<b>Models of Computation</b>	<b>29</b>
3.1	Regular Expressions . . . . .	29
3.1.1	Regular Language . . . . .	29
3.1.2	Non-regular Language . . . . .	30
3.2	Turing Machines . . . . .	31
3.2.1	Turing Machines . . . . .	31
3.3	Complexity . . . . .	31
3.3.1	Complexity of Programs . . . . .	31
3.3.2	P and NP . . . . .	32
3.3.3	NP-Completeness . . . . .	32
3.4	Decidability . . . . .	32
3.4.1	Reduction and Decidability . . . . .	33
3.5	Lambda Calculus . . . . .	34
3.5.1	Reduction and Equivalence . . . . .	34
<b>4</b>	<b>Introductory Databases</b>	<b>35</b>
4.1	Databases and Sets . . . . .	35
4.1.1	Definitions . . . . .	35
4.1.2	SQL Statements . . . . .	36
4.2	JDBC . . . . .	39
4.2.1	Statements . . . . .	39
4.3	ER Diagrams . . . . .	39
<b>5</b>	<b>Computer Systems &amp; Architecture</b>	<b>41</b>
5.1	Architecture . . . . .	41
5.1.1	von Neumann . . . . .	41
5.1.2	Execution . . . . .	42
5.1.3	CPI and Execution Time . . . . .	42
5.2	MIPS Microarchitecture . . . . .	42
5.2.1	Building Datapaths . . . . .	42
5.2.2	Combining Datapaths . . . . .	45
5.2.3	Control . . . . .	46
5.2.4	State Control . . . . .	47
5.3	Optimisation . . . . .	48
5.3.1	Pipelining . . . . .	48
5.3.2	Caches . . . . .	48
5.4	Number Representations . . . . .	49
5.4.1	Types . . . . .	49
5.4.2	Arithmetic . . . . .	50
5.4.3	Hex Representation . . . . .	51
5.5	Logic . . . . .	51
5.5.1	Logic Gates . . . . .	51
5.5.2	Latches . . . . .	52
5.5.3	Datapath Components . . . . .	53
<b>6</b>	<b>C/C++</b>	<b>55</b>

6.1	C . . . . .	55
6.1.1	Data Types . . . . .	55
6.1.2	Function Calls and Scope . . . . .	55
6.1.3	Arrays . . . . .	56
6.1.4	Pointers . . . . .	57
6.1.5	User Defined Types . . . . .	58
6.1.6	Optimization . . . . .	60
6.2	C++ . . . . .	61
6.2.1	Introduction . . . . .	61
6.2.2	Inheritance . . . . .	62
6.2.3	Memory Management . . . . .	63
6.3	Generic Programming . . . . .	64
6.3.1	Pointers . . . . .	64
6.3.2	Templates . . . . .	64
<b>7</b>	<b>Mathematical Techniques for Computer Science</b>	<b>65</b>
7.1	Vectors and Matrices . . . . .	65
7.1.1	Vector Operations . . . . .	65
7.1.2	Matrix Operations . . . . .	65
7.2	Fields . . . . .	66
7.2.1	Notes . . . . .	66
7.3	Lines and Planes . . . . .	66
7.3.1	Gaussian Elimination . . . . .	66
7.3.2	Lines . . . . .	68
7.3.3	Planes . . . . .	68
7.3.4	Basis and Orthogonality . . . . .	69
7.4	Set Theory . . . . .	70
7.4.1	Notation . . . . .	70
7.4.2	Cardinality . . . . .	70
7.4.3	Relations . . . . .	70
7.4.4	Examples . . . . .	71
7.5	Functions . . . . .	71
7.5.1	Notes . . . . .	71
7.5.2	Examples . . . . .	71
7.6	Probability . . . . .	71
7.6.1	Notes . . . . .	71
7.6.2	Examples . . . . .	71
7.7	Random Variables . . . . .	71
7.7.1	Discrete . . . . .	71
7.7.2	Continuous . . . . .	71
7.7.3	Examples . . . . .	71
7.8	Equations . . . . .	71
<b>8</b>	<b>Introduction to Computer Security</b>	<b>72</b>
8.1	Cryptography . . . . .	72
8.2	Access Rights . . . . .	72
8.3	Web Security . . . . .	72
8.3.1	Communication Protocols . . . . .	72
8.3.2	Web Security . . . . .	72
8.4	Assembly . . . . .	72

<b>9</b>	<b>Professional Computing</b>	<b>73</b>
9.1	Computer Misuse . . . . .	73
9.2	GDPR . . . . .	73
9.3	Liability . . . . .	73
9.4	Intellectual Property . . . . .	73
9.5	Human Resources . . . . .	73
9.6	Internet and ISPs . . . . .	73
9.7	Ethics . . . . .	73
<b>10</b>	<b>Functional Programming</b>	<b>74</b>
10.1	Functional Programming . . . . .	74
10.2	Algebraic Data Types . . . . .	74
10.3	Imperative OCaml . . . . .	74
10.4	Modules and Functors . . . . .	74
10.5	Monads . . . . .	74
10.6	GADTs . . . . .	74
10.7	Lazy Programming . . . . .	74

# Chapter 1

## Graphics

### 1.1 Surface Geometry

This section covers the basics introduced in how to represent shapes in a computer.

#### 1.1.1 Notes

- Graphics Pipeline: It refers to the sequence of steps used to create a 2D raster representation of a 3D scene. It is the process of turning a 3D model into what the computer displays.
- Vertex: A point with three numbers representing its XYZ position in a plane
- Edge: An edge is the difference between two vertices; the segment connecting them
- Surface: A closed set of edges representing a face of a 3D object
- Polygon: A shape in space usually representing by a set of surfaces (other methods listed below)
- Polygon Table: A table containing a set of either vertices, edges and/or surfaces that is used to define the boundaries of a polygon. This is one method to define Polygons.
- Delaunay Triangulation: Given a set  $P$  of points in a plane, creates a triangular mesh  $DT(P)$  such that no point in  $P$  is inside the circumcircle of any triangle in  $DT(P)$ .

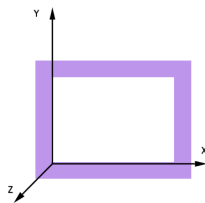


Figure 1.1: Coordinate system assumed throughout module

The default coordinate system assumed is right-handed: the positive  $x$  and  $y$  axes point right and up, and the negative  $z$  axis points forward. Positive rotation is counterclockwise about the axis of rotation.

Polygon Table consistency checks:

1. Every vertex is listed as an endpoint of at least two edges
2. Every surface is closed
3. Each surface has at least one shared edge

The order the vertices/edges are listed in a Geometric Polygon table do matter. Vertices written in clockwise order represent a surface pointing outwards. Whereas listing them counterclockwise represents an inwards pointing surface.

Meshes are a wireframe representation in which all vertices form a single set of continuous triangles, and all edges are a part of at least two triangles. Meshes can be generated by triangulation; but we covered just Delaunay Triangulation, defined above. Meshes can also be progressive. Detail in meshes is unnecessary at farther distances, so vertices can be removed and added to create less detailed or more detailed meshes, respectively. Progress meshes do this dynamically based on viewer distance.

There are a few ways to represent polygons in a space, with boundary representations being only one method.

1. Boundary Representation: Using vertices and drawing edges and surfaces from them
2. Volumetric Models: Using simple shapes and various operations to create more complex shapes
3. Implicit Models: Using implicit equations, such as that of a sphere, to generate shapes
4. Parametric Models: Uses parametric equations to plot the multiple axes of a shape

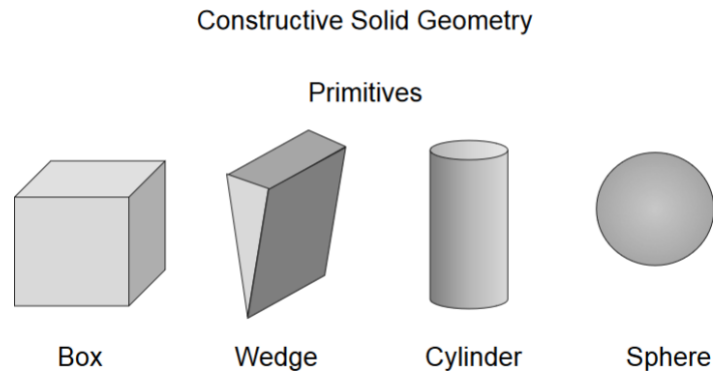


Figure 1.2: Constructive Solid Geometry (CSG) Primitives

We covered a few volumetric models in the module.

1. CSG: Uses primitive shapes and combines them uses set operations (union, difference, exclude, etc.) to generate new, more complex shapes.
2. Voxels: 3D Pixels, unit cubes
3. Octrees: Quad trees that divide in 3D space. Individual partitions are voxels
4. Sweep: Using a 2D shape, moves that shape across a path, generating a volume in position the 2D shape occupies during its path

One can also use implicit or parametric equations to generate shapes. Below is a list of equations that are common.

2D Circle:

$$\left(\frac{x}{r}\right)^2 + \left(\frac{y}{r}\right)^2 = 1 \quad (1.1)$$

2D Circle - Parametric:

$$\begin{aligned} x &= r \cos \theta \\ y &= r \sin \theta \\ -\pi &\leq \theta \leq \pi \end{aligned} \quad (1.2)$$

2D Ellipse - Parametric:

$$\begin{aligned} x &= r_x \cos \theta \\ y &= r_y \sin \theta \\ -\pi &\leq \theta \leq \pi \end{aligned} \quad (1.3)$$

3D Sphere:

$$\left(\frac{x}{r}\right)^2 + \left(\frac{y}{r}\right)^2 + \left(\frac{z}{r}\right)^2 = 1 \quad (1.4)$$

3D Ellipsoid:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \quad (1.5)$$

3D Sphere - Parametric:

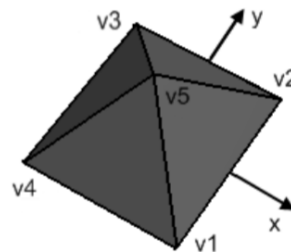
$$\begin{aligned} x &= r \cos \phi \cos \theta \\ y &= r \cos \phi \sin \theta \\ z &= r \sin \phi \\ -\pi &\leq \theta \leq \pi \\ -\pi/2 &\leq \phi \leq \pi/2 \end{aligned} \quad (1.6)$$

3D Ellipsoid - Parametric:

$$\begin{aligned} x &= r_x \cos \phi \cos \theta \\ y &= r_y \cos \phi \sin \theta \\ z &= r_z \sin \phi \\ -\pi &\leq \theta \leq \pi \\ -\pi/2 &\leq \phi \leq \pi/2 \end{aligned} \quad (1.7)$$

### 1.1.2 Examples

Define a vertex table and a surface table for the pyramid, depicted on the right. The base of the pyramid is a square with the side  $a=2$  centered in the origin. The height of the pyramid is equal to 2. Work in the right handed coordinate system.



<b>v1</b> [1; -1; 0]	<b>f1</b> : <b>v1</b> - <b>v2</b> - <b>v5</b>
<b>v2</b> [1; 1; 0]	<b>f2</b> : <b>v2</b> - <b>v3</b> - <b>v5</b>
<b>v3</b> [-1; 1; 0]	<b>f3</b> : <b>v3</b> - <b>v4</b> - <b>v5</b>
<b>v4</b> [-1; -1; 0]	<b>f4</b> : <b>v4</b> - <b>v1</b> - <b>v5</b>
<b>v5</b> [0; 0; 2]	<b>f5</b> : <b>v1</b> - <b>v4</b> - <b>v3</b> - <b>v2</b>

Figure 1.3: Example from Lecture



Further Examples are taken from quizzes and assignments

Consider the following vertex table and edge table for a convex 3D shape. Create the corresponding polygon (surface) table for that shape. Use vertex indices in your table.

vertex table

vertices	x	y	z
V1	2	0	-2
V2	0	1	-3
V3	1	2	-5.5
V4	2	3	-8
V5	4	3	-9
V6	5	1	-5.5
V7	4	2	4

edge table

E1	V7	V1
E2	V7	V2
E3	V7	V3
E4	V7	V4
E5	V7	V5
E6	V7	V6
E7	V1	V2
E8	V2	V3
E9	V3	V4
E10	V4	V5
E11	V5	V6
E12	V6	V1

Figure 1.4: Example from Quiz

Surfaces:

S1 = V1, V2, V3, V4, V5, V6

S2 = V1, V7, V2

S3 = V2, V7, V3

S4 = V2, V7, V3

S5 = V4, V7, V5

S6 = V5, V7, V6

S7 = V6, V7, V1

### 1.1.3 Normal Vectors

The normal vector of a surface points outwards from the surface. This is later used for lighting, projection and culling. Calculating normal vectors is a fairly simple task. For boundary polygons, the normal of a face is the cross product of two edges. Assuming vectors A and B, the cross product is the determinant of the following matrix;

$$\begin{bmatrix} i & j & k \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{bmatrix} \quad (1.8)$$

Which can be minimized to the following (longer) equation;

$$N = \begin{bmatrix} A_y B_z - A_z B_y \\ A_z B_x - A_x B_z \\ A_x B_y - A_y B_x \end{bmatrix} \quad (1.9)$$

To know which vectors to use for A and B, simply select an edge on a surface, and you use your right hand with your thumb pointing outwards and curl your hand around in the direction until the first vector hits your hand. Alternatively, you can piece it together by looking at the figure.

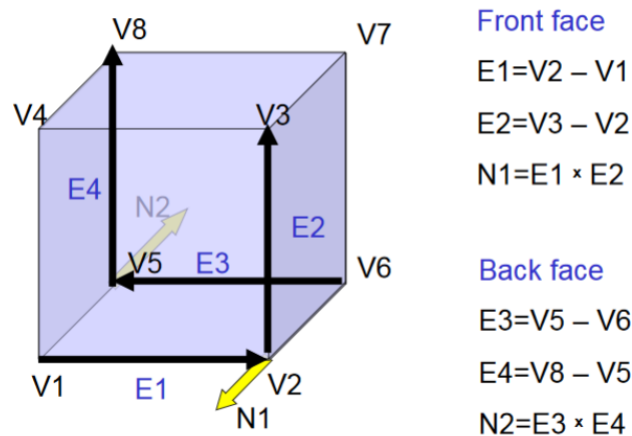


Figure 1.5: Normal Vector of cube from lecture

### 1.1.4 Further Sources

[Surface Representations](#)

[Alternate Lecture](#)

## 1.2 Transforms

This section covers simple transformation matrices.

### 1.2.1 Notes

- Transformation: a function that can be applied to each of the points in a geometric object to produce a new object.
- Translation: A geometric transform that adds a given translation amount to each coordinate of a point. Translation is used to move objects without changing their size or orientation.
- Rotation: A geometric transform that rotates each point by a specified angle about some point (in 2D) or axis (in 3D).
- Scaling: A geometric transform that multiplies each coordinate of a point by a number called the scaling factor. Scaling increases or decreases the size of an object, but also moves its points closer to or farther from the origin.

Transformations are applied to geometric objects to move them around. This is valuable when considering camera positions, or when laying out a world in a video game. Transformations can be applied as equations for each dimensions eg.

$$T_x = x + t_x$$

is the new x-position when applying the translation. However, there is a lack of uniformity between different transforms, some requiring x and y more than once, others being matrices. To standardize transforms, we instead use *Homogeneous Transformation Matrices*. Convert a 2D point to a 3D point by setting  $z = 1$ , and apply the transforms as matrices by replacing the variable found in each matrix template. This is an easy way of standardizing the equations, and allows for easy transforms by multiplying the transformations together before multiplying them with the point.

For example, applying a Translation T and then a Rotation R can be done by multiplying RT first and then multiplying the new transform matrix with the original points. This also makes it more efficient to move more than one point when they share the same transform, as it only need one multiplication per-point rather than one per-transform per-point.

### 1.2.2 Transformation Matrices

$$T_{2D} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \quad (1.10)$$

$$S_{2D} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1.11)$$

$$R_{2D} = \begin{bmatrix} \cos \theta & -\sin \theta & T_x \\ \sin \theta & \cos \theta & T_y \\ 0 & 0 & 1 \end{bmatrix} \quad (1.12)$$

$$T_{3D} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.13)$$

$$S_{3D} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.14)$$

$$R_{3D_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.15)$$

$$R_{3D_y} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.16)$$

$$R_{3D_z} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.17)$$

### 1.2.3 Examples

#### MATLAB Assignment Rotating

```

1      2. Apply rotation transformation to bowl object mesh model
2      %%% Define matrices for rotation of the bowl object around x, y and z ...
        axes by 20, 80 and 55 degrees respectively.
3      %%% Apply these three transformations to the original ...
        (non-transformed) bowl object in the given order
4      %%% and visualize the result using trisurf function.
5      %%% Save the result of your visualization to "2.png" file and include ...
        this file in your submission.
6      rx = [1 0 0 0; 0 cosd(20) -sind(20) 0; 0 sind(20) cosd(20) 0; 0 0 0 1];
7      ry = [cosd(80) 0 sind(80) 0; 0 1 0 0; -sind(80) 0 cosd(80) 0; 0 0 0 1];
8      rz = [cosd(55) -sind(55) 0 0; sind(55) cosd(55) 0 0; 0 0 1 0; 0 0 0 1];
9      rotated_object_vertices = rz*ry*rx*obj_v4;
```

#### Scaling

```

1      %% 3. Apply scaling transformation to bowl object mesh model
2      %%% Define a matrix for scaling of bowl object with scaling factor f = ...
        [3.5, 1.5, 2] in direction of x, y and
3      %%% z axes. Apply this matrix to your original bowl object ...
        (non-transformed) and visualize the result
4      %%% using trisurf function. Save the result of your visualization to ...
        "3.png"? file and include this file in your
5      %%% submission.
6
7      scaling = [3.5 0 0 0; 0 1.5 0 0; 0 0 2 0; 0 0 0 1];
8      scaled_object_vertices = scaling*obj_v4;
```

## Translating

```
1 %% 4. Apply translation transformation to bowl object mesh model
2 %%% Define a matrix for translation of bowl object by [-500, 50, -100] in ...
   direction of x, y and z axes. Apply
3 %%% this matrix to your original bowl object and visualize the result ...
   using trisurf function. Save the result
4 %%% of your visualization to "4.png"? file and include this file in your ...
   submission.
5
6 translate = [1 0 0 -500; 0 1 0 50; 0 0 1 -100; 0 0 0 1];
7 translated_object_vertices = translate*obj_v4;
```

## 3-in-1 Wombo Combo

```
1 %% 5. Apply all 3 transformations defined above to your original ...
   (non-transformed) bowl object one after the other in the given order.
2 %%% Display transformed meshes in the figure using trisurf.
3 %%% Save the result of your visualisation to "5.png" file and include ...
   this file to you submission folder.
4
5 %% Compute transformations (4x4 transformation matrices)
6
7 object_transformation = translate*scaling*rz*ry*rx;
```

## 1.2.4 References

[Quick Overview](#)

## 1.3 Lighting

This section covers things related to lighting and shading of objects in a scene.

### 1.3.1 Notes

- Diffuse: Non-shiny illumination
- Specular: Shiny reflections
- Ambient: background illumination

#### Ambient Light

- Global background light
- No direction
- Does not depend on anything

#### Diffuse Light

- Parallel Light Rays originating from a source direction
- Contributes to Diffuse and Specular Term

#### Spot Light

- Originates from a single source point
- Conic dispersion of light, intensity is a function of distance
- More realistic

#### Surface Properties

- Geometry - Position, orientation
- Colour - reflectance and Absorption spectrum
- Micro-structure - defines reflectance properties

#### Shading Models

- Flat shading is the simplest shading model. Each rendered polygon has a single normal vector; shading for the entire polygon is constant across the surface of the polygon. With a small polygon count, this gives curved surfaces a faceted look.
- Phong shading is the most sophisticated of the three. Each rendered polygon has one normal vector per vertex; shading is performed by interpolating the vectors across the surface and computing the color for each point of interest.
- Gouraud shading is in between the two: like Phong shading, each polygon has one normal vector per vertex, but instead of interpolating the vectors, the color of each vertex is computed and then interpolated across the surface of the polygon.

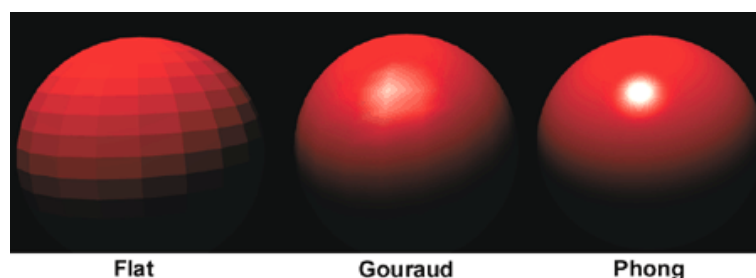


Figure 1.6: Shading Model Differences

### 1.3.2 Phong Shading Equation

$$\begin{aligned} \text{Colour} &= \text{Ambient} + \text{Diffuse} + \text{Specular} \\ \text{Colour} &= I_a K_a + I_d K_d \cos \theta_L + I_s K_s \cos^n \theta_S \end{aligned} \quad (1.18)$$

**Ambient Term** This is very easy. It is the  $K_a$  term multiplied with the Ambient intensity  $I_a$ .

$$\text{Ambient} = I_a K_a \quad (1.19)$$

**Diffuse Term** The diffuse term is usually straight forward.. It is the  $K_d$  term multiplied with the Light source intensity  $I_d$ . The angle  $\theta$  between the light source and the normal of the surface is then computed, and the  $\cos(\theta)$  is multiplied to obtain the full term.

$$\text{Diffuse} = I_d K_d \cos \theta_L \quad (1.20)$$

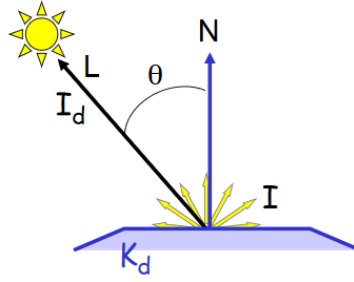


Figure 1.7: Diffuse term overview

**Specular Term** The specular term needs a few more steps. It is the  $K_s$  term multiplied with the reflected Light source intensity  $I_s$ . This is the ray that is bounced off of the surface, and is  $\theta_L$  away from the normal of the surface. This intensity is multiplied by the  $\cos$  of the angle  $\theta_S$ , the angle between the reflected ray and the line of sight from the camera. The  $\cos$  is raised to the  $n_{th}$  power, a factor known as a shininess factor that is usually given.

$$\text{Diffuse} = I_d K_d \cos^n \theta_S \quad (1.21)$$

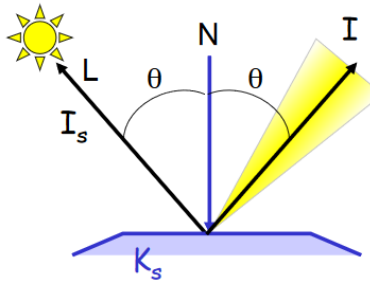


Figure 1.8: Specular term overview

### 1.3.3 Examples

#### MATLAB Assignment

##### Calculating Ambient Term

```
1 function colour = calcAmbient_skeleton(pixel.colour.current, Ia,Ka)
2
3 % Colour at current pixel
4 colour = pixel.colour.current;
5
6 % TO DO: Compute colour of the pixel here and write values to the
7 % corresponding place in image ImA
8
9 ambient = Ia.* Ka;
10 for i = 1:size(colour, 3)
11 colour(:, :, i) = ambient(i, i, :);
12 end
13 end
```

##### Calculating Diffuse Term

```
1 function colour = calcDiffuse_skeleton(pixel.colour.current, ...
    point.position, light.position, surface.normal, Id, Kd)
2
3 % Colour at current pixel
4 colour = pixel.colour.current;
5
6 %TO DO: Calculate light direction from light position and point position
7 light_direction = light.position - point.position;
8 %TO DO: Calculate normalised light direction
9 normalised_light = light_direction / norm(light_direction);
10 %TO DO: Calculate cos light direction, removing negative
11 %values
12 cos_light = dot(surface.normal, normalised_light);
13
14 %TO DO: Compute colour colour of the pixel here and write
15 % values to the corresponding place in image ImD
16 diffuse = Id.*(Kd.*cos_light);
17 for i = 1:size(colour, 3)
18 colour(:, :, i) = colour(:, :, i) + diffuse(i);
19 end
20 end
```



## Calculating Specular Term

```
1 function colour = calcSpecular_skeleton(pixel_colour_current, ...  
    point_position, light_position, surface_normal, normal_towardsViewer, ...  
    shininess_factor, Is, Ks)  
2  
3 % Colour at current pixel  
4 colour = pixel_colour_current;  
5  
6 %TO DO: Calculate light direction from light position and point position  
7 light_direction = point_position - light_position;  
8  
9 %TO DO: Normalised light direction  
10 normalised_light = light_direction / norm(light_direction);  
11 %TO DO: Normal component  
12 n = surface_normal / norm(surface_normal);  
13 %TO DO: Reflected Ray (tangent + ray in one step)  
14 R = n*2*dot(n, -1*normalised_light) + normalised_light;  
15 %TO DO: Normalised Reflected Ray  
16 normalised_reflection = R / norm(R);  
17 %TO DO: Calculate cos_spec, removing negative values  
18 cos_light = dot(normal_towardsViewer, normalised_reflection);  
19 if (cos_light < 0)  
20     cos_light = 0;  
21 end  
22 %TO DO: compute colour colour of the pixel here and write  
23 % values to the corresponding place in image ImS  
24 specular = Is.*(Ks.*(cos_light^shininess_factor));  
25 for i = 1:size(colour, 3)  
26     colour(:, :, i) = colour(:, :, i) + specular(i);  
27 end  
28  
29 end
```

### 1.3.4 References

[WebGL Specular Term](#)

## 1.4 Projection

This section deals with the virtual camera and how objects are projected from a proposed 'world' to a camera space.

### 1.4.1 Notes

- View Reference Point (VRP): The centre point/position where the eyes/camera is positioned
- Viewing Plane: The 2D plane in which images are rendered onto in relation to the camera
- Gaze Vector (N): The direction vector from the VRP facing towards the viewing plane
- Up Vector (U): The vector perpendicular to the normal facing upwards in relation to the Gaze Vector
- Handedness Vector (V): The vector perpendicular to the Gaze and Up Vectors facing right in relation to the gaze vector.
- Camera Coordinate System: The coordinate system formed by the N, U and V vectors acting as axes

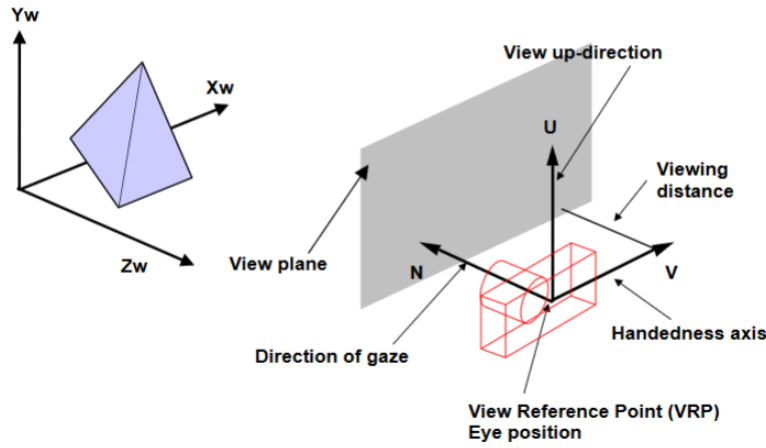


Figure 1.9: Visualised definitions

When rendering to camera, you first need to have a World Coordinate System and objects in that world. From there, you should know VRP and the Viewing Plane. (TP is the position on the Viewing Plane the camera points to.) Using this information, we can calculate the Camera Coordinate System (N V U).

$$\begin{aligned}
 U_{temp} &= [0 \ 1 \ 0] \\
 N &= TP - VRP \\
 V &= U_{temp} \times N \\
 U &= N \times V
 \end{aligned}$$

We use transformation matrices to move objects from the World Coordinate System to the Camera Coordinate System. Translate to the camera position, rotate to orient with the camera, then scale the x-axis  $-1$  to convert to the left-hand coordinate system to avoid mirroring.

$$T_{cam} = \begin{bmatrix} 1 & 0 & 0 & -x_{vrp} \\ 0 & 1 & 0 & -y_{vrp} \\ 0 & 0 & 1 & -z_{vrp} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.22)$$

$$R_{cam} = \begin{bmatrix} \frac{V_x}{|V|} & \frac{V_y}{|V|} & \frac{V_z}{|V|} & 0 \\ \frac{U_x}{|U|} & \frac{U_y}{|U|} & \frac{U_z}{|U|} & 0 \\ \frac{N_x}{|N|} & \frac{N_y}{|N|} & \frac{N_z}{|N|} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.23)$$

$$S_{cam} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.24)$$

- Centre of Project (COP): The view point the perspective is drawn from, usually the camera position
- Perspective Projection: Objects further away from the COP are scaled smaller
- Orthographic Projection: All objects are on the same scale, regardless of COP

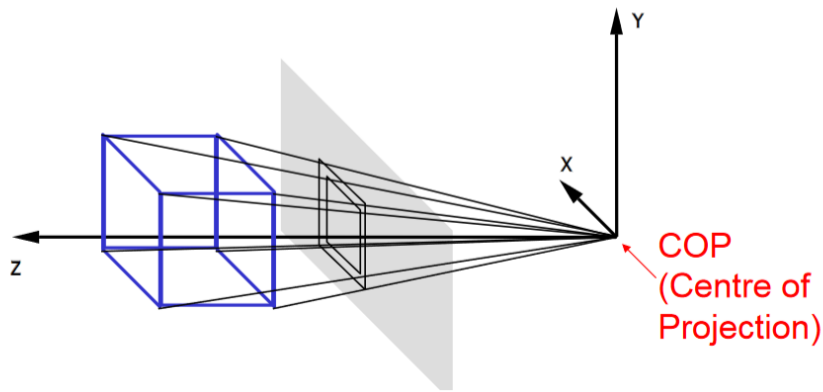


Figure 1.10: Projection Lines meet at the COP

The centre of project can be placed at either the centre/origin of the viewing coordinate system with the viewing plane on the positive z size, or the negative z side with the viewing plane placed on the centre/origin of the viewing coordinate system. Depending on which method is chosen, the computation differs slightly.

#### COP at Z=0

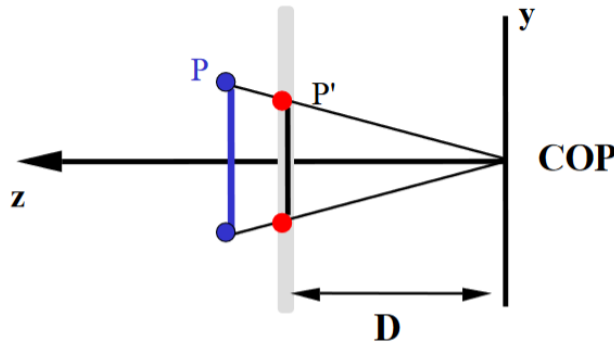


Figure 1.11: COP = 0

$$P_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 0 \end{bmatrix} \quad (1.25)$$

Where  $\mathbf{D}$  is the distance from the COP and the viewing plane. The generated matrix after multiplication is not homogenous; to convert it to homogenous form we simply divide all terms by the 4th term, which equates to

$$z/D$$

With the perspective project matrix, we can create a general form for moving the object from the World Space to the Camera Space.

$$C = P_{per} * S_{cam} * R_{cam} * T_{cam} \quad (1.26)$$

With the new vector converted to homogenous coordinates after applying  $C$ . All new coordinates should also have their Z coordinate equal to the Z coordinate of the viewing plane.

**COP at  $\mathbf{Z_i0}$**  When the viewing plane is at the origin,  $P_{per}$  changes slightly.

$$P_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/D & 1 \end{bmatrix} \quad (1.27)$$

Converting to homogenous would also be slightly different, but will still be fairly simple. All Z coordinates should end up at  $Z=0$ .

### 1.4.2 Examples

The questions here are from the quiz.

Consider the following vertex coordinates:

$$\begin{aligned} V1 &= [0, 10, 0] \\ V2 &= [5, 0, 5] \\ V3 &= [15, 5, 0] \\ V4 &= [5, 10, 15] \end{aligned}$$

Define the viewing (camera) coordinate system by computing its axes, i.e., direction of gaze  $\mathbf{N}$ , handedness vector  $\mathbf{V}$ , and vector  $\mathbf{U}$  which is the correct up-vector. Camera view reference point is  $\mathbf{VRP} = [60, 30, 100]$ , and the target point  $\mathbf{TP}$  is at the origin  $([0, 0, 0])$ . The camera coordinate system should be defined in the same way as it is shown in the lecture slides: using temporary up vector of  $[0,1,0]$  to compute handedness and up vectors of the camera.

1. What is the direction of the gaze vector?
2. What is the handedness vector?
3. What is the up vector?
4. Compute the matrix  $C_1$  to transform an object from the world coordinate system to the camera coordinate system (without project).
5. Compute the matrix  $C_2$  to transform an object from the world coordinate system to the camera coordinate system with the COP at the origin and the viewing plane at  $z = +40$ .
6. Compute the positions of the final transformed vertices.

## Answers to the previous questions

1.  $N = [-0.50, -0.25, -0.83]$

2.  $V = [-0.86, 0, 0.51]$

3.  $U = [-0.13, 0.97, -0.21]$

4.  $C_1 = \begin{bmatrix} 0.857 & 0 & -0.514 & 0 \\ -0.128 & 0.968 & -0.214 & 0 \\ -0.498 & -0.249 & -0.830 & 120 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

5.  $C_2 = \begin{bmatrix} 0.857 & 0 & -0.514 & 0 \\ -0.128 & 0.968 & -0.214 & 0 \\ -0.498 & -0.249 & -0.830 & 120 \\ -0.013 & -0.006 & -0.021 & 3.010 \end{bmatrix}$

6.  $V_1 = [0.00, 3.29, 40.0]$

$V_2 = [0.60, -0.60, 40]$

$V_3 = [4.61, 1.05, 40]$

$V_4 = [-1.33, 2.27, 40]$

### 1.4.3 References

No external references were needed on this section.

## 1.5 Misc. Definitions

This section covers general algorithms and definitions from Rendering to Texture Mapping. Since the exam will focus more on the previous parts for the more technical questions, this part will mostly be structured as a set of definitions.

### 1.5.1 Rasterisation

- Rasterisation: Converting an object from vector world coordinates to a raster image to display
- Digital Differential Analyzer (DDA): Rasterisation algorithm, interpolates values in an interval by computing separate equations for  $(x, y)$ . Is expensive and inefficient.
- Bresenham Algorithm: Incremental, integer only algorithm. Has many implementations.
- Antialiasing: utilizes blending techniques to blur the edges of the lines and provide the viewer with the illusion of a smoother line.

Circles can be plotted either directly with a circle equation, directly with polar coordinates, or using Bresenham's by looping across a set of values until all values on a circle are plotted. Bresenham and Polar methods abuse the symmetry of a circle's points.

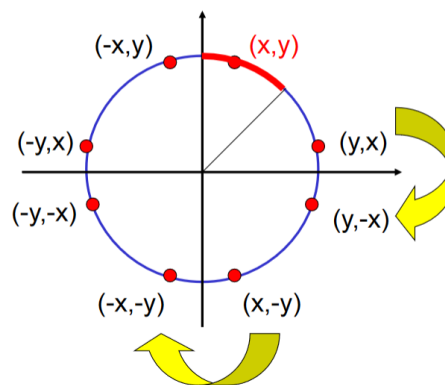


Figure 1.12: Symmetry of circle

### 1.5.2 Texture Mapping

Texture mapping can be defined as using an image and pasting the image onto a geometric model. There are three types of Texture Mapping.

1. Texture Image: uses iamges to fill inside polygons; inverse mapping using an intermediate surface
2. Environment/Reflection: uses a picture of the scene for texture maps
3. Bump: Emulates altering normal vectors during render process

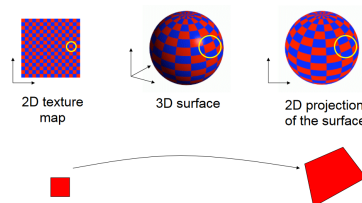


Figure 1.13: Image Mapping flow

When it comes to image mapping, there are two main methods.

1. Forward: copy pixel at source  $(u, v)$  to image destination  $(r, c)$ . Its easy to compute but leaves holes.
2. Backward: for image pixels  $(r, c)$ , grab texture pixel  $(u, v)$ . Harder to compute but looks better.

Environment mapping uses the direction of te reflected ray to index a texture map rather than using the ray projected to its surface. This approach isn't completely accurate as it assumes all reflected rays begin from the same point and that all objects in a scene are the same distance from that point.

Bump mapping is a method used to make a surface look rough. There are two variants.

1. Displacement Mapping: Height filed is used to perturb surface point along the direction of its surface normal. Inconvenient to implement since map must perturb geometry of model.
2. Bump Mapping: A perturbation is applied to the surface normal according to the corresponding value in the map. Convenient to implement as it automatically changes the shading parameters of a surface.

There is also mip-mapping. Mapping can cause aliasing to occur; mip-mapping is an anti-aliasing technique that stores texture as a pyramid of progressive resolution images, filtered down from the original. The further away a point is to be rendered, the lower lower resolution MIP-map is used.

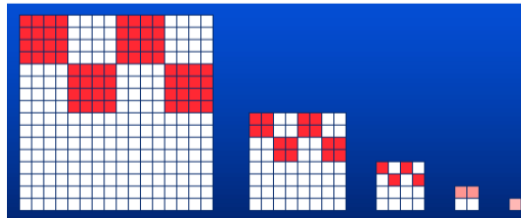


Figure 1.14: MIP-map examples

### 1.5.3 Hidden Surface Removal

1. Object-space method (OS): Operate on 3D Object entities.(vertices, edges, surfaces)
2. Image-spced (IS): Operate on 2D images (pixels)

There are a few algorithms to go over.

- Polygon Culling: removes all surfaces pointing away from the viewer; renders only what is visible; can be done by comparing if the z-value of the surface normal has the same sign as the z-value of the gaze vector. (If same, remove surface)
- Z-buffer Algorithm: Test visibility of surfaces one point at a time. Easy to implement, fits well with render pipeling, but some inefficiency with large distances. Standard algorithm in many packages, eg. OpenGL
- Painter's Algorithm: OS algorithm; Draw surfaces from back to front. Problems occur with overlapping polygons; as it will always render an object either above or below absolutely.
- Depth Sort: Painter's extension; sorts objects by depth like painters, but resolves overlap issues, splitting polygons if necessary. Does overlap/collision testing and splits on intersection point.

### 1.5.4 Splines

Splines are smooth curves generated from an input set of user-specified control points.

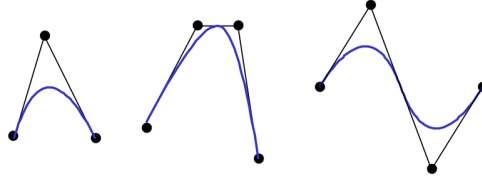


Figure 1.15: Splines with different control points

Bezier curves are generated by forming a set of polynomial functions formed from the coordinates of the control points. In parametric form, a Bezier Function  $P(u)$  can be represented as

$$P(u) = \sum_{k=0}^n p_k B_{kn}(u)$$

There are many ways to formulate Bezier curves. One such method is the de Casteljau algorithm. This algorithm describes the curve as a recursive series of linear interpolations. We draw lines between each of our control points in order, then continuously Lerp (linearly interpolate) each of the points generated by the previous Lerp until a curve is formed.

Bezier curve shape is influenced by all of its control points. There are B-splines that are only influenced by up to four of the nearest control points. This allows for interesting shapes without the inefficient calculations from insanely high polynomial Bezier curves.

Bezier surfaces are similar to Bezier curves, but instead of just one parameter  $t$ , there are two parameters  $s$  and  $t$ , and instead of a curve, it is a surface mesh.



# Chapter 2

## Computational Vision

### 2.1 Light Capturing Devices

This section deals with the basic evolution of light capturing devices.

#### 2.1.1 Notes

Initially, there was a single cell 1D capture of light. This had many problems, but the main one being it can only capture light from one direction, and had no understanding of the intensity of it (binary). Using multiple 1D Cells allowed for more directions to be captured. Having



Figure 2.1: 1D Cell

multiple cells curved allowed for capture of light from various directions, somewhat conic. But it had difficulty keeping track of an image, as the same image would be hitting too many cells, so it would be all over the place. The concept of a pinhole camera was the next step, but there

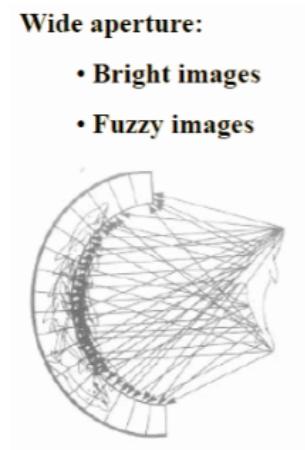


Figure 2.2: 1D Cell Curved Array

was an issue. With the wide aperture, the images were bright but they were fuzzy. With the pinhole, the images were sharp but they were too dim. How can we get the positives without the negatives? Simple, refraction from lenses. By having a lens refract light into the pinhole

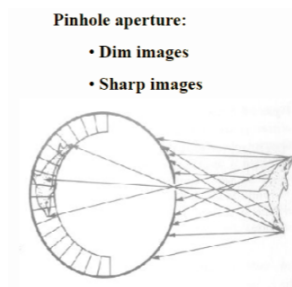


Figure 2.3: Pinhole

camera, or image point, it would allow for all the light to pass through the pinhole and hit different cells based on the initial angle the light hit the lens, maintaining the brightness from the wide aperture with the clarity of the pinhole aperture.

The virtual image created is upside down, and our eyes, based on this concept, simply allow the brain to flip it back.

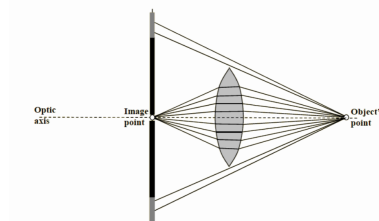


Figure 2.4: Lens in action

## 2.2 Human Vision

This section deals with human vision, a fairly straight forward topic (at least in our scope, this stuff can get wild pretty fast).

### 2.2.1 Notes

The human retina contains two types of photo receptor cells. Rods, of which there are 120 million, and Cones, of which there are 6 million. Rods are extremely sensitive to light and respond to single photons. They have poor spatial resolutions as multiple of them converge to the same neuron for data handling. However, thanks to their sensitivity, they help us see light in the dark. This is different with cones, which are active at higher light levels. Several neurons process Cone data, so they have higher spatial resolution than rods.

#### Receptive Field

The receptive field is the area on which light must fall for neurons to be stimulated. The size of a receptive field determines a few things. Small receptive fields are stimulated by high spatial frequencies; and large spatial fields are stimulated by low spatial frequencies. There are differences between the centre and periphery of field; the periphery of field contains more rods than cones (and vice versa). This is the reason that when we squint, we can see things in a higher quality. There are two types of ganglion cells, on-centre and off-centre, as seen in the image. On-centre cells are stimulated when the center is exposed to light, and are inhibited when the surrounded area is exposed. This works opposite for off-centre cells, as the name suggests.

Ganglion cells have a higher action potential rate depending on the intensity and location of

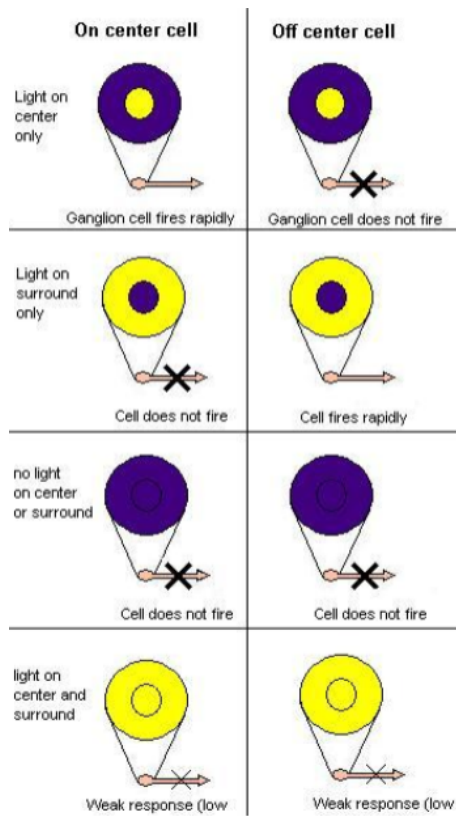


Figure 2.5: Ganglion responses

light hit. This allows us to have a grasp at contrast, as it responds differently to different intensities of light.

### Visual Pathway

1. Vision generated by photoreceptors in the eyes (as explained above)
2. The information leaves the eye by way of the optic nerve. Special note: The humans have a blind spot in the eye that does not allow them to see at one specific part of the eye; this is the optic nerve's location. Without it, we wouldn't be able to actually see.
3. There is a partial crossing of axons at the optic chiasm; this allows the brain to receive data on the same visual field from both eyes, superimposing images, creating a sense of depth, etc.
4. The axons following the chiasm, also known as the optic tract, wraps around the midbrain to get to the lateral geniculate nucleus (LGN).
5. The LGN axons fan out to the deep white matter of the brain before ultimately travelling to the primary visual cortex at the back of the brain, where the magic happens.

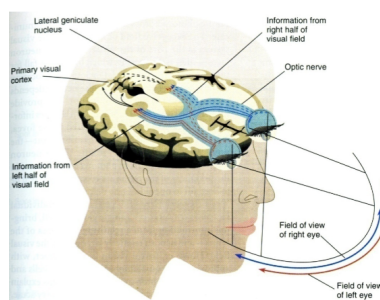


Figure 2.6: Visualising the visual pathway

## 2.3 Edge Detection

### 2.3.1 Convolving a Matrix

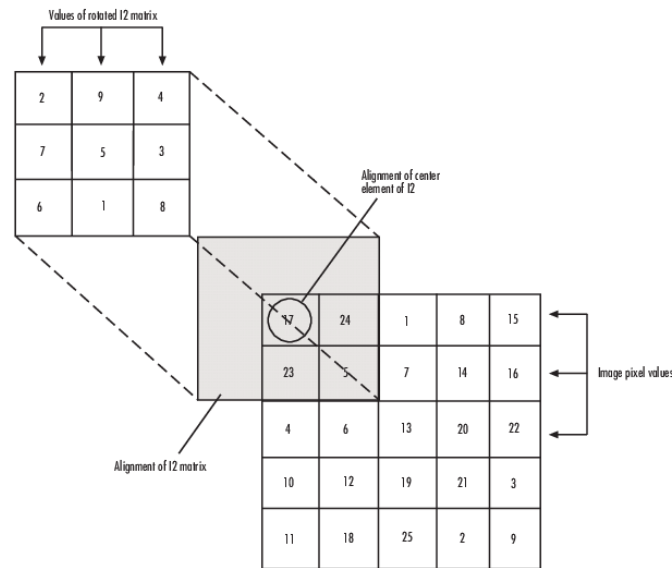


Figure 2.7: 3x3 convolution example

2D convolution of matrices is basically what the course is built off of, at least for edge detection. So its important to know how to do it. Given a 3x3 kernel and an MxN matrix, how does one convolve? Looking at the image, you will create a new matrix by overlaying the kernel with the matrix, multiplying each term on the kernel with the value it overlays and then sum them all up. The new generated value will then be placed in the same position as the centre of the overlay.

**Note:** for even kernels, you can select either the top-left or top-right centre value as the "centre" of the matrix.

### 2.3.2 First Order Operators

Edge detection operators are approximation of the first order derivative of the colours. The change in intensity of the colour from a set of points. The gradient of the intensity of colours. I could explain it in many different ways, but basically since edges are generally a different shade from its background, checking for a larger change in the intensity would usually return an edge point.

The approximations take the form of different kernels. Small kernels usually don't give a good enough approximation, and using too large a kernel would just blur all the values together and miss edges.

1. Sobel Operator: The sobel operator uses two kernels; one for the x-gradient and one for the y-gradient.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \text{ and } G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

2. Roberts Operator: The roberts operator can also use two kernels, but it instead measures the change diagonally.

$$G_x = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \text{ and } G_y = \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$$

Once we have applied  $G_x$  and  $G_y$ , and assuming we have a threshold  $h$  (the threshold can be found either with trial and error or setting up some Ai to test with a control set); the final edge

pixels can be generated with the approximation:

$$I = (h > |G_x| + |G_y|) \quad (2.1)$$

Edges generated like this might be susceptible to noise, because derivatives are looking for changes in the intensity. Noise are not part of the source, but rather a bi product from the limitation of our cameras and files. (JPG compressed image for example) When a derivative filter is used, it will simply return the change in intensity from the noise as part of the image. Depending on the quantity of noise, it could end up being a complete mess.

We can remove noise using the most common kernel, with the power of Gaussian. It is a gaussian distribution in the form of a 2D kernel, with the highest value in the centre of the matrix. The size of the matrix alters the effect of the noise filtering. A smaller matrix won't filter much, since it's not looking at a large enough space, and using too large a matrix will just blur the image together, removing edges.

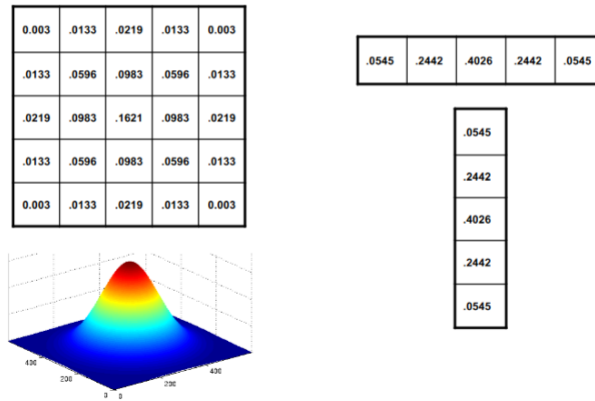


Figure 2.8: Gaussian Noise Filter overview

Rather than convolving the 2D Gaussian with the image, there is a more efficient way to go about this. You can create two 1D matrices ( $1 \times n, n \times 1$ ) and convolve those instead. The result will be the same, and it will be much faster than just doing the 2D matrix as there are less computations to deal with.

### 2.3.3 Second Order Operators

There is also the option of using Second Order Derivative operators. These function slightly different than first order, as seen in the image. Second Order Derivates look for the zero-crossings. These are the points the values of the graph change signs by crossing the 0-point of the axis. These points are the local maxima/minima in the first order operators. This method is very susceptible to noise, as noise will cause the function to cross zero many times. To alleviate these potential errors, we can apply a gaussian noise filter and a threshold for the distance needed to travel after crossing 0 to be counted as a "zero-crossing". The second order

operator, Laplacian Operator, can be derived as follows:

$$\begin{aligned}
\frac{\partial^2 f}{\partial x^2} &= \frac{\partial G_x}{\partial x} \\
&= \frac{\partial(f[i, j+1] - f[i, j])}{\partial x} \\
&= \frac{\partial f[i, j+1]}{\partial x} - \frac{\partial f[i, j]}{\partial x} \\
&= (f[i, j+2] - f[i, j+1]) - (f[i, j+1] - f[i, j]) \\
&= f[i, j+2] - 2f[i, j+1] + f[i, j]
\end{aligned} \tag{2.2}$$

The equation above is centred on  $[i, j+1]$ ; if we want to centre it on  $j$ , we simply do  $-1$  on all terms, and up with the following stuff.

$$\begin{aligned}
\frac{\partial^2 f}{\partial x^2} &= f[i, j+1] - 2f[i, j] + f[i, j-1] \\
\frac{\partial^2 f}{\partial y^2} &= f[i+1, j] - 2f[i, j] + f[i-1, j] \\
\Delta^2 &\approx \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}
\end{aligned} \tag{2.3}$$

We can also place the equation inside of a gaussian distribution function, and end up with a Laplacian of Gaussian (LoG), a potentially efficient second order operator.

### 2.3.4 Canny Edge Detection

Our new friend and scientist, J. Canny, has shown that the first derivative of the Gaussian closely approximates the operator that optimizes the product of signal - to - noise ratio and localization.

Canny Edge Detection is sort of a standard (its pretty good). There are four major steps to canny edge detection.

1. Find  $f_x = f * G_x$  and  $f_y = f * G_y$  where  $G(x, y)$  is the gaussian function and  $G_{x/y}$  is the derivative with respect to the required variable.  $G_a = \frac{-a}{\sigma^2}(G(x, y))$
2. Compute the gradient magnitude and direction.

$$mag(x, y) = |f_x| + |f_y|$$

$$dir(x, y) = \arctan \frac{f_y}{f_x}$$

3. Apply non-maxima suppression. In simpler maths terms, check if the gradient magnitude at a pixel is a local maximum along the gradient direction.
4. Apply hysteresis thresholding. This is fancy for applying a threshold at a low value  $t_l$  and a high value  $t_h$ , which is usually twice  $t_l$ . First mark the edges generated from the high threshold, as these are strong edges and are generally genuine. Trace an edge with the bidirectional information and, while tracing, apply the lower threshold to trace faint sections of edges that have a start point.

### 2.3.5 References

[Canny Edge Detection](#)  
[A paper on Edge Detection](#)  
[Good Lecture on Topic](#)

## 2.4 Facial Recognition

This section deals with the general idea of Facial Recognition.

### 2.4.1 Eigenfaces

Lets start by thinking of the face as a weighted combination of different components or representations of said face. These components are referred to as Eigenfaces (because who needs real words). Luckily, we can represent all these components as vectors. Eigenfaces are used in two ways; either as a form to store faces for later reconstruction or for recognition of a new image of a familiar face. We can do the learning using Principle Component Analysis (PCA). Given a set of points in 2D, as in the image, you can use vectors in the place of the usual line of best fit. When we use vectors, we would ideally want it to be in normal form with a scaling factor applied to it. This is where Eigenvectors come in. Eigenvectors take the following form:

$$A\vec{v} = \mu\vec{v}$$

where  $\mu$  is referred to as the "eigenvalue". Eigenvectors are obtained from a matrix, and follow some rules.

- Only square matrices have eigenvectors
- Different matrices have different eigenvectors
- Not all square matrices have eigenvectors
- An  $n \times n$  matrix has at most  $n$  distinct eigenvectors
- All distinct eigenvectors of a matrix are orthogonal

You ask, where do we get the matrix in this instance? We do it by finding a covariance matrix. Since we have a 2D plot of points, we can generate a 2x2 covariance matrix. This can be done by the following equations, the first for the covariance matrix, and the other for the variance function.

$$C = \begin{bmatrix} Var(x_1) & Covar(x_1, x_2) \\ Covar(x_1, x_2) & Var(x_2) \end{bmatrix}$$
$$Covar(x_1, x_2) = \frac{\sum_{i=1}^n (x_1^i - \bar{x}_1)(x_2^i - \bar{x}_2)}{n - 1}$$

With faces, we treat the face as a point in a high-dimensional space, and then treat the training set of faces as our set of points. We calculate the eigenvectors of our training set, and these become our eigenfaces. **TL;DR Eigenfaces are eigenvectors of covariance matrix of training set of faces**

When given an image of a face, we can transform it into face space (Facebook?) by applying the following equation:

$$w_k = x^i \vec{v}_k$$

where  $k$  is the number of eigenfaces, with the  $i^{th}$  face in image space, and a corresponding weight  $w$ . Image recognition becomes easy, simply find the euclidean distance of the desired face and all faces stored; the closest face is most likely our match. You can also reconstruct a face from the eigenfaces. The more eigenfaces the better the reconstruction.

### 2.4.2 References

Just the slides

## 2.5 Vision Systems

This section deals with feature detection of vision systems.

### 2.5.1 Invariants

Ideally, we would like some invariance in different settings for what we would like to see.

1. Illumination: Can be achieved by normalising the light levels and storing the new normal intensities. Can also use difference based metrics (sift).
2. Scaling: Store pyramids of the image, with each step half the size of the original. Find pixel values with Gaussian blur. Can also use Scale Space method, using Difference Gaussians to find repeatable points (invariant) in scale.
3. Rotation: Rotate all features to go the same way in a determined manner. Take histogram of gradient directions and rotate to the most dominant.

### 2.5.2 Feature Detection

In motion tracking, our goal is to see things that move between a set of images. You can begin by knowing which of the four following cases fits best:

1. Stationary Camera, Stationary Objects
2. Stationary Camera, Moving Object
3. Moving Camera, Stationary Objects
4. Moving Camera, Moving Object

When detecting change, you can compare the intensity at one point in time with its previous incarnation (or any other, if you need something specific). However, this lends to issues appearing from random noise, which will pop up everywhere. What we want is to then filter out our noise, but how do we do such a thing?

We can use the idea of 8 or 4 connectedness. Two pixels are 4-neighbours if they share an edge, and are 8-neighbours if they share a corner. Two pixels are 8-connected if we can create a path of 8-neighbours from one to another. We can use these concepts and group pixels into clusters. We can then threshold the clusters and remove any clusters that are below said threshold, ideally leaving behind just the changes and removing noise. However, just seeing the difference in

## Moravec Operator

measure the intensity variation by placing a small square window (typically, 3x3, 5x5, or 7x7 pixels) centered at P

- then shifting this window by one pixel in each of the eight principle directions (horizontally, vertically, and four diagonals).

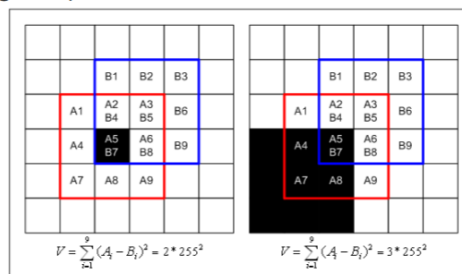


Figure 2.9: Gaussian Moravec Operator Overview

pixels might not be enough. Ideally, we would want to find our points of interest and compare



those points between images. For this, we can use the Moravec operators, a corner detector. Corners are better than edges for this task because corners can measure the highest intensity change, as oppose to edges. The image explains how the Moravec operators works. Once we obtain our new values from our friend Moravec, we can just keep the local maxima.

Motion Correspondence has three main principles we need to worry about.

1. Discreteness: Measure of the distinctiveness of individual points
2. Similarity: Measure of how closely two points resemble one another
3. Consistency: Measure of how well a match conforms with nearby matches

### 2.5.3 Object Detection

Object recognition needs the following components:

1. Model Database: Stores all models known by the system; information stored depending on approach for recognition. Generally, these values are abstract (shape, colour, size, etc.)
2. Feature Detector: Applies operators to image and identifies location of features. How a feature detector works depends on what types of models are stored in the database.
3. Hypothesizer: Assigns likelihoods to objects present to reduce computational expense.
4. Hypothesis Verifier: Uses object models to verify hypothesis and refines the hypothesizer likelihood.

Each of these components come with issues, mostly related to what sort of thing you want to do. (2D vs 3D, model vs object, etc.)

# Chapter 3

## Models of Computation

(Thanks to Liam for helping out on this section)

### 3.1 Regular Expressions

#### 3.1.1 Regular Language

Strings can be broken down as "regular expressions", a logical expression that can be used to solve matching and searching problems. A matching problem can be checking if a string is a valid password that contains at least  $X$  characters and at least  $Y$  digits. A searching problem can be, given a string, list all occurrences of an email address in it. Regexp are used to solve problems like these.

For example, the regular expression  $\mathbf{c(bb|ca)^*}$  will return true when matched with **cbbbbca** but return false when matched with **bca**. To solve problems using regexps, we can use Deterministic Finite Automatons (DFAs). They are deterministic because the initial state and the result of each transition are specified, and finite because there set of states is a finite set. When given a string, in the DFA seen, it will process each letter one at a time, going from state and following the transitions.

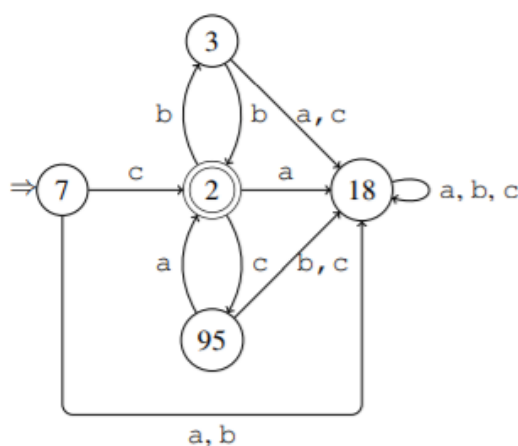


Figure 3.1: Example of a DFA

As you can see, there are a few loops/areas where the DFA will definitely return false once it reaches a certain point. A more efficient way to deal with these things is to completely remove the transitions, and have it return a failure at points we know it will definitely fail. These are called partial DFAs, denoted by the  $\delta$  symbol ( $\delta$ DFA). These are usually faster to write and run faster.

What happens when the DFA is not deterministic? An NFA if you will. NFAs differ from DFAs in two ways.

- An NFA can have several initial states
- From a given state, when  $a$  is input, there can be several possible next states.

NFAs are a relation and not a function. They can still return acceptable words, but a program can make no use of it since its, well, non-deterministic. However, there are ways to determinize an NFA. The best, basic way to accomplish this is to combine states into a set of state transitions if they match similar inputs/outputs. Conversion can take a few steps, but it's fairly simple and systematic in this regard.

Some trouble can pop up from NFAs with an  $\epsilon$  transition, or an empty transitions. These bad boys can change from one state to another for literally no reason. These NFAs are called  $\epsilon$ NFAs, which seems obvious. With this out of the way, let's make note of Kleene's Theorem.

**For a language  $L$ , the following are equivalent:**

- $L$  is regular
- The matching problem for  $L$  can be solved by a DFA.

A systematic way to turn a regexp to an equal DFA:

1. Convert the regexp into an  $\epsilon$ NFA
2. Convert it down to an NFA
3. Convert it further down to a DFA.
4. Minimize DFA size by identifying equivalent states, removing unreachable states and just general refactoring.

### 3.1.2 Non-regular Language

We know that some languages are not regular, because there are uncountably many languages and only countably many regexps. An example of a useful non-regular language is the language of well-balanced brackets. The check for balance can be done in a stack, pushing and popping from a stack when reading '(' and ')' respectively. However, if we have a large amount of '(', the stack will overflow thanks to our finite memory. When talking about non-regular languages,

$$\begin{aligned} \Rightarrow A &::= 3 \mid 5 \mid A + A \mid A \times A \mid (A) \mid \text{if } B \text{ then } A \text{ else } A \\ B &::= A > A \mid B \text{ and } B \mid (B) \end{aligned}$$

Figure 3.2: Example of a BNF

we can talk about context-free grammars. Grammars are written in Backus-Naur Form (BNF) and, in basic terms, represent different characters as symbols and use these symbols to derive a word. These symbols are either terminal or non-terminal; terminal means it can be expanded to a final value of sorts. The grammars we covered are context-free because you can apply a production rule to any non-terminal regardless of the other symbols in the word.

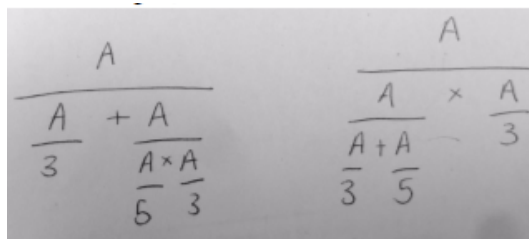


Figure 3.3: Ambiguous derivation tree

## Some Definitions

- Leftmost Derivation: Converting a grammar to a word by replacing the leftmost non-terminal
- Derivation Tree: A tree structure representing a derived word where each leaf is a non-terminal.
- Ambiguous: A grammar is ambiguous when there is more than 1 derivation tree possible with left-most derivation

## 3.2 Turing Machines

### 3.2.1 Turing Machines

A turing machine is the precise model of computation (roll credits) that are used to determine decidability and complexity of programs. A turing machine has finitely many states, but it also has access to an external, infinite memory in the form of a tape. The machine has a head that sits over one of the cells in a tape. A concise definition of the turing machine consists of the following data:

- A finite state  $X$  of states
- An initial state  $p$
- A transition function  $\delta$  from  $X$  to:
  1. Read
  2. Write
  3. Left
  4. Right
  5. No-op
  6. Return

There are also a few variants that expand on the basic turing machine.

**Auxiliary Characters** In addition to the input alphabet and "blank", a turing machine can have auxiliary characters that are used during the completion of a program, but these characters cannot appear at the start or at the end of a task.

**Auxiliary Tape** A two-tape Turing Machine has, well, two tapes. A program is now capable of completing tasks that read/write etc. on two tapes (Main  $x$ , Aux  $x$ ). The copy-reverse problem can be solved in linear time using a two-tape machine.

**2D Turing Machine** Instead of a tape, a 2D machine has a sheet instead of a tape, so it can go up and down. It must ensure that all but one row are blank on completion of a program.

## 3.3 Complexity

### 3.3.1 Complexity of Programs

When trying to find the number of steps, it is helpful to remember some laws of summations. The most important part of complexity for our course is to practice deriving worst case and average case complexities. Hopefully you remember how Big O worked from Year 1 DSA. There's a bit more in Models than what we learned, a more interesting way of deriving it.

$$\exists M. \exists C. \forall n \geq M. f(n) \leq C \times g(n)$$

$$\begin{aligned}
0 + 1 + 2 + \cdots + (n-1) &= \frac{1}{2}n(n-1) \\
1 + 2 + 3 + \cdots + n &= \frac{1}{2}n(n+1) \\
1 + b + b^2 + \cdots + b^{n-1} &= \frac{b^n - 1}{b - 1} \quad (b \neq 1) \\
1 + b + b^2 + \cdots + b^n &= \frac{b^{n+1} - 1}{b - 1} \quad (b \neq 1)
\end{aligned}$$

Figure 3.4: Helpful sums to memorize

$M$  is an arbitrary number. The set from  $n = [M, +\infty]$  will hold true.

There is one more variation to this for polynomial complexity. Polynomial time solutions are said to be *feasible*.

$$\exists M. \exists C. \exists k. \forall n \geq M. f(n) \leq C \times n^k$$

### 3.3.2 P and NP

#### Definitions

- Search Problem: Relation from the set of words (problems) to the set of words (solutions).
- NP Search Problem: Search problems with solution of polynomial size and checked in polynomial time. Will have a solution in exponential time.

What is the NP = P problem? It is the question, can every NP search problem be solved in polynomial time? If the answer is yes for Sudoku, then it's yes for every NP search problem. This is because Sudoku is a special kind of NP search problem called "NP-complete", meaning that it's as hard as an NP search problem can be. Any NP search problem can be reduced (in polynomial time) to Sudoku. By contrast, it is not known whether factorization is NP-complete, so showing that factorization can be solved in polynomial time won't prove anything major.

### 3.3.3 NP-Completeness

**SAT** A problem instance of this NP search problem consists of propositional variables and a formula built from this variables. The formulas are generated from a specific grammar. A satisfying assignment is an assignment of truth variables that makes the formula true. SAT is useful in solving constraint problems, like "I need to study Models with Jon and I need to study Security with Clair, but all of our free times can only coincide once." You can use SAT to potentially find the magic timings.

**Cook's Theorem** Any NP problem can be reduced to SAT, like  $n$ Sudoku for example. Find a mapping of different characters and constraints to propositional logic, and you have reduced your problem to SAT.

## 3.4 Decidability

A decision problem is said to be decidable when there is some program that, when given an argument, says whether the answer is Yes/No.

### 3.4.1 Reduction and Decidability

**Church's Thesis** Any decision problem on words that can be solved by an algorithm can be solved by a Turing machine.

**Reduction** Suppose we have two problems P and Q. Suppose that we show how to solve Q using a black box that solves P. Then is called reducing problem Q to problem P. For example, I saw a recipe for profiteroles that said “take some pastry balls” and “take some chocolate sauce”. The author reduced the problem of making profiteroles to the problems of making pastry balls and making chocolate sauce. Suppose we've reduced problem Q to problem P.

- If P is decidable, then Q is decidable
- If Q is undecidable, then P is undecidable

**Halting Problem** Some programs run forever, others halt. Usually we want our programs to halt; if they run forever, that's a bug. Can we test this automatically? Our main man Turing proved the halting problem to be undecidable. We begin the proof by assuming it is decidable. Consider the unary halting problem: given a unary Java method.

```
void f(String x) {  
    ...  
}
```

and a String  $y$ , does  $f$  terminate when called with argument  $y$ ? We can reduce this problem to a nullary one. Take the code of  $f$  and replace  $x$  with  $y$ ; then the nullary method ( $g$ ) terminates when called if  $f$  terminates when called with argument  $y$ . Since the nullary problem is, on assumption, decidable, the unary one is too. So we have a program:

```
boolean haltcheck(String somemethod, String y)
```

Where *somemethod* is the body of a unary method. When applied to  $M$  and  $y$ , this method returns true if  $M$  applied to  $y$  terminates, otherwise it returns false.

2. Next, we turn this into a program.

```
hangcheck(String somemethod, String y) {  
    if haltcheck(somemethod, y) {  
        while (true) {}  
    } else {  
        return;  
    }  
}
```

This method, when applied to  $M$  and  $y$ , hangs if  $M$  and applied to  $y$  terminates, otherwise it returns.

3. Next, we turn this into a program:

```
void doublehang(String y) {  
    hangcheck(y,y);  
}
```

This method, when applied to  $y$  (the body of a unary method), will hang if  $y$  applied to  $y$  terminates, otherwise it returns.

4. Finally let  $z$  be the body of *doublehang*. We see that *doublehang*, when applied to  $z$ , terminates if it hangs. Contradiction. (Objection!)

**Semantics** Semantic properties of a code are properties that affect the behaviour of the program. For example, does this software only print polite words? Does this software, when supplied with two positive integers, always return the highest common factor? Surprise surprise, we can't. Thanks to our good friend from down the tape, Rice's Theorem states that, in all but two cases, every semantic property is undecidable. That's because there are three cases, and Rice only covers the third one, since the other two are decidable.

1. A property never holds (decidable)
2. A property always holds (decidable)
3. A property sometimes holds and sometimes doesn't (Undecidable due to Rice's Theorem)

## 3.5 Lambda Calculus

**Conventions**  $\lambda x$  is equivalent to **fun** **x**  $\rightarrow$  in OCaml. Just a quick convention comparison to get you up to speed.

- Juxtaposition is application
- Application has highest precedence
- $\lambda x$  has lowest precedence
- Arithmetic Operations are intermediate
- Application associates to the left ( $MNP$  is bracketed as  $(MN)P$  not as  $M(NP)$ )

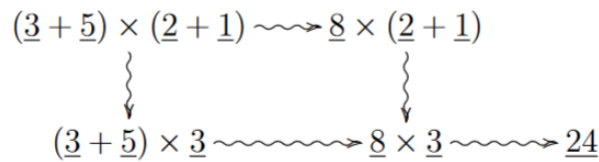


Figure 3.5: Reduction Graph with more than one order

### 3.5.1 Reduction and Equivalence

There are two types of reduction,  $\delta$ -reduction and  $\beta$ -reduction.

- $\delta$ -reduction: Basic arithmetic reduction ( $m \times n \rightarrow mn$ )
- $\beta$ -reduction: Applies  $\lambda$ -abstractions ( $((\lambda x.x + 3)7) \rightsquigarrow_{\beta} (7 + 3)$ )

We use a combination of both to formulate reduction graphs. Reduction can branch of and be done in different evaluation orders, as seen in the figure. There is also the notion of  $\alpha$ -equivalence, which is simply when two lambda function evaluate to the same result despite being different originally.

# Chapter 4

## Introductory Databases

### 4.1 Databases and Sets

This section deals with some definitions and how to formulate SQL Queries

#### 4.1.1 Definitions

##### Relations

- Domain (D): An arbitrary (non-null) set of atomic values
- Attribute Name (A): A symbol with an associate domain
- Relational Schema (R): A finite set of attribute names
- Tuple (t): a mapping from the attributes of a relational schema to the union of their domains
- Relation (r): A finite set of tuples of relational schema
- Degree: Number of attributes [columns in an SQL table]
- Cardinality: Number of tuples [rows in an SQL table]

##### Keys

- Superkey: a set of attributes that can *always* be used to differentiate one tuple from another
- Key - a minimal superkey
- Concatenated Key - a key with more than one attribute
- Candidate Key - any key
- **Primary Key** - one of the candidate keys used to reference from other relations
- **Foreign Key** - an attribute of the relation which is a key for another relation

##### Data Types

- BOOLEAN: TRUE, FALSE, or NULL
- Strings:
  - CHAR(length)
  - CHARACTER(length)
  - VARCHAR
  - TEXT
- INTEGER
- REAL: A floating point



- Arbitrary Precision:
  - NUMERIC
  - DECIMAL
  - MONEY
- Date and Time
  - TIMESTAMP
  - DATE
  - TIME

### Constraints

- DOMAIN: The domain/type of the attributes; eg INTEGER, VARCHAR
- NOT NULL: Ensures the column will not have a null value
- UNIQUE: Ensures uniqueness
- PRIMARY KEY: Not-null, unique column used as a primary identifier of a row
- FOREIGN KEY: Unique identifier to a row from another table
- CHECK(condition): Ensures all values in the column meet the defined condition (eg. cost > 10)
- DEFAULT(default): Sets the value to the given value if no other value is specified on creation

### Foreign Key Syntax

- ON DELETE (action): Does the specified action when the key or referenced table is deleted
- ON UPDATE (action): Does the specified action when the key or referenced table is updated
- NO ACTION: Action - Does nothing
- RESTRICT: Action - will not allow user to delete/update the key/referenced table
- CASCADE: Action - cascades the change on the original to the value found in the foreign table
- SET NULL: Action - sets the value to null in the foreign table
- SET DEFAULT: Action - sets the value to default in the foreign table

## 4.1.2 SQL Statements

### CREATE

```
CREATE TABLE tableName (
    attributeName TYPE CONSTRAINTS,
    attributeName2 TYPE CONSTRAINTS,
    exampleID TYPE CONSTRAINTS,
    exampleID2 TYPE CONSTRAINTS,

    PRIMARY KEY (exampleID),
    FOREIGN KEY (exampleID2) REFERENCES otherTable(id)
        ON UPDATE action
        ON DELETE action;
)
```

## INSERT

```
INSERT INTO tableName VALUES (valueA, valueB, valueC, ...);
```

```
INSERT INTO tableName(attributeA, attributeB, attributeC)
VALUES (valueA, valueB, valueC);
```

## UPDATE

```
UPDATE tableName SET attributeA = valueA WHERE id = valueID;
```

## DELETE

```
DELETE FROM tableName WHERE attributeA = valueA;
```

## SELECT

```
SELECT * AS resultTableName FROM tableName WHERE condition;
```

### Selections

- \*: Returns every attribute
- attributeName1, attributeName2, ...: Returns the requested attributes only
- tableName1.attributeName, tableName2.attributeName: Returns values from multiple tables

## ORDER BY

```
SELECT * FROM tableName ORDER BY column1, column2, ... ASC|DESC;
```

Orders the result table in the defined order. Can be in ascending (ASC) or descending (DESC) order. Usually defaults to ASC.

## DISTINCT

```
SELECT DISTINCT column1 FROM tableName;
```

The SELECT DISTINCT statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

## WHERE

```
SELECT * AS resultTableName FROM tableName WHERE condition AND|NOT condition;
```

Where uses a boolean condition and returns a resultTable where each row matches the given condition (or conditions)

## LIKE

```
SELECT * AS resultTableName FROM tableName WHERE condition LIKE pattern;
```

Like is used in the where condition and allows for checking of a specified pattern. There are two wildcard characters you can use in LIKE.

- % - Represents zero, one, or multiple characters (eg. %a% matches with Faisal)
- \_ - Represents a single character (eg. a\_ does not match with Rajhi)

## FUNCTIONS

```
SELECT FUNCTION(column) AS resultTableName FROM tableName;
```

Some predefined functions:

- COUNT(): Returns the number of rows in the supposed resultant table
- SUM(): Returns the sum of a numerical column
- AVG(): SUM()/COUNT()

## JOIN

```
SELECT a.column, b.column
FROM tableName1 a
(INNER|LEFT|RIGHT|FULL OUTER) JOIN tableName2 b
ON a.column = b.column;
```

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. There are a few types of joins. Joins also removed duplicates, a sort of built in DISTINCT

- INNER: Returns records that have matching values in both tables
- LEFT: Return all records from the left table and the matched record from the right table
- RIGHT: Return all records from the right table and the matched record from the left table
- FULL OUTER: Return all records matched from either table

## UNION

```
SELECT * AS resultTableName1 FROM tableName1
UNION (ALL|)
SELECT * AS resultTableName2 FROM tableName2;
```

Union combines the query from multiple SELECT statements. It automatically removes duplicates, to maintain duplicates, use UNION ALL.

## SUBQUERY

```
SELECT * AS resultTableName1
FROM (
    SELECT * FROM tableName2
) ;
```

Subqueries run a query on a result table instead of a table/set of tables from the database. Just us normal parenthesis ().

## GROUP BY

```
SELECT COUNT(column1), column2
FROM tableName
GROUP BY column2
ORDER BY COUNT(column1) DESC;
```

Usually used with functions, groups resultant values with one or more columns. For example, counting how many times each country appears in a table; using group by will place the count next to its respective country.

## HAVING

```
SELECT COUNT(column1), column2
FROM tableName
GROUP BY column2
HAVING condition //eg HAVING COUNT(column1) > 3
ORDER BY COUNT(column1) DESC;
```

HAVING exists because SQL people realized WHERE didn't work with functions. Use this with count, sum and all that jazz.

## 4.2 JDBC

We won't be covering details regarding JDBC and how to use it (such as using Connection objects or how to insert password) since that is more programming and should be covered already.

### 4.2.1 Statements

There are two types of statement objects; Statement and PreparedStatement. Statements are simply strings that get converted to an SQL Query, whereas PreparedStatements are a little more modular. PreparedStatements are a generic sort of strings, with a variable, usually denoted by ?, that can be replaced and dynamically altered. There are a few benefits to using PreparedStatements;

1. Reusable and dynamic, so it's faster
2. Separates logic of query and parameters, so it's clearer to code for
3. No need to convert parameters to Strings, so it's faster and uses less bandwidth
4. More generic, so it's type safe
5. Protects against SQL injection (mostly), so it's more secure

SQLInjection is a sort of security hax that injects SQL code as an input from the user and returns undesired results. Sanitizing results can help against such security risks, but it's much easier to at the very least use preparedStatements for most queries.

Note; concurrency and race conditions can also cause issues for databases that need to be accurate, such as banks. Databases can handle these things via "transactions", but also these issues can arise from code. Each concurrency issue needs to be handled on its own. A race condition needs synchronicity, but it could cause large queues and slow downs. The solution to these issues is on a system to system basis, but they should be noted.

## 4.3 ER Diagrams

Entity Relation (ER) Diagrams are graphical representations of tables, and are a database design practice used to, well, design a database. ER Diagrams are easily converted to SQL Create statements and tables. Shapes in an ER diagram mean different things.

- DIAMOND: A relation set (A shop needs customers and an inventory; the shop is a relation set)
- RECTANGLE: An entity set (The customers and inventory are entities)
- OVAL: An attribute (The customers have names; these are attributes)

We also use some special marks or labels that can have different meanings.

- Line: Defines a connection/relationship between two things

- Cardinality: Number placed next to the shape on a line. For example, [Students] 3 -; 1 [Course] means that Each Student has only 1 course, and each course has 3 Students.
- Underline: Underlining an attribute represents that it is a Primary Key
- Double Outline: An entity with a double outline is a weak entity; one that depends entirely on another.

Consider this example from the lecture. Every rental must record the date and time at which the rental started and finished, the details of the driver and the details of the car rented. Each car has its registration number recorded, the date it was purchased and a record of all maintenance work undertaken. For a driver, the company needs to have their name, driver's licence number, address and at least one telephone number.

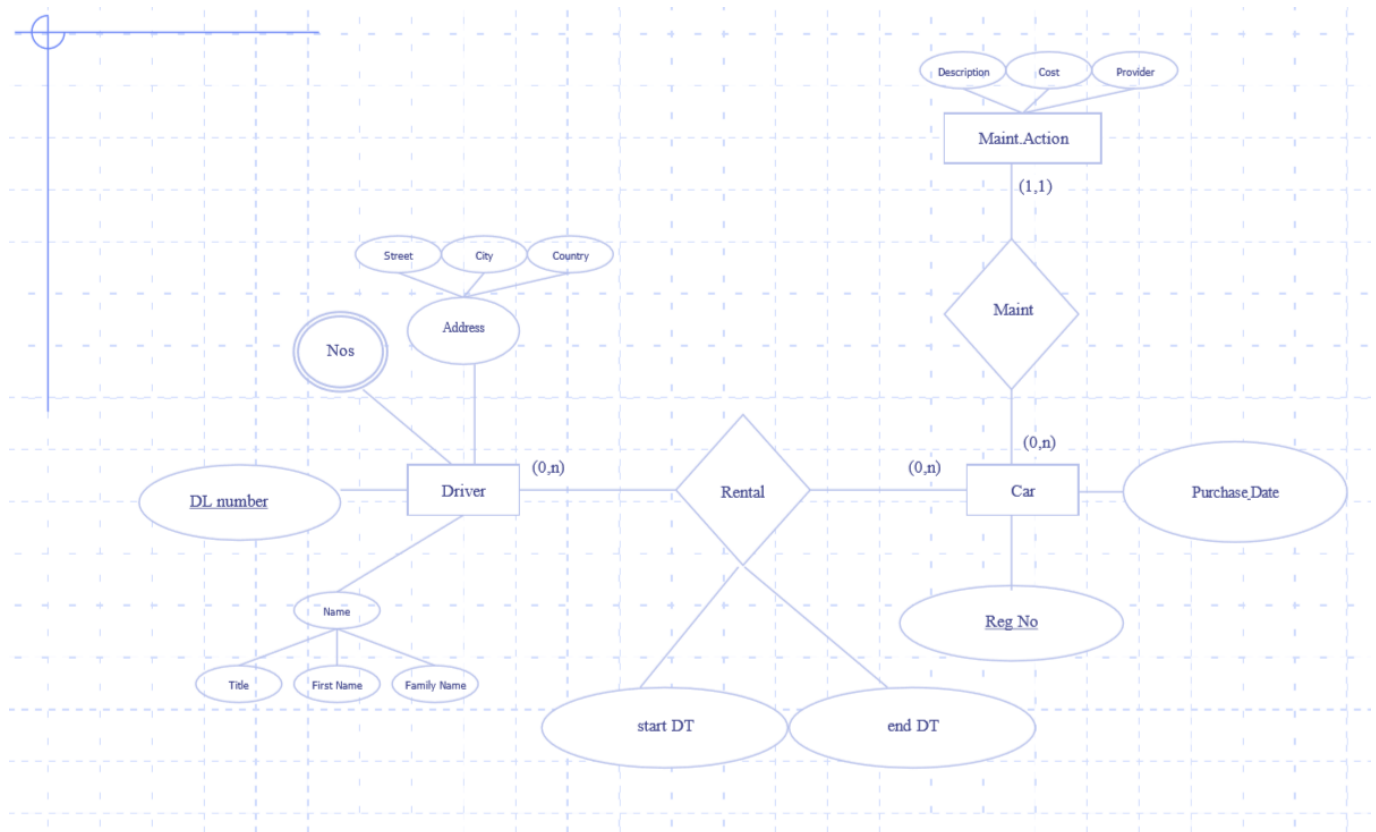


Figure 4.1: ER Diagram Example Solution

NOTE: Your ER diagram does not have to look like this to be correct, there can be multiple variations.

# Chapter 5

## Computer Systems & Architecture

### 5.1 Architecture

#### 5.1.1 von Neumann

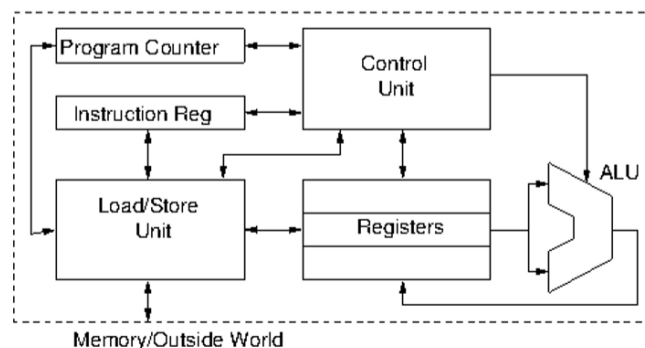


Figure 5.1: von Neumann Overview

von Neumann architecture is the more common CPU architecture used in modern times. There are a few components, as seen in the figure, and they each serve their unique purpose.

1. Main Memory: Logically separate from the CPU; it holds all the instructions and data that make up a program. This data is stored in distinct locations, referred to as "memory addresses."
2. Load/Store Unit: The interface between the CPU and the outside world; transfers data via the bus.
3. Registers: Local, fast storage that holds data currently in use. Each register holds one "word" of data. Does not hold instructions.
4. Instruction Register: Holds instruction. This is used by the control unit to configure the ALU. In usual design, only one instruction is active at any given time.
5. ALU: Arithmetic Logic Unit; where computations take place. Reads data from registers and writes results back into registers.
6. Program Counter: Special register that contains the memory address of the next instruction to execute.
7. Extra:
  - (a) Cache: Intermediate memory between CPU and Main Memory used for fast access of external memory
  - (b) IO Controller: Handles peripheral devices via an extension of memory addressing protocol or interrupt-based protocol. Connects via bus.

memory, load/store unit, registers, instruction register, alu, program counter,

### 5.1.2 Execution

#### Fetch-Decode-Execute

1. Inspect the program counter to find the address of the next instruction
2. Load the next instruction from memory into the instruction register
3. Update the program counter to point at the next instruction
4. Determine the type of instruction fetched
5. If the instruction requires data from memory, determine its address
6. Fetch any required data from memory into one of the CPU registers
7. Execute the instruction
8. Return to step 1 for next instruction

### 5.1.3 CPI and Execution Time

**CPI** CPI is the average clock cycles per instruction. CPU processor speeds are calculated as cycles per second. To find CPI, you simply take the total number of cycles  $IC * CC$  and divide by the total number of instructions  $IC$ . Giving us the formula:

$$CPI = \frac{\sum_i IC_i * CC_i}{IC}$$

.

#### Clock Time

**MIPS** Millions of Instruction Per Second is what MIPS stands for. To get this value, find the clock frequency and divide it by 1 million multiplied with the CPI.

$$MIPS = \frac{CPU(Hz)}{CPI * 1,000,000}$$

**Execution Time** To find execution time, the total time it takes to execute the instruction, we simply multiply CPI, IC and the clock time (or divide by clock frequency).

$$ET = \frac{CPI * IC}{CPU(Hz)}$$

## 5.2 MIPS Microarchitecture

This section deals with the MIPS microarchitecture and how to construct it.

- Microarchitecture: Detailed structure and organisation of the machine.
- Datapath: Collection of functional units which implement the instruction set.
- Control Logic: Configures the datapath in the correct way so that it implements the desired instruction.

### 5.2.1 Building Datapaths

**Instruction Fetch** The first step of the instruction cycle is to get the address of the next instruction and then fetch the instruction itself. To do this, we must send the contents of the

program counter to memory, as an address, and bring the contents of that address into the CPU as the next instruction. We will need the Program Counter, Main Memory (usually physically off-chip) and an Instruction Register. We will also need an "adder" of sorts to increment the Program Counter.

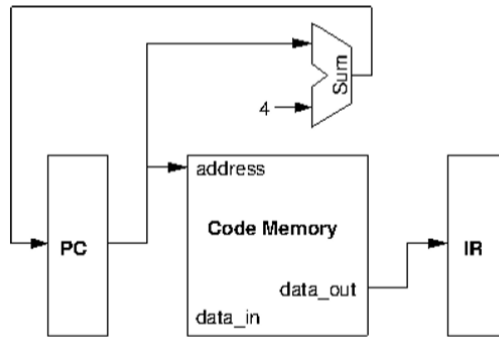


Figure 5.2: Instruction Fetch datapath sketch

**R-Type Instruction** R-Type instructions are instructions where all data values used are in the register. Each 32-bit R-Type instruction is partitioned as follows:

- 6-bits: Opcode (machinecode representation of the instruction)
- 5-bits: Source 1
- 5-bits: Source 2
- 5-bits: Destination
- 5-bits: Shift
- 6-bits: Funct (For instructions that share an opcode, the funct parameter contains the necessary control codes to differentiate the different instructions.)

As such, 15 of these bits are dedicated to selecting registers that are involved in ALU computations.

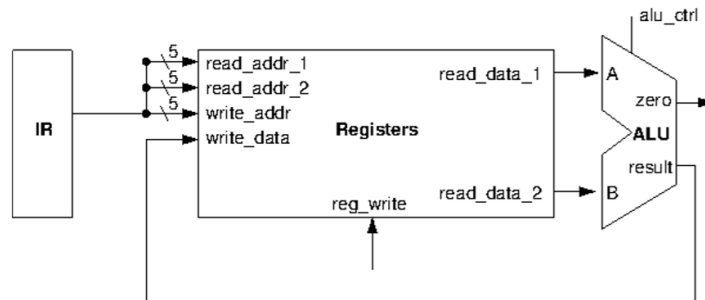


Figure 5.3: R-Type instruction datapath

**Load/Store Instruction** Load/Store Operations are more complex than ALU instructions; they need to have R/W access to registers, memory and be able to use the ALU to calculate addresses.

- 6-bits: Opcode
- 5-bits: Base Address (Register containing such address)
- 5-bits: Source/Destination (Register containing such address)
- 16-bits: Address Offset



```
lw $t0, 8($sp)
```

In this example, lw is the opcode, \$t0 is the destination, \$sp is the base address, and 8 is the offset. One thing to note, is that the address offset is 16 bits, but the ALU processes values at 32-bits. Due to how we represent negative values, and the fact we want to be able to offset in the negative, we will need a Sign Extension Unit to increase our offset to 32-bits before moving it to the ALU.

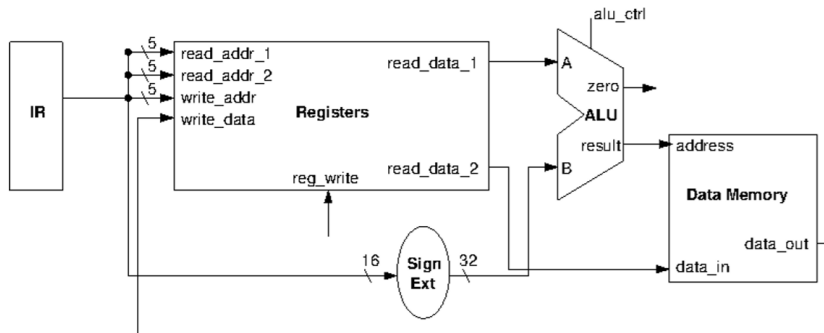


Figure 5.4: Load/Store Operations Datapath

**Branches and Jumps** Branches and jumps essentially achieve the same result; they must both be able to change the contents of the program counter. However, branches require a comparison to be done beforehand, whereas a jump does not. Jumps are a small addition to the structure of a branch datapath, so let's look at branches first.

- 6-bits: Opcode
- 5-bits: Base Address (Register containing such address)
- 5-bits: Source/Destination (Register containing such address)
- 16-bits: Address Offset

This is the same instruction structure as Load/Store. However, instead of accessing the memory to, well, load/store, it instead needs to add/increment the PC. This is shown in the diagram.

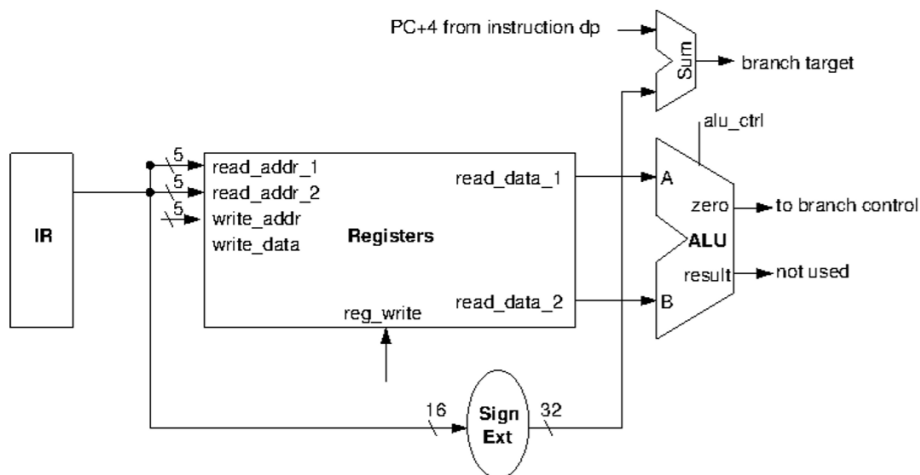


Figure 5.5: Branch Instruction Datapath

**Multiplexing** As you can see, the different instruction can use the same components. This is a problem if we want to combine them into one big boy datapath. To do this, we use a technique known as multiplexing. Multiplexers allow us to transmit multiple signals along the same channel, selecting which one to pass. A multiplexer has three sets of wires, an input, an output, and a control. For every  $N$  control wires, we can choose between  $2^N$  inputs, so a 2-1 multiplexer has 1 control, a 4-1 has 2 control, an 8-1 has 3 control, etc. Let's look at this one bit multiplexer example;

```
if (C==0)
    Q = x + y;
else
    Q = x + z;
```

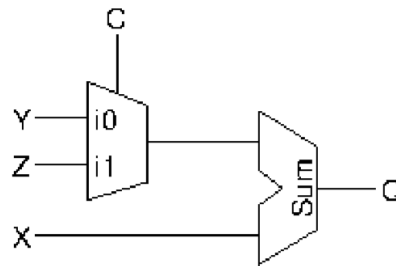


Figure 5.6: Multiplexer Example

### 5.2.2 Combining Datapaths

**ALU and Load/Store** In an ALU instruction, both inputs to the ALU are from the registers, and the registers provides data to just the ALU. In a Load/Store instruction, the second ALU input is from the sign extender, and the data written to a register is from the Data Memory. We will need two multiplexers and two control lines, referred to as **alu\_src** for the ALU input and **mem\_to\_reg** for the register writing.

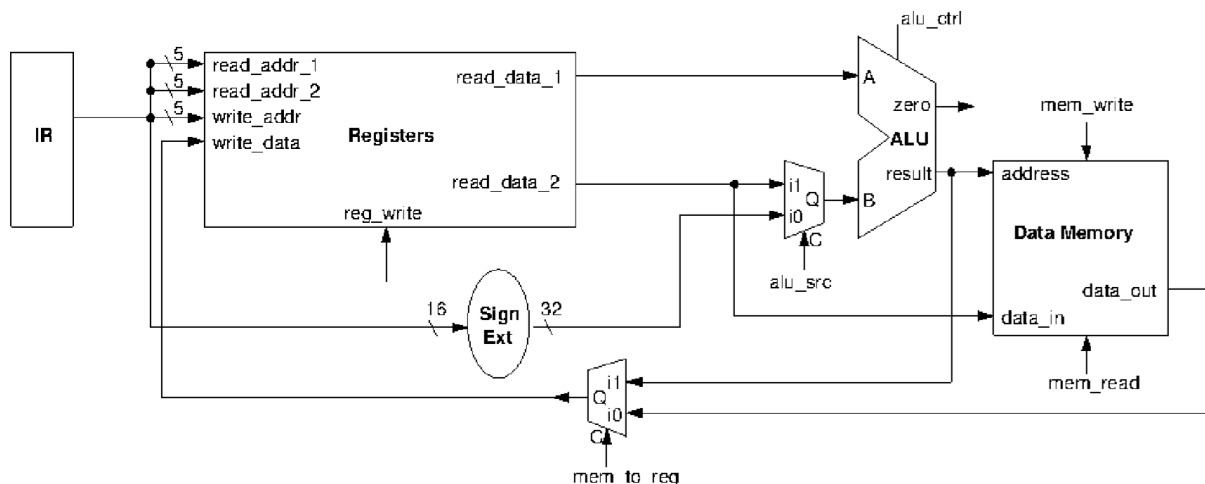


Figure 5.7: ALU and Load/Store Integration

**Instruction Fetch and Branches** Instruction fetch doesn't use any of the same resources as ALU or Load/Store instructions, so it's easy to integrate. We will look into integrating branches with them. Either the PC incremented with 4 is set as the new PC, or the added branch offset is added. 2 to 1 can only mean one thing, multiplexing. A multiplexer is added after the second adder component from the branch datapath, using a **pc\_src** control wire. The rest of the components in the branch datapath are not unique to it, and does not require any other changes or multiplexing.

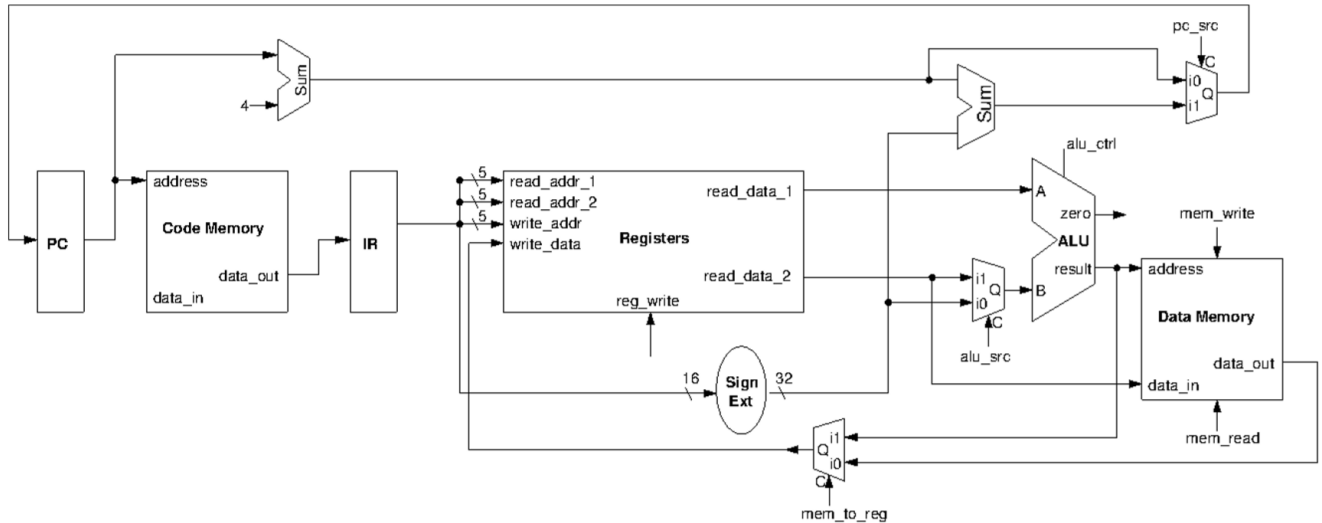


Figure 5.8: Mostly integrated MIPS datapath

### 5.2.3 Control

In the datapath there are a few control wires added; the Main Control unit is in charge of decoding the instruction and transmitting the proper control signal. We need to know what each control signal means first.

- **reg\_write**: If true, allows writes to registers
- **alu\_src**: Chooses between register (False) or sign extension unit (True) input for ALU
- **mem\_to\_reg**: Selects whether ALU output (False) or memory (True) is sent to registers
- **reg\_wsrc**: Selects whether registers write\_address is bits [20-16] (True) or [15-11] (False) of the instruction
- **pc\_src**: Selects whether PC+4 (False) or PC+4+branch (True) is sent to PC
- **jump**: If true, sends jump address to PC; if false, sends output to pc\_src to PC
- **mem\_read**: If true, read from memory is permitted
- **mem\_write**: If true, writes to memory are permitted
- **alu\_op**: Control bits for the ALU

We will look at two ways to implement CPU Control in our every day von Neumanns. We have **Single Cycle Control** that uses a stateless logic whose input is the 6-bit opcode and whose outputs are the control signals. This is useful if the instructions can be completed in one cycle. In reality though, some operations will take multiple cycles due to the different operations, such as memory access, being slower than others.

Instruction	lw	sw	r-type	beq	j
reg_write	1	0	1	0	0
alu_src	1	1	0	0	X
memtoreg	0	X	0	X	X
reg_wsrc	0	X	0	X	X
branch	0	0	0	1	X
jump	0	0	0	0	1
mem_read	1	0	0	0	0
mem_write	0	1	0	0	0
alu_op	00	00	10	01	XX

Figure 5.9: Control Requirements Table per instruction

## 5.2.4 State Control

Finite State Machines, rather than being stateless, has, well, states. We learned two ways of implementing them, Mealy states machines and Moore state machines. In a Mealy state machine, the outputs and the next state are a combinatorial function of both the state and the inputs, whilst in a Moore state machine, the outputs are a function of the state only while the next state depends on the inputs and current state.

What this means is that the output of a Moore machine can only change on the clock edge, whilst the output of a mealy machine can change asynchronously when the inputs change. Which of these is desirable is based on the application. Here are the main factors to consider when designing a FSM:

- Determine how many states are needed and how to enumerate them
- Determine the sequence of state transitions
- Write down truth tables for the combo logic
- Implement the logic

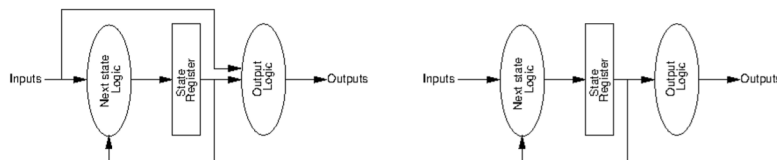


Figure 5.10: Mealy VS Moore (brawl of the century)

In general, these are the major differences between Mealy and Moore.

1. Mealy machines tend to have fewer states
  - Different outputs on arcs rather than states
2. Moore machines are safer to use:
  - Outputs change at clock edge (delayed one cycle)
  - In Mealy, only when logic is done is the output changed, causes problems with interconnected machines
3. Mealy machines react faster to inputs
  - React in same cycle, no need to wait for the clock
  - In moore machines, more logic may be necessary to decode state into outputs; more gates == more delays

## 5.3 Optimisation

### 5.3.1 Pipelining

**Overview** Pipelining is defined as (thanks google): a form of computer organization in which successive steps of an instruction sequence are executed in turn by a sequence of modules able to operate concurrently, so that another instruction can be begun before the previous one is finished. Normally, each instruction set is executed before the next one begins. However, we can instead make it where, once a task is finished, the next one is fetched before the original is completed, think in a factory production-line fashion. While each instruction will take the same time (or longer, as all instruction have to be done before any instruction can proceed), the throughput overall is much higher.

**MIPS Pipeline** The MIPS pipeline has registers added to store intermediate results plus any additional control information.

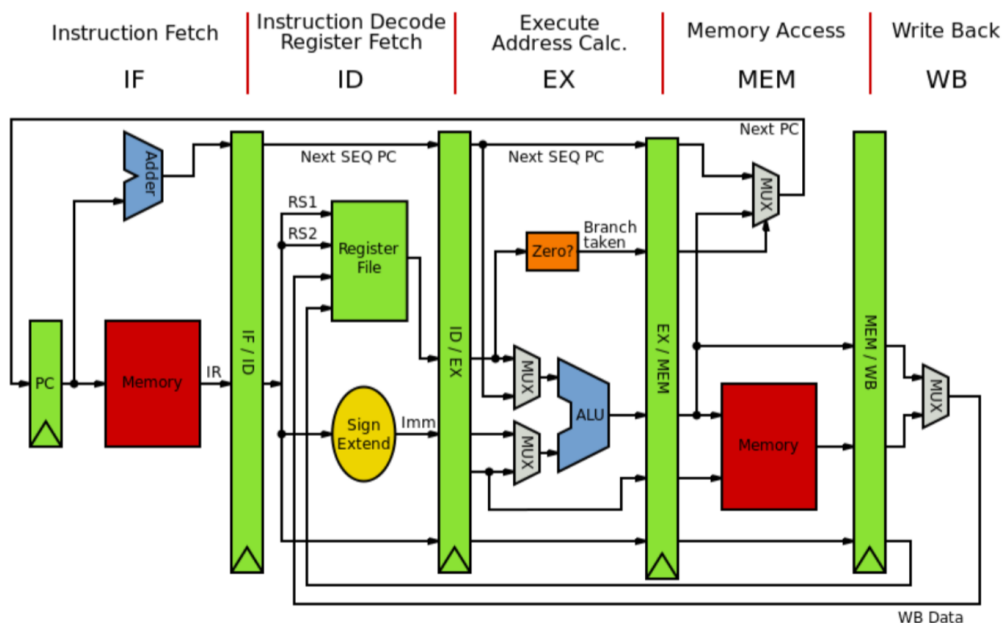


Figure 5.11: MIPS Datapath Pipeline

### 5.3.2 Caches

**Overview** The cache is a moderately fast, medium-sized memory which acts as a buffer between the main memory component and the registers. Caches are used because of the two basic properties of computer programs.

1. Spatial Locality: When an item is referenced, you are likely to access nearby items soon.
2. Temporal locality: When an item is referenced, you are likely to access it again soon.

At the most fundamental level, caches are just a block of memory. However, caches need to store a copy of a value from the main memory and its address in the main memory. This address is often called the cache tag. When the CPU is accessing the main memory, it goes through the cache and compares the tag with the address its looking for. It can either hit or miss; because this check is done every time, a cache will add more time than without a cache on a cache miss.

**Cache Associativity** One of the biggest factors affecting cache performance is its degree of associativity. This is the number of different locations that an item of data can occupy in the cache. One of the most common cache designs is the direct-mapped cache AKA one way associative. Data from each address in main memory can only go in one location in the cache, making access time to the cache fast. The way this works is, a subset of the address is hard-coded into the cache (eg. 111xxxx) The remaining bits are checked against the tag for the entry.

The main issue with direct-mapped caches is that two items of the same subset (111x, 111y) cannot be stored simultaneously, resulting in potential thrashing, where entries in a cache continuously swap, destroying any performance benefit you dreamt of. The best way to potentially deal with thrashing is in an n-associative cache, where each address maps to  $n$  cache entries (eg. 10, 01) and multiples of them can exist. A cache where the  $n$  is the same number of locations in the cache is said to be fully associative. Positive include no thrashing and lower main memory accesses, negatives include slower access time to accommodate for the cache.

**Cache Refill Strategies** Following a cache miss, temporal locality states the new data accessed in the main memory should be moved to the cache. We use a data structure known as least-recently-used (LRU) queue, which removes the least recently used item when inserting an item to a full list. Ofcourse, this might not be the best plan, but it is up to the designer to figure out the best refill policy as it can differ between machines, code, and any other cache usage factor.

**Cache Writing Strategies** Reading from a cache is hit or miss, pretty simple.

**Performance Benefits** Cache performance is a calculation that is really simple. You take the proportion of instruction time used on on memory access, and convert the amount that would be a hit on the cache to the cache access time, and just like magic, you have the new time.

$$T = N(t_{cyc} + m(t_{cache} + (1 - h)t_{mem}))$$

- $N$  = Number of instructions
- $t_{cyc}$  = The cycle time
- $t_{mem}$  = Memory cycle time (Additive to cycle time)
- $t_{cache}$  = Cache access time
- $m$  = The proportion of memory access instructions
- $h$  = Cache hitrate

## 5.4 Number Representations

### 5.4.1 Types

**Base Two** Numbers in computers are represented in binary, base two, format. Normally, our normal understanding of numbers is in Base Ten.

$$3 = 1 \times 10^0$$

but this can also be represented in Base Two

$$3_10 = 11_2 = 1 \times 2^1 + 1 \times 2^0$$

. In terms of integers at least, this means that the maximum size we can represent in  $n$  bits is

$$2^n - 1$$

so a 32-bit value is, at most,  $2^{32} - 1$ .

**Representations** Ofcourse, this method of representation using twos also works for decimals, but where in a byte do we store decimals? In the fixed point representation of real numbers, the decimal is implied. For example, assuming a 16-bit piece of memory, we imply that the integer portion will cover the first 8 bits and the decimal portion the remaining 8 bits. As such, the computer will recognize the values with no manual decimal placed. Modern computers now use the floating point representation.

$$V = M \times 2^E$$

Floating point uses a mantissa  $M$  and an exponent  $E$  to store its values. Part of the bits, for example 4 bits in an 8-bit system, are used to store the mantissa and the rest represent the exponent. In an 8 bit system as a result, you can represent a value like 112 as

$$01110111 \rightarrow 0.111 \times 2^{0111}$$

**Negative Value** Negative values are represented using something we like to call Two's Complement. What this simply means is, the left-most bit is "0" for the positive variant. We know that each value we can represent should have a value in the same field that, when added together, form 0. We represent this negative value by inverting every bit and adding 1 to the LSB.

$$0101 \rightarrow 1011$$

. The system reads the left-most bit as a 1, so we know its negative, the rest follows.

## 5.4.2 Arithmetic

			1	1	0
	+		1	0	1
			<hr/>		
Carry	1				
			<hr/>		
Sum	1	0	1	1	

Figure 5.12: Arithmetic example for base 2

Adding bits works the same as adding and subtracting the numbers in base 10. For example, if we were to add 0110 and 0101, the result would be 1011 as seen by the figure. When our column adds to a total of 2, a one is carried on column to the right and the 2 becomes a 0. If the total becomes 3, a 1 is still carried, but the column becomes a 1. For subtraction, you can add the base value and the two's compliment of the other value. This simplifies the process greatly.

### 5.4.3 Hex Representation

Decimal Number	4-bit Binary Number	Hexadecimal Number
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10 (1+0)
17	0001 0001	11 (1+1)
Continuing upwards in groups of four		

Figure 5.13: Deciaml  $\rightarrow$  Binary  $\rightarrow$  Hex Representation

## 5.5 Logic

### 5.5.1 Logic Gates

The basic building block of our CPUs today are transistors. Transistors are made up of three parts, a gate, a source, and a drain. They are used in circuits; it is enough to understand that they are switches that are controlled by the gate voltage. Transistors process bits (0 voltage and  $> 0$  voltage). The output changes based on how the circuit is built. For example, we have the NOT gate which takes a signal and inverts it on output. For simplicity, gates are drawn in an abbreviated form as seen in Figure 5.14. If you've done Minecraft Redstone, you might find yourself at home on this section. Sadly real life is a little more complicated.

**Combination Logic** A general term for blocks of digital logic which contain no kind of memory. For each  $n$  inputs, there are  $2^n$  possible output states. Each of these outputs can be represented by a basic boolean algebra (eg.  $A \wedge B$ ) and written into a circuit as such.



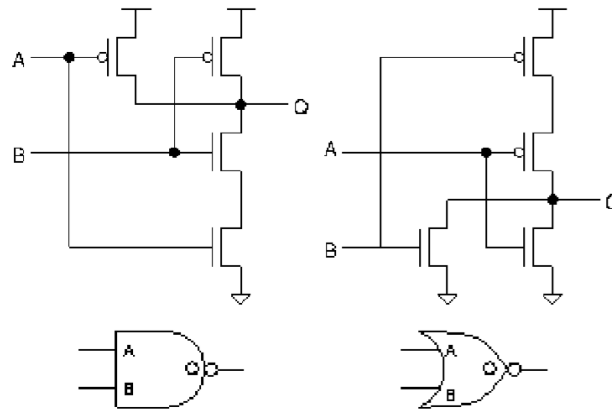


Figure 5.14: NAND and NOR gates

### 5.5.2 Latches

**Latch** A latch is a circuit that has two stable states and can be used to store state information. The circuit can be made to change state by signals applied to one or more control inputs, and will have one or two outputs. It is the basic storage element in sequential logic. NAND and NOR (pictured) are "programmable inverters"; cross coupled NAND/NOR gates are used. We will be covering two types, SR (Set-Reset) and D(data/delay).

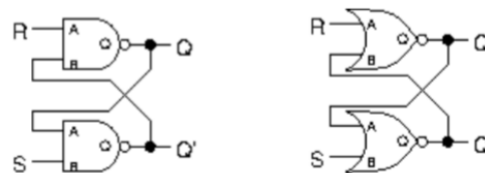
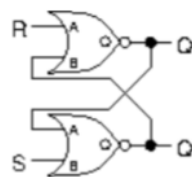


Figure 5.15: NAND and NOR Crosscoupled

**RS NOR Latch** The most basic latch is the RS NOR Latch constructed from a pair of cross coupled NOR gates, as pictured. Due to the nature of the setup,  $R=S=1$  should be avoided.



R	S	Q	Q'
0	0	$\neg Q'$	$\neg Q$
0	1	1	0
1	0	0	1
1	1	0	0

Figure 5.16: RS NOR Latch

**D Latch** To avoid the case of  $R=S=1$ , we can use a D Latch to change the input into the RS Latch. A simple addition to the circuit removes this issue. At  $C=0$ ,  $R=S=0$ , and  $C=1$ ,  $R=\neg D$

and  $S=D$ , giving the RS control to the D value.

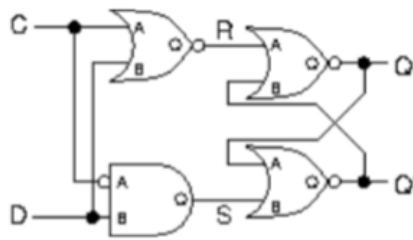


Figure 5.17: D-Latch

**Edge Triggered Flip Flop** The RS latch responds to the data inputs (S-R or D) only when the enable input is activated. However, it is desirable to limit the responsiveness of a latch circuit to a very short period of time instead of the entire duration that the enabling input is activated. We can do this using an edge triggered flip flop. By placing some latches with some neat logic, we can obtain this behaviour. The first half becomes transparent at  $C=0$ , and the second half at  $C=1$ . When  $C: 0 \rightarrow 1$  the second stage "captures" output of the first stage. This way,  $Q$  is stable for the whole clock cycle.

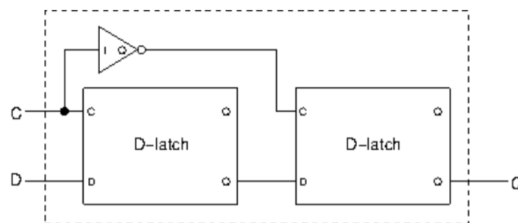


Figure 5.18: Edge Triggered Flip Flop

### 5.5.3 Datapath Components

**Multiplexer Construction** Multiplexers are used commonly in the MIPS datapath. Let's look at the logic of a 2-1 multiplexer with inputs  $i0$ ,  $i1$  and  $C$  with output  $Q$ .

$$Q = (C \wedge i1) \vee (\neg C \wedge i0)$$

As you can see, this is a simple construction of two AND gates followed whose outputs are the inputs of an OR gate. For larger bits like 32, we use 32 single-bit multiplexers that share the same selection signal  $C$ .

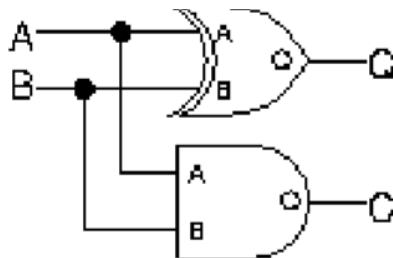


Figure 5.19: The circuit for a half adders; an XOR outputs the addition and an AND outputs the carry

**Adders and Half-adders** Thanks to our good friend google, we can find the definition of an adder as a unit which adds together two input variables. A full adder can add a bit carried from another addition as well as the two inputs, whereas a half adder can only add the inputs together, throwing its carry to the shadow realm (a different output signal). When thinking of half adders, we want to find a way to manage handling the carry going in  $C_{in}$  and the carry going out  $C_{out}$ . This can be handled easily by using two half adders and an XOR with the carries from the two adders. We can then use half adders for a 32-bit adder, with two halves creating one whole. Each bit uses a full adder, and a half adder is used for the LSB, since a carry there indicates an overflow.

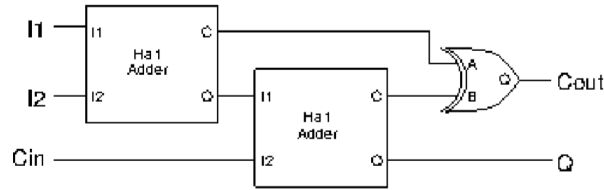


Figure 5.20: The full adder circuit generated from two halves

**Adder to ALU** From here, all we do is take our 32-bit adder, stick in some AND and OR gates (to handle the appropriate operation) and some multiplexers to know which operations are required. This creates an ALU.

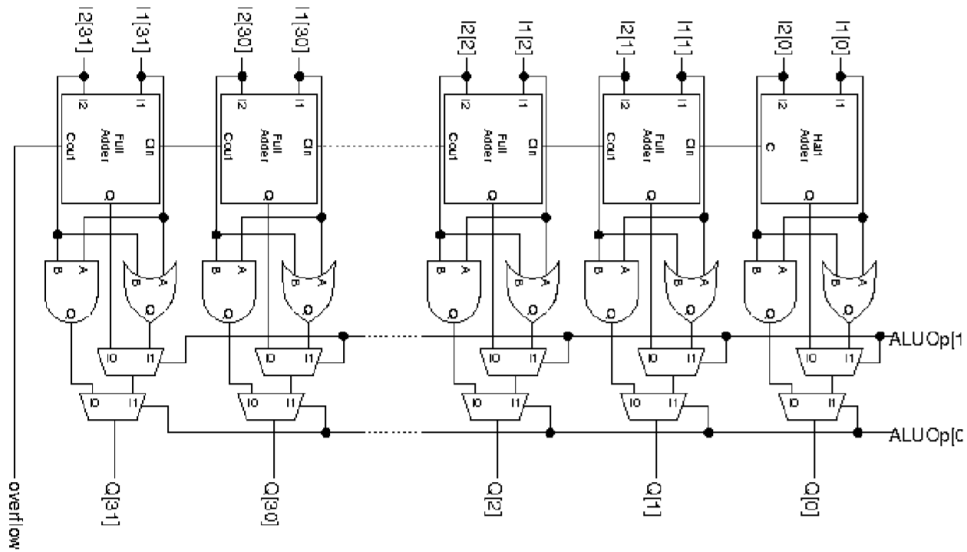


Figure 5.21: General 32-bit ALU design with the adders

# Chapter 6

## C/C++

### 6.1 C

This section deals with the basics of C and things you will use a lot in C.

#### 6.1.1 Data Types

There are three major types of data in C; int, float, and char.

**int** Integers in C are 32-bits long, represented by the 2's complement system, so they can represent values  $[-2^{31}, 2^{31} - 1]$ . You can use **unsigned int**, which removes the negative from the int, representing values  $[0, 2^{32} - 1]$ . If we wanted to represent larger numbers, we can use **long long int**, which is 64-bit int with values  $[-2^{63}, 2^{63} - 1]$

**float** Floats in C are of size 32-bits; they are represented in IEEE 754 single precision, hence 15 and 15.0 have different representations. The range of a float is  $[1.401298 \times 10^{-45}, 1.401298 \times 10^{38}]$ . We use **double** for 64-bit long double precision values.

**char** Chars are 1-byte integers. It uses 8-bit ASCII encoding to represent characters.

**Memory Allocation** Each data type is allocated memory of its respective fixed size. When allocated, there is a garbage value until a value is assigned. Constants, declared as **const** are stored in read-only memory, causing errors when attempting to change the value. Pointers have a size of 4 bytes.

#### 6.1.2 Function Calls and Scope

**Functions** Function calls are executed on the stack. For each function call, a stack frame is allocated. The stack grows upwards from low addresses to high addresses. The scope of values is within their own function, so a value initialized in one function is out of the scope of another function. However, with the nature of the stack, sometimes uninitialized 'garbage' values are not so garbage. This is because the second function is in the same location in the stack as the previous function, so if they have the same types of variables, they could inherit the same value. This can be seen in the figure, where the **d** is initialized with the value of **a**.

```

void fun1() {
    int d;
    printf("d = %d\n", d);
}
void fun2() {
    int a=10, b=20;
    printf("a = %d b=%d\n", a, b);
}
void main() {
    int a;
    fun2();
    fun1();
}

```

Figure 6.1: Function examples

**Recursion** Recursion is possible in C. At a low level, each recursive call creates a new stack frame. If there is no stop condition, and with the usual finite stack, there will be overflow and the program will crash.

**Preprocessor Directives** The `#` used at the start of a C file is referred to as a C preprocessor directive. They allow the inclusion of:

- Header Files
- Macro Expansions
- Conditional Compilation
- Line Control

A macro is a fragment of code which has been given a name. **`#define NAME tokens`** Think in the style of finals or constants; you can do **`#define MAX 100`** and use the word MAX in the rest of your code, the compiler will switch out the macro with the value it is defined as. You can also define functions in macros, but should be used for basic functions like multiplying two values and returning the result.

### 6.1.3 Arrays



An array is a data structure that stores a fixed sized, sequential collection of same typed elements. Arrays start at 0, and increment. In memory, they are stored, unsurprisingly, from low address to high address. This is because arrays are accessed by taking the base address and adding onto it the index with the distance between addresses of the object. (eg, an object of 32 would have addresses 0, 32, 64, etc.)


Arrays can be accessed with `a[index]`, where `a` is the name of the array. In C, `a` cannot be assigned to anything. It is equivalent to `&a[0]`, the address of `a[0]`. Doing `*&a[0]++` is the same as `a[1]`. Passing an array to a function is simply just passing the address of its first value, similar to a pointer if you will. You also can't return an array, so you either have to change the values directly in the address or return a pointer.


A note, the compiler does not know the size of an array during compile time, so arrays can potentially lend themselves to creating potential issues in your code. Array sizes also can't be dynamically changed. When two arrays are created one after the other, the 2nd array address is directly after the end of the 1st array, similar to initializing two variables of any type. Since array access isn't checked and garbage isn't collected, you can access values you don't mean to access with an array.

## 6.1.4 Pointers

**Pointers** A pointer is an address variable. It stores the address of a data item. The type of the pointer is the type of the object it points to. Data pointers have size of 4 bytes in a 32-bit system and size of 8 bytes in a 64-bit system. When using pointers, there are two key characters we need to understand; '\*' and '&'. \* is used to declare pointers and obtain the value its pointing to. & is used before a non-pointer variable to obtain its memory address. This can be seen in the figure.

```
int number = 10;
int *number_pointer;  Pointer declaration
number_pointer = &number;  Pointer initialization

float price = 5.6;
float *price_pointer = &price;  Declaration and
                                initialization

char symbol = 'A';
char *symbol_pointer = &symbol;  Declaration and
                                initialization

return 0;
```

Figure 6.2: Pointer Declaration and Initialization

One way to think of pointers vs variables, is to think of a variable as something that contains stored data and requires a symbol to obtain its address, whereas a pointer is something that contains an address and requires a symbol to obtain its stored data. Multiple pointers can point to the same address, and this can be used to pass data into functions.

**Pointers in Functions** When passing variables into a function, it does not pass the declared variable, instead it passes the data into a new memory address, so assignments in the function will not change the variable passed in during the function call. However, using the magic of pointers, we can change that. Instead, all we have to do is pass the memory of the variable, and in the function create a pointer that points to that address and change its value by obtaining it from our pointer. This nifty trick is how we pass arrays into a function. An example is seen below. The final values of x and y in the main function will be 10 and 4, respectively.

```
void foo(int *x, int y)
{
    (*x)++;
    y++;
    return;
}

int main()
{
    int x, y;
    x = 9;
    y = 4;
    foo(&x, y);

    return 0;
}
```

You can also return pointers from functions. This is simple to do.

```
int *foo()
{
    int x = 10;
    int *p = &x;
    return p;
}
```

There is also some syntax to be careful of with pointers. For example, `(*p)++` and `*p++` are not the same thing. The first one first pulls the value stored in the pointer's memory, and applies '++' to the value. The second one increments the memory address first, then pulls out the value stored in the memory. Parenthesis are important!

**Type of Pointer** When pulling a value out of a pointer, it's takes the appropriate amount of bits it needs from the memory address pointed to. So if you have a char pointer pointed to an integer, it will pull some bits from the int and represent it as a char. In addition, when increment the pointer with `p++`, it will scale it's memory address based on the type its pointing to. `p+1` is the same as `p + 1*scaling_factor`, this scaling factor is based on the size of the type the pointer is pointing to. This is behavior seen in arrays.

**Arrays and Pointer Movement** It is important to realize that arrays are simply pointers, as mentioned earlier. If we have a pointer `p = &a[0]`, then doing `*p++` is the same as `a[1]`. As such, pointers can end up pointing to places we probably don't want it pointing. Pointers can also be added together, subtracted, etc. What it does is offset the memory address based on the operation. Messing around with addresses can easily lead to broken, buggy code, but it's still useful for managing code.

**Allocating Memory** When declaring arrays and pointers, it will allocate memory, but we don't always know how much memory we want to allocate, or perhaps we want to do it sort of in code to not be wasteful. How do we set aside memory for a pointer or array? How do we free up our memory once we are done with the data? This is easily achieved by using the functions `malloc` and `free`.

**malloc(size)** allocates a space in memory equal to the size passed in, and returns the address of the first position in memory allocated. If there isn't enough space in memory for allocation, it will return a null value. You can check for the pointer assigned with its address if it is null or not to manage the code. This memory is obtained from the heap.

Finally, to free up space, all we need to do is do a good old **free(pointer)** command, and it will free all memory associated with that pointer or array and return back to the usable/free heap for later mallocs. The pointer will then have garbage value, so it is suggested to set it to null and do null checks to ensure the free went through later and avoid odd bugs caused from freeing. This also helps to be mindful of double free errors, where we free the same pointer twice. This causes the address of the pointer to be seen as "free" for allocation twice, meaning it can be allocated later, and then replaced without being freed.

### 6.1.5 User Defined Types

C doesn't have objects. So how are we supposed to do things that work super well in user defined object form? Oddly specific question, but there is an oddly specific answer. C has the ability to create **struct** and **union** instead of classes.

**Structs** Assume we have a table of students, with their name, id number and their marks. These values all have different types (`char*`, `int`) but they can be logically grouped. We can create our own type and call it struct. Let's get some definitions out of the way. Structs are used to pack logically related data items and can be treated as a single item. Think of them as a data container. The data items packed in a struct are referred to as members. These members can be of different types. They can even be structs!

```
struct tag {
    type1 member_1;
    type2 member_2;
    :
    typen member_n;
};
```

Figure 6.3: Declaring a new struct

**Memory Allocation** The layout of a struct in memory is fairly straightforward. Structs in memory are allocated the size of each member in the order they are declared in the struct. This can result in some padding, for example one char followed by a pointer could result in three bytes being wasted/padded, if we assume the pointer to need be in a multiple of 4. (This is because the char takes index 0, and index 3 is the next multiple of 4, so the three bytes are padded to allow the pointer to be allocated in the same general area as the char in the struct).

**Struct Pointers** Structs are treated like a data type, so you can create a pointer for a struct. The syntax is as follows, **struct tag \*p**. A note here is that all structs need the word "struct" before their name. You can avoid this by adding **typedef struct tag TAG**. This defines out "struct tag" as just "TAG" for all uses, allowing us to declare variables as **TAG \*p**. Less verbose, so its usually a good idea.

**Accessing Members** There are two ways to obtain the member from a struct. Assume we have a struct in the variable "t1" and the pointer "p" and has a member "name". You can access it using the usual way to access public variables in the likes of Java; as **t1.name**. With a pointer, however, there are two ways to accomplish this. You can get the value out of a pointer and access using a period from there; **(\*p).name**, but there is a shorthand way. You can just do **p → name**. It's less to type, and is equivalent to the period syntax.

```
union item {
    int id;
    float price;
    char name_of_item[15];
};
struct member {
    int id;
    char name[15];
};
```

Figure 6.4: Struct VS Union

**Unions** Unions are similar to structs. They have names, can be typedef'd, contain members and all that stuff. They differ from structs in how it appears in and manages memory. Where structs allocate memory for each of its members, unions only allocate as much space as their largest member, with some potential padding. As seen in the figure, the struct has size at least 19, whereas the union will be smaller at size at least 15, the size of its largest member, despite having a combined size of 23.



**Nested Types** A special note, a struct/union cannot contain a member variable of their own type. However, a neat work around is that you can have a member that is a pointer to its own type. Using these pointers, you can create recursive data types such as binary trees and linked list. We did an assignment on this, and it is best to look back at that for specifics on how to implement such features.

### 6.1.6 Optimization

When we talk about optimization, there are really three different directions we can take, and the desired direction is dependent on context. We have flexibility, time, and space.

- Flexibility: How flexible is our code? How many different situations can it handle, and how well can it handle them?
- Time: How fast does our code achieve what it was meant to achieve? Can it be *even faster*?
- Space: How much memory does our program use it any one time? Can we reduce it without changing what it was meant to do?

In an ideal world, we would optimize all three to their full extent. Sadly we don't live in an ideal world, so some sacrifices in some areas have to be taken to optimize for others.

**Speed Optimizations** When it comes to speed, we can compare speed by using ratios.  $\frac{Speed_{old}}{Speed_{new}}$  and speed is usually a measure of time from the start of the function or program to the end. Ofcourse, this just tells us how to measure time, but how do we actually improve the speed of our programs? There are many ways to go about this, but the first step is to identify costly functions. The simplest example would be looking at two functions, one that adds matrices and the other that subtracts them. The matrix multiplication has a higher complexity, so it could be a cause for concern.

Speed optimization tactics:

1. Use a better algorithm: Straight-forward, some algorithms are just faster.
2. Use inline functions: For small functions, we can macro them and let the compiler just apply the function's operations without the overhead in the stack from function calls. You can achieve this using the "inline" declaration instead of a macro, in fact inlines are usually preferred. Be careful when inlining larger or more complex functions, because inlining could then just make a giant compiled code that takes more space and has higher load times.
3. Loop unrolling/jamming: Loops add an overhead on their check conditions on each iteration. You can reduce this by going through multiple iterations at once (5 at a time instead of 1). You can also attempt to combined two loops together even though they might do different things. Since they both will loop, combining into one completely removes the loop overhead from one of them; this is called loop jamming.
4. Pass by references: Passing by values is slower than passing by references, because new memory is allocated for the new values and things like this take time (and additional space!)
5. 5. Avoid recursion: Recursion kills your (stack) frames
6. Avoid dynamic memory allocation: Constant access and change in memory results in slow access operations, as well as overhead from the memory manager knowing which heap space is used up or free.

These are just some optimization techniques. In practice, code optimization is on a code to code basis.

## 6.2 C++

This section deals with features in C++ and why its used more than C.

### 6.2.1 Introduction

**C++** C++, as the name suggest, is an increment to C. While it has all of the features C has, it also has some bonus features. It's primary purpose is to make C an object oriented programming language, adding classes, constructors/destructors, and various other features for polymorphism and memory management.

**Reference Variable** One thing we can do in C++ is creating a reference variable. Reference variables are similar to pointers in that you can pass them into functions to pass by reference, but you do not need to dereference it with (\*p), as it is already dereferenced by nature. This allows you to bypass on using pointers in the case you only want to pass a reference variable. The syntax to declare a reference is **int &r**; much like declaring a pointer type but using & instead of \*.

**Overloaded Functions** In C, a function's signature is its name, so you cannot have two different functions with the same name even if they had different parameters. This is not the case with C++. A function's signature is comprised of three things.

1. Name
2. Number of Parameters
3. Type of Parameters

Meaning the functions **void foo()** and **void foo(int a)** have different signatures. Note that the return type of a function is not a part of its signature, meaning **int foo()** and **void foo()** have the same signature and will cause an exception.

**Classes** C++ has the ability to have classes for the object-oriented programming. When declaring a class, you can define members and functions in either the private or the public region. By default, members and functions are private, so the private region can be omitted.

```
class proc_info{  
    float processor_data;  
    int ram_data;  
    int disk_data;  
  
    public:  
    start_process();  
    get_disk_data();  
};
```

Figure 6.5: Private members and public functions example

**Constructor** C structs are not exactly objects. They cannot be initialized at time of declaration, they have to done at separate lines. C++ classes, on the other hand, can be initialized at time of declaration. We do this using constructors. Constructors are a special member function that has the same name as the class. In the body of the constructor method, we can initialize values to default, or accept them from an input. The syntax for a constructor is the same as any other method, but the name has to be the same name as the class.

**Copy Constructor** A copy constructor is one that receives a reference to its own class as a parameter, such as `class(class &c)` as a constructor. You CANNOT pass in the value of the class, it has to be by reference. You can then access the passed in reference and copy the values as you see fit, as if it was any other method.

**Destructors** A destructor is a special member function whose name is the same as the class with a `~` appended behind it. `~ class()`. Destructors are called in two instances; when a temporary object passes out of its scope, and when we use the "delete" operator (next section). It is the responsibility of the programmer to deal with what happens in the destructor.

**New and Delete** C++ introduces two operators we can use for managing memory; **new** and **delete**. These are used to manage dynamically allocated memory; new is for declaring memory blocks and delete is for freeing memory blocks (similar to malloc and free). Using new allows us to initialize and declare an object at the same time, while delete explicitly calls the destructor and does whatever you need it to do.

## 6.2.2 Inheritance

**Operator Overloading** C++ aims to make user defined classes almost the same as built in types, like int. One of things we can do with built in types is manipulate them with operators, like `+` `-` `*` `/`. User defined classes don't have these operators by default, so we have to define them ourselves. We do this using a special method referred to as an operator function. For example, if we wanted to define `+` for our class, we would declare our function's return type as its own class, and the function's signature would look like **operator +(class c)**. This is results in the following to mean the same thing/

- `c_sum = c1 + c2;`
- `c_sum = c1.operator+(c2);`

The left hand side calls the function with the right hand side as the argument. You can also have operators without arguments, such as negation `(-c1)`.

**Friend Function** What if we wanted to add a float with our own class? This is easy if we use the float as the argument as **operator +(float f)**. But what if the float is the one calling its own function? We can't just redefine the operators in the float data type, so we need to use a different method, called friend functions. Friend functions are defined outside of the class scope, but it has access to all private and protected members of a class. Our **operator +(float f)** can now be defined as **friend operator +(class c, float f)** and we can now add in when our argument is float on the left as **friend operator +(float f, class c)** Note: the overloaded operator function must have at least one operand of the class it is in.

**Inheritance** Classes in C++ can inherit from one another, much like in java. There are a few types of inheritance to cover in C++.

- Single Inheritance: A derived class inherits from a single base class
- Multiple Inheritance: A derived class inherits from more than one base class
- Multilevel Inheritance: A derived class inherits from another derived class; like a chain.
- Hybrid Inheritance: A wombo combo of other inheritances

**Visibility** Access mode, as seen in Figurefig:inheritance, can either be private or public, with the default being private. If it is public, then the public members of the base class are inherited as public members, otherwise the inherited members become private. When doing multiple

```

class derived_class : <access_mode> base_class
{
    ... // own members
}

```

Figure 6.6: Syntax for Single Inheritance

inheritance, the syntax is the same as for one base class, but we separate the multiple base classes with commas. Note: private members of base classes are not inherited!

Now what if we wanted to inherit something, but we didn't want it to be public in the base class? There is a third visibility identifier we call "protected." Protected members are public for its children, but are private for all other areas. Access mode can also be protected in inheritance declaration.

**Virtual** Sometimes, we can accidentally inherit functions with the same signatures from different classes. What we can do is declare functions as virtual. Virtual functions are ones that are expected to be overridden in their child classes (similar to abstract in Java, but they still function as normal in the parent class) You can also declare parent classes to be virtual, this helps to stop multiple copies of the same grandparent to be created.

Declaring a function as virtual also allows for polymorphism, similar to Java. If we have a reference for the base class, by default, C++ is not polymorphic and won't connect it with a child class that inherits it. However, virtual functions allow for polymorphism. This is because polymorphic functions need the use of a hidden "vtable" which uses extra space in memory, and C++ is all about that #Efficiency, so the power of polymorphism is in the hands of the programmer.

### 6.2.3 Memory Management

**C vs C++** C provides **malloc()**, **calloc()**, **realloc()** for dynamic memory allocation, and **free()** for dynamic memory deallocation. C++ provides **new** for dynamic memory allocation, and **delete** for dynamic memory deallocation. C++ provides some additional bonuses too, it provides the manual memory management as we have seen so far, but it can also provide the convenience of garbage collection when desired. We get garbage collection C++ using smart pointers.

**Smart Pointers** Smart pointers are pointers whose heap memory is managed by the stack-/garbage collector. They are custom classes found in the `<memory>` header file. There are two types of smart pointers we covered.

1. Unique Pointer: Stylized as **unique\_ptr**, the raw pointer it connects to cannot be copied to another unique pointer, and it cannot be passed to a function by value. It can only be moved if ownership of the memory resource is transferred to another unique pointer or after the original unique pointer no longer owns it.
2. Shared Pointer: Stylized as **shared\_ptr**, the raw pointer it connects to can be copied to another shared pointer, and it can be passed to a function by value. All instances point to the same object and share access to one "control block" for reference counting.

When declaring the smart pointer without initializing, it needs to be initialized with a "**nullptr**", this is due to how it is set up.

## 6.3 Generic Programming

### 6.3.1 Pointers

**Void Pointers** Since C doesn't have classes, we will use what was the next best thing: void pointers. Void pointers are pointers declared with type "void", which is no type at all. Therefore, we can't operate on void pointers or dereference them without typecasting them. Not typecasting them would not scale operations properly or return some garbage values.

**Function Pointers** Since functions have an address in memory they start in, we can have function pointers that point to that address! The syntax to declare function pointers is as such: **return\_type (\*name) (parameterType1, parameterType2, ...)** for example, a function that returns an int and takes one int as a parameter would have its pointer declared as **int (\*foo) (int)**. To call the function from a pointer, we just dereference the pointer and throw in our parameters in parenthesis after the dereferencing.

### 6.3.2 Templates

**Function Template** A template in C++ is a generic object. A function template can have parameters that are an undefined type, defined later by the programmer in some areas or by the compiler. This is preferred over void pointers and function pointers because it has less source code, is more flexible and easier to implement. If we were to represent the template type as **T**, before our function declaration we need to add **template<typename T>**, representing that the typename "T" is a template type. Ofcourse, this works for functions, but if we wanted to have a generic class, say like a data structure that can be of any type?

**Class Template** We can declare members of a class to have the generic type "T" if we add before the class declaration **template <typename T>**. If we wanted to return generic objects, such as a pointer of type T, **\*T**, then we need to add **typename** before the return declaration of a function. We do this to make sure the compiler knows what it is we want, and that is a generic return.

# Chapter 7

## Mathematical Techniques for Computer Science

### 7.1 Vectors and Matrices

This section deals with operations on vectors and matrices that are commonly used.

#### 7.1.1 Vector Operations

**Magnitude**

$$\|\vec{V}\| = \sqrt{\sum_{i=1}^n v_i^2} \quad (7.1)$$

**Dot Product**

$$\vec{V} \cdot \vec{U} = \sum_{i=1}^n v_i u_i \quad (7.2)$$

**3x3 Cross Product**

$$\vec{V} \times \vec{U} = \begin{bmatrix} v_2 u_3 - v_3 u_2 \\ v_3 u_1 - v_1 u_3 \\ v_1 u_2 - v_2 u_1 \end{bmatrix} \quad (7.3)$$

**Equations with angles**

$$\cos \theta = \frac{\vec{V} \cdot \vec{U}}{\|\vec{V}\| \|\vec{U}\|} \quad (7.4)$$

$$\sin \theta = \frac{\|\vec{V} \times \vec{U}\|}{\|\vec{V}\| \|\vec{U}\|} \quad (7.5)$$

#### 7.1.2 Matrix Operations

**Multiply**

$$a \times b = \begin{bmatrix} (a_{1,1} * b_{1,1} + a_{1,2} * b_{2,1} + \dots) & (a_{1,1} * b_{1,2} + a_{1,2} * b_{2,2} + \dots) \\ (a_{2,1} * b_{1,1} + a_{2,2} * b_{2,1} + \dots) & (a_{2,1} * b_{1,2} + a_{2,2} * b_{2,2} + \dots) \\ \dots & \dots \end{bmatrix} \quad (7.6)$$

**2x2 Determinant**

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc \quad (7.7)$$

### 3x3 Determinant

$$\det\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \times \det\begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \times \det\begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \times \det\begin{pmatrix} d & e \\ g & h \end{pmatrix} \quad (7.8)$$

## 7.2 Fields

This section deals with the little we need to know about Galois Fields, primarily GF(2).

### 7.2.1 Notes

**Definition** Fields can be thought of as a "number system" in mathematics that has the ability to do, at least, be added and multiplied. The number system can be defined however we like, but it needs to follow a set of rules.

- There must be a definition for "0" and "1" such that  $x + 0 = x$  and  $x * 1 = x$
- Addition and Multiplication are associative and commutative
- Multiplication distributes over addition
- There must be valid solutions/proofs for  $a + x = 0$  and  $a * x = 1$

**GF(2)** One particular field we use a lot is GF(2). the binary field. GF(2) only has the values "0" and "1" in its field. Multiplying in this field is the same in the field of integers we are used to, but there is one change in addition.  $1 + 1$  evaluates to 0 in GF(2).

**Proving Laws** Prove that  $a + x = 0$  and  $a * x = 1$  only have **one** solution.

$$\begin{aligned} x_1 &= x_1 + 0 \\ &= x_1 + (a + x_2) \\ &= (x_1 + a) + x_2 \\ &= (0) + x_2 \\ &= x_2 \end{aligned} \quad (7.9)$$

$$\begin{aligned} x_1 &= x_1^* 1 \\ &= x_1^* (a^* x_2) \\ &= (a^* x_1)^* x_2 \\ &= (1)^* x_2 \\ &= x_2 \end{aligned} \quad (7.10)$$

## 7.3 Lines and Planes

This section covers everything done regarding systems of equations regarding lines and planes.

### 7.3.1 Gaussian Elimination

**Algorithm** Gaussian Elimination can be described as a recursive algorithm used to solve a system of linear equations. It's a process we can use that is fairly robust. Gaussian elimination

converts the system into matrix form, and we then perform row operations on it. Take for example this system.

$$\begin{aligned}x_1 + 5x_2 - 2x_3 &= -11 \\3x_1 - 2x_2 + 7x_3 &= 5 \\-2x_1 - x_2 - x_3 &= 0\end{aligned}\tag{7.11}$$

If we were to translate this to matrix form, it would look like this:

$$\left[ \begin{array}{ccc|c} 1 & 5 & -2 & -11 \\ 3 & -2 & 7 & 5 \\ -2 & -1 & -1 & 0 \end{array} \right]$$

**Row Operations** In matrix form, we can now commit to row operations. There are three possible row operations that we can use and mix and match.

- Swap rows: Swap the location of two rows. (eg.  $R1 \leftrightarrow R3$ )
- Multiply a row with a constant: You can multiply each term in a row with a constant
- Add two rows: You can add two rows and replace one of the addends with the sum.

Our goal with row operations to reach row echelon form; such as this form:

$$\left[ \begin{array}{ccc|c} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{array} \right]$$

This is interpreted as  $x_1 = a$ ,  $x_2 = b$ , and so forth.

**Special Cases** There are two special cases

1. Contradictory equations. If an equation results in a contradiction (eg.  $0 + 0 = 1$ ) then we can determine that there is no solution to our system.
2. Irrelevant equation. If one the equations becomes entirely made of 0, then that equation is irrelevant and we have one less equation to work with. This usually results in one unknown that is then declared to be chosen freely. This can also happen when we have two equations for one unknown.

**Matrix Inverse** The gaussian can also be interpreted in the form

$$\mathbf{A}\mathbf{x} = \mathbf{C}$$

Where  $\mathbf{A}$  is the  $N \times N$  constants applied to the unknowns,  $\mathbf{x}$  is the  $N \times 1$  unknowns, and  $\mathbf{C}$  is the  $N \times 1$  constants in the equation. To solve an equation set up this way, we would normally divide A from both sides, but with matrices division isn't defined. So we need to do accomplish the inverse of multiplication another way. We do this using inverse matrices,  $\mathbf{A}^{-1}$ .

The inverse is defined with the following equation;

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

Where I is the identity matrix. There a few rules regarding inverse matrices that the base matrix has to pass.

- It must be a square matrix.
- The determinant of the matrix cannot be 0.

Once we know our base matrix has an inverse, we can find it using row operations. We do row operations on our base matrix until it reaches row echelon form. While doing so, we apply the same operations to the identity matrix. The result of the operations done on the identity matrix is the inverse matrix.



### 7.3.2 Lines

**Parametric Representation** Lines, we know 'em, we love 'em. Lines are comprised of many points on a grid, and a point is comprised of coordinates. These coordinates are often represented as vectors. We are used to the normal form of a line equation;

$$y = m \times x + b$$

but we have also learned how to represent a line in vector form. Given two points,  $P_1$  and  $P_2$ , we can construct an equation. We need to find the vector between  $P_1$  and  $P_2$ ; the difference (as  $\vec{V}$ ) which can be interpreted as the slope. We can then take one of the points, and add it with our difference vector  $\vec{V}$  multiplied with a scaling factor, and we can, using the scaling factor, generate any point on the line.

$$\mathbf{X} = P_1 + Vt$$

**Intersecting Lines** What if we want to find the point two lines intersect? This is really easy! What we do is we set the lines equal to each other  $X = Y$ , and this generates a system of linear equations of sorts (where each row in the vector is an equation). Move the numbers around, and we can then use Gaussian elimination to solve for the value scaling factors. To get the point, simply plug in the appropriate scaling factor back to its parent parametric equation and the result is your point of interest.

**Normal Form** To convert a line back to its normal form, we need to find the (unsurprisingly) normal. The normal is a  $2 \times 1$  vector for lines, and can be found with the following methods. Given  $X = P + Vt$ ,  $P = \begin{bmatrix} a \\ b \end{bmatrix}$  and  $V = \begin{bmatrix} c \\ d \end{bmatrix}$ , then the normal  $N = \begin{bmatrix} n_1 = d \\ n_2 = -c \end{bmatrix}$  from here on, you plug in the values into the equation

$$n_1x_1 + n_2x_2 = d$$

all that's left is to solve for "d". in this case,  $d = ad - cd$ . (The d is the determinant!).

### 7.3.3 Planes

**Parametric Representation** Lines are two dimensional. What if we wanted three dimensional parametric equations? These 3D sheets are called planes, and are mathematically similar to lines. Instead of being represented by two points, they are represented by three points. Given three points,  $P$ ,  $Q$  and  $R$ , and two scaling factors  $s$  and  $t$ , the equation for a plane is represented by

$$X = P + s \cdot \overrightarrow{PQ} + t \cdot \overrightarrow{PR}$$

It's similar to that of a 2D line, just with an extra scaling factor. Solving for intersecting planes works the same way as for lines, set them equal to each other and do Gaussian Elimination.

**Normal Form** Converting a plane to normal form is also similar to that of a line. Given the scaleable vectors  $\vec{v}$  and  $\vec{w}$ , we can find the normal  $\vec{n}$  of the plane by taking the cross product,  $\vec{v} \times \vec{w}$ . Given  $X = P + Vt + Ws$ , we find the normal as

$$n_1x_1 + n_2x_2 = N \cdot P$$

**Mirroring a Point** Sometimes, we want to mirror points on one side of a plane to the opposite side. Thinking of this theoretically, we want to find the smallest distance vector between our point  $Q$  and our plane  $X$ , and move the point that distance on the opposite side. We need to first find the smallest distance from the plane to point. This can be done with the following formula.

$$\mathbf{dist}_Q = \frac{d - (\vec{n} \cdot Q)}{\|\vec{n}\|} \cdot n$$

Where

- $d$  is the constant in the normal form of the plane
- $\vec{n}$  is the normal of the plane
- $\vec{Q}$  is the point of interest

This will spit out a vector. Now, we can use this distance to find the  $Q'$  point on the plane closest to the point of interest, and from there find the  $Q''$  point that is the reflection of our point of interest.

$$\begin{aligned}\vec{Q}' &= \vec{Q} + \mathbf{dist}_Q \\ \vec{Q}'' &= \vec{Q} + 2 \times \mathbf{dist}_Q \\ &= \vec{Q}' + \mathbf{dist}_Q\end{aligned}\tag{7.12}$$

We can also reflect lines fairly simply. All we need to do is find the reflection of two points on the line and generate a new line from the new points. Simple and nice.

### 7.3.4 Basis and Orthogonality

**Orthogonality** Vectors can have a property referred to as orthogonality. Vectors that are orthogonal are vectors that are at right angles to each other. For example, the vectors represent the x-axis and y-axis in Euclidean space are orthogonal. To test orthogonality of vectors, we just take the dot product  $\vec{v} \cdot \vec{u}$  of the two vectors. If the dot product is "0", then the vectors are orthogonal. Furthermore, we can test orthogonality of lines and planes. We do this by obtaining their normals, and doing orthogonality checks on the normal vectors.

**Vector Space** Vector space can be thought of as the number of dimensions over a certain scalar field. For example,  $GF(2)^3$  is the 3-dimensional vector space over  $GF(2)$ . Movements across n-dimensions are movements in the vector space. There is also the notion of vector sub-spaces. These are spaces of the form  $s \cdot \vec{v}$ , which in our work so far, are lines in 2D and planes in 3D. The equation for a line/plane is the subspace of 2D/3D vector space.

**Linear Independence** We know how to generate subspace as lines and planes, referred to as generators. If we have generators that all point to different points in our subspace, then that's all good. But what happens if we have two generators that are scalar multiples of each other? Then we have redundant, or linearly dependent, generators. We want to be remove all redundant generators and bring our world back down to linearly independent generators. A set of linearly independent generators is referred to as a **basis**, and the number of elements in each generator is the **dimension** of the subspace.

We can check linear independence using, you guessed it, Gaussian Elimination. We place all our vectors together and reduce them all to row-echelon form. Any redundant vectors will end up with all 0's in its respective row. If we have a row-echelon matrix with no special cases, then we have linear independence.

## 7.4 Set Theory

This section covers the basics of Set Theory, with some examples.

### 7.4.1 Notation

**Set** A set is a collection of things. These things are called elements or members, and they follow the following ideas:

1. Each member must be identifiable and distinguishable (aka unique)
2. There is clear criterion defined as to whether an element is part of a set
3. A member of a set is only counted once
4. There is no order in a set
5. Sets are only defined by their members. If two different sets have the same elements, as far as math is concerned, they are the same set.

**Set Notation** To indicate if a member  $x$  is a part of a set  $A$ , we use the following notation.

$$x \in A$$

Small, finite sets can be defined by listing out all its members;  $\{Summer, Winter, Spring, Fall\}$ , while infinite sets usually have a clear pattern;  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ . We can also define subsets. A subset is a set whose elements are selected from another set. If we see  $B$  as a subset of  $A$ , we note it as  $B \subseteq A$ . If we want to define  $B$  with a condition, we do it as such:

$$B = \{x \in A | x \text{ satisfies condition } c\}$$

We can use formal logic, such as "there exists  $\exists$ " to help us define some conditions, as they can get pretty wild. Note: We can also have an empty set; an empty set is noted as  $\emptyset$ .

**Set Operations** We can also do operations between sets. These are fairly simple to understand. Assume we have sets  $A \subseteq X$  and  $B \subseteq X$ .

1. Union -  $A \cup B$ : Returns the set whose members are in  $A$  OR  $B$
2. Intersection -  $A \cap B$ : Returns the set whose members are in both  $A$  AND  $B$
3. Complement -  $\bar{A}$ : Returns the set whose members are NOT in  $A$ .
4. Difference -  $A \setminus B$ : Returns the set whose members are IN  $A$  and NOT IN  $B$

You can mix and match operations and create new ones, like the difference operation. It is equivalent to the operations  $B \cap \bar{B}$ .

### 7.4.2 Cardinality

**Cardinality** The cardinality of a set can be defined as the size, or amount of members in the set.

**Countable Sets**

**Proving Countable Sets**

**Uncountable Set**

### 7.4.3 Relations

**Relation**

**Reflexivity**

**Symmetry**

**Transitivity**

**Types of Relations**   order / equivalence

#### **7.4.4   Examples**

### **7.5   Functions**

Expanding from Set Theory, this section deals with Functions and our new understanding of them.

#### **7.5.1   Notes**

#### **7.5.2   Examples**

### **7.6   Probability**

Continuing from Year 1 AI, probability.

#### **7.6.1   Notes**

#### **7.6.2   Examples**

### **7.7   Random Variables**

This section deals with the handling of both discrete and continuous random variables. Don't go gambling after you cover this section.

#### **7.7.1   Discrete**

#### **7.7.2   Continuous**

#### **7.7.3   Examples**

### **7.8   Equations**

Appending-like area to store all equations that will likely be used in the exam.

# Chapter 8

## Introduction to Computer Security

### 8.1 Cryptography

different modes of AES RSA and Diffie Helman MACs and Validation attack example

### 8.2 Access Rights

reading access rights privilege elevation

### 8.3 Web Security

#### 8.3.1 Communication Protocols

#### 8.3.2 Web Security

php, sql injection, file injection, SSH

### 8.4 Assembly

the stack reverse engineering overflow injection code injection

# Chapter 9

## Professional Computing

9.1 Computer Misuse

9.2 GDPR

9.3 Liability

9.4 Intellectual Property

9.5 Human Resources

9.6 Internet and ISPs

9.7 Ethics

# Chapter 10

## Functional Programming

### 10.1 Functional Programming

define key features pure vs not so pure examples

### 10.2 Algebraic Data Types

algebra and logic isomorphism

### 10.3 Imperitive OCaml

I/O and machine efficiency assignment and dereferencing records exceptions example

### 10.4 Modules and Functors

modules mathematical functor ocaml functor type abstraction ocaml abstraction lifehack fix

### 10.5 Monads

monad lift and bind the option monad mult, unit, identity state monad monads in ocaml

### 10.6 GADTs

overview example

### 10.7 Lazy Programming

evaluation order streams example