Legal Advice Project

📘 Detailed Notes on the PDF → Embedding → FAISS Indexing Script

1. 🔎 Purpose of the Script

- Automates the process of converting **Supreme Court judgments (PDFs)** into a searchable **vector database**.
- Enables **semantic search** (find similar content by meaning, not keywords).
- Outputs:
  - `supreme_court.faiss` → vector index
  - `metadata.json` → mapping of vectors → original PDF text

2. 📂 Workflow / Procedure

Step 1: **Collect PDFs**

- Place all judgment PDFs inside `supreme_court_judgments/`.
- Expected structure (for year extraction):

```
supreme_court_judgments/
├── 2020/case1.pdf
├── 2021/case2.pdf
└── ...
```

Step 2: **Extract Text**

- Uses **PyPDF2** (`PdfReader`) to read text page by page.
- If page text is missing (`None`), adds empty string.

- ⚠️ Limitation: For scanned image PDFs → text = `""`. Needs **OCR** separately.

Step 3: **Chunk Text**

- Splits text into **500-word segments**.
- Overlap = 50 words (to maintain context between chunks).
- Example:

```
Chunk 1 → words 0–499
Chunk 2 → words 450–949
...
```

Step 4: **Encode Chunks**

- Uses **SentenceTransformer** (`all-MiniLM-L6-v2`) model → 384-d embeddings.
- Encoding done in **batches of 128 chunks** for efficiency.
- Embeddings are **normalized** → allows cosine similarity with FAISS.

Step 5: **Store in FAISS Index**

- FAISS index type: `IndexFlatIP`
  - Uses **inner product similarity**.
  - Since vectors are normalized, **cosine similarity** = inner product.
- Metadata for each chunk stored in `metadata.json`.
- Example metadata entry:

```
{
  "file": "supreme_court_judgments/2020/case1.pdf",
  "year": "2020",
  "chunk_index": 3,
  "text": "...."
}
```

Step 6: **Save & Resume**

- Every **500 PDFs**, script saves:
  - `supreme_court.faiss`
  - `metadata.json`
  - `completed_files.txt`
- If rerun, skips already processed PDFs → continues where left off.

---

3. ⚙️ Key Functions

Function	Role
`extract_text_from_pdf(file_path)`	Reads PDF text with PyPDF2
`chunk_text(text, size, overlap)`	Splits text into overlapping word chunks
`save_progress(index, metadata, completed_files)`	Writes index, metadata, and progress files
`main()`	Orchestrates whole process: PDF scanning, extraction, embedding, indexing

---

4. 📦 Required Installations

Core Dependencies

```
pip install sentence-transformers
pip install PyPDF2
pip install tqdm
pip install numpy
```

PyTorch (required for embeddings)

- CPU-only:

```
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
```

- GPU (if Nvidia GPU present, match CUDA version from PyTorch site):

```
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
```

FAISS (Vector Database)

- CPU version (easier, recommended):

```
pip install faiss-cpu
```

- GPU version (may fail on Windows):

```
pip install faiss-gpu
```

---

5. 💻 How to Run

1. Put PDFs in `supreme_court_judgments/`.
2. Install dependencies (above).
3. Run script:

   ```
   python chat.py
   ```

4. Wait for progress bar → Index is built.
5. Outputs:
   - `supreme_court.faiss`
   - `metadata.json`
   - `completed_files.txt`

---

6. 🔍 How to Query the Index

```
import faiss, json, numpy as np
from sentence_transformers import SentenceTransformer

# Load model, index, metadata
model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
index = faiss.read_index("supreme_court.faiss")
with open("metadata.json", "r", encoding="utf-8") as f:
    metadata = json.load(f)

# Encode query
query = "Can the police arrest without a warrant?"
q_vec = model.encode(query, convert_to_numpy=True, normalize_embedding
s=True)
q_vec = np.ascontiguousarray(q_vec.reshape(1, -1).astype("float32"))

# Search top 5 results
D, I = index.search(q_vec, 5)
for idx in I[0]:
    print(metadata[idx]["file"], metadata[idx]["chunk_index"])
    print(metadata[idx]["text"][:300], "...\n")
```

```
import os                    # operating system utilities (file checks, env, etc.)
import glob                  # file pattern matching (find PDFs recursively)
import json                  # read/write JSON for metadata
import faiss                 # FAISS library for fast vector (similarity) search
import numpy as np           # numerical arrays and operations
from pathlib import Path     # convenient path handling (Path objects)
from tqdm import tqdm        # progress bars for loops
from concurrent.futures import ProcessPoolExecutor  # parallelize CPU-boun
d tasks
from sentence_transformers import SentenceTransformer  # embedding mode
l wrapper
from PyPDF2 import PdfReader  # read and extract text from PDFs
import torch                 # used to check CUDA availability and for model backe
```

```
nd

# ---------------- CONFIG ----------------
ROOT_DIR = Path("supreme_court_judgments")  # root folder containing PDFs
OUTPUT_INDEX = "supreme_court.faiss"        # filename for saved FAISS inde
x
OUTPUT_METADATA = "metadata.json"           # filename for saved metadata
CHUNK_SIZE = 500                            # number of words per text chunk
CHUNK_OVERLAP = 50                          # overlap (words) between consecutiv
e chunks
BATCH_SIZE = 128                            # how many chunks to encode at once
MODEL_NAME = "sentence-transformers/all-MiniLM-L6-v2"  # embedding m
odel to use
SAVE_INTERVAL = 500                         # save progress after processing this m
any PDFs
# ----------------------------------------

device = "cuda" if torch.cuda.is_available() else "cpu"
# choose 'cuda' if GPU available else 'cpu'

model = SentenceTransformer(MODEL_NAME, device=device)
# load the SentenceTransformer model onto the chosen device

def extract_text_from_pdf(file_path):
    try:
        reader = PdfReader(file_path)                    # open the PDF
        text = "\n".join([p.extract_text() or "" for p in reader.pages])
        # extract text from every page; if extract_text() returns None → use empt
y string
        return file_path, text                           # return tuple (path, text)
    except Exception as e:
        return file_path, ""                             # on error return empty text

def chunk_text(text, chunk_size=500, overlap=50):
    words = text.split()                                 # split text into words (list)
    chunks = []                                          # list to hold chunks
```

```
    start = 0
    while start < len(words):                          # loop until all words consumed
        end = min(start + chunk_size, len(words))      # compute chunk end in
dex
        chunk = " ".join(words[start:end])             # join words back to a string
        chunks.append(chunk)                           # add chunk to list
        start += chunk_size - overlap                  # advance start (with overlap)
    return chunks                                      # return list of chunks

def save_progress(index, metadata, completed_files):
    faiss.write_index(index, OUTPUT_INDEX)             # write FAISS index to d
isk
    with open(OUTPUT_METADATA, "w", encoding="utf-8") as f:
        json.dump(metadata, f, ensure_ascii=False, indent=2) # save metadata lis
t to JSON
    with open("completed_files.txt", "w") as f:
        f.write("\n".join(completed_files))            # save processed file paths
    print(f"✅ Progress saved: {len(completed_files)} PDFs processed.")
    # print confirmation showing how many PDFs marked completed

def main():
    pdf_files = glob.glob(str(ROOT_DIR / "**/*.pdf"), recursive=True)
    # find all .pdf files under ROOT_DIR recursively and return a list of file paths
(strings)

    print(f"Found {len(pdf_files)} PDF files.")        # show how many were fo
und

    completed_files = set()                            # set to track already-processed
files
    if os.path.exists("completed_files.txt"):          # if checkpoint exists
        with open("completed_files.txt", "r") as f:
            completed_files = set(f.read().splitlines())   # read completed file paths
into set
        pdf_files = [f for f in pdf_files if f not in completed_files]
        # filter out files already processed (resume behavior)
```

```
        print(f"Resuming: {len(pdf_files)} files remaining.")# report how many re
main

    dim = model.get_sentence_embedding_dimension()         # embedding dim
ension (e.g., 384)
    if os.path.exists(OUTPUT_INDEX):
        index = faiss.read_index(OUTPUT_INDEX)             # load existing FAISS i
ndex from disk
        with open(OUTPUT_METADATA, "r", encoding="utf-8") as f:
            metadata = json.load(f)                        # load existing metadata list
    else:
        index = faiss.IndexFlatIP(dim)                     # create new FAISS index (inne
r-product)
        metadata = []                                  # start empty metadata list

    with ProcessPoolExecutor(max_workers=os.cpu_count() // 2) as executor:
        results = list(tqdm(executor.map(extract_text_from_pdf, pdf_files),
                            total=len(pdf_files), desc="Extracting PDFs"))
    # parallelize PDF text extraction across CPU cores (half of available cores)
    # executor.map runs extract_text_from_pdf(file) for each file and returns tup
les (file, text)
    # tqdm wraps the iterator to show a progress bar; results is a list of (file_pat
h, text)

    batch_embeddings = []    # list to accumulate numpy arrays of embeddings
(batches)
    batch_metadata = []      # list to accumulate metadata entries (parallel to em
beddings)
    processed_count = 0      # counter for how many PDF files processed in this
run

    for file_path, text in tqdm(results, desc="Chunking + Embedding"):
        if not text.strip():                           # if extracted text is empty/whitespace
            continue                                   # skip this PDF (nothing to embed)

        chunks = chunk_text(text, CHUNK_SIZE, CHUNK_OVERLAP)
```

```
        if not chunks:                          # if chunker returned no chunks
            continue                            # skip to next PDF

        for i in range(0, len(chunks), BATCH_SIZE):   # process chunks in batches
of BATCH_SIZE
            batch = chunks[i:i + BATCH_SIZE]         # slice one batch (list of text ch
unks)
            vectors = model.encode(batch, convert_to_numpy=True, normalize_em
beddings=True)
            # encode text batch → numpy array of vectors; normalized (unit lengt
h) for cosine search
            batch_embeddings.append(vectors)         # store the batch array in bat
ch_embeddings

            for j, chunk in enumerate(batch):        # iterate over chunks in the batc
h
                batch_metadata.append({
                    "file": str(file_path),          # original file path (string)
                    "year": Path(file_path).parts[1], # grabs the 2nd path component a
s 'year'
                    "chunk_index": i + j,            # chunk index relative to this file
                    "text": chunk                    # the chunk's raw text
                })
                # NOTE: Path(file_path).parts[1] assumes the path has at least two p
arts;
                # if structure differs this may raise IndexError or give wrong value.

        completed_files.add(file_path)               # mark file as processed (for res
ume)
        processed_count += 1                         # increment processed-count

        if processed_count % SAVE_INTERVAL == 0:        # every SAVE_INTERV
AL PDFs, flush+save
            if batch_embeddings:
                matrix = np.vstack(batch_embeddings).astype("float32")
                # vertically stack all stored batch arrays into one 2D matrix, cast to fl
```

```
oat32
            index.add(matrix)                    # add vectors to FAISS index
            metadata.extend(batch_metadata)        # append metadata entries i
n same order
            batch_embeddings, batch_metadata = [], []# clear temporary accum
ulators
        save_progress(index, metadata, completed_files)
        # persist index, metadata, and completed_files to disk (checkpoint)

    if batch_embeddings:                           # after loop, if leftovers exist
        matrix = np.vstack(batch_embeddings).astype("float32")
        index.add(matrix)                        # add remaining vectors to index
        metadata.extend(batch_metadata)              # append remaining metada
ta

    save_progress(index, metadata, completed_files)      # final save to persist e
verything
    print("🎉 All PDFs processed and indexed successfully!")  # completion me
ssage

if __name__ == "__main__":
    main()                                    # run main() only when script executed di
rectly
```

📘 Detailed Notes on the Script

```
import faiss
import json
import numpy as np
from sentence_transformers import SentenceTransformer
from phi.agent import Agent
from phi.model.google import Gemini
```

```
# ---------------- CONFIG ----------------
FAISS_INDEX = "supreme_court.faiss"
METADATA_FILE = "metadata.json"
EMBED_MODEL = "sentence-transformers/all-MiniLM-L6-v2"
TOP_K = 5
SHOW_PREVIEW_CHARS = 500  # characters to preview from each chunk
# ----------------------------------------

print("📂 Loading FAISS index & metadata...")
try:
    index = faiss.read_index(FAISS_INDEX)
except Exception as e:
    raise RuntimeError(f"❌ Could not load FAISS index: {FAISS_INDEX}\n{e}")

try:
    with open(METADATA_FILE, "r", encoding="utf-8") as f:
        metadata = json.load(f)
except FileNotFoundError:
    raise RuntimeError(f"❌ Metadata file not found: {METADATA_FILE}")

print(f"✅ Loaded FAISS index with {index.ntotal} vectors and metadata for {le
n(metadata)} chunks.")

# Load embedding model
print("⚡ Loading embedding model...")
embed_model = SentenceTransformer(EMBED_MODEL)

# Initialize PhiData agent
agent = Agent(
    model=Gemini(id="gemini-1.5-flash"),
    markdown=True,
)

def expand_query(original_query: str) -> str:
    """Use LLM to rewrite/expand user query strictly for Indian Supreme Court
context."""
```

```
    expansion_prompt = f"""
    Expand the following query strictly in the context of the Constitution of Indi
a
    and Supreme Court of India case law. Do NOT include foreign cases or US a
mendments.

    Return only the expanded query as a single line of text.

    Query: {original_query}
    """
    run = agent.run(expansion_prompt)
    return run.content.strip()

def retrieve_chunks(query: str, k: int = 5):
    """Embed query, search FAISS, return top-k metadata chunks."""
    query_vector = embed_model.encode([query], normalize_embeddings=Tru
e)
    D, I = index.search(np.array(query_vector, dtype="float32"), k)
    retrieved = [metadata[i] for i in I[0]]
    return retrieved

def build_context(original_query: str, retrieved_chunks):
    """Combine original query + top chunks into a context string for LLM."""
    context = "You are an expert legal assistant.\n"
    context += "Answer the query STRICTLY using ONLY the following excerpts
from Supreme Court of India judgments.\n"
    context += "If no relevant information is found, say exactly: 'No relevant inf
ormation found in the provided judgments.'\n"

    for i, chunk in enumerate(retrieved_chunks, 1):
        context += f"\n[Case {i} | File: {chunk['file']} | Year: {chunk['year']} | Chu
nk: {chunk['chunk_index']}]\n"
        context += chunk["text"] + "\n"

    context += f"\nUser Query: {original_query}\n"
    context += "Provide a concise, legally accurate answer, and cite case numb
```

```
ers [Case 1], [Case 2], etc. where applicable."
    return context

def run_query_pipeline(user_query: str):
    print(f"\n🔎 Original Query: {user_query}")
    expanded_query = expand_query(user_query)
    print(f"📖 Expanded Query: {expanded_query}\n")

    retrieved = retrieve_chunks(expanded_query, TOP_K)
    if not retrieved:
        print("❌ No relevant chunks found in FAISS index.")
        return

    print(f"✅ Retrieved {len(retrieved)} chunks:\n")
    for i, chunk in enumerate(retrieved, 1):
        preview = chunk["text"][:SHOW_PREVIEW_CHARS].replace("\n", " ")
        print(f"--- Chunk {i} ---")
        print(f"File: {chunk['file']} | Year: {chunk['year']} | Chunk Index: {chunk['c
hunk_index']}")
        print(f"Preview: {preview}...\n")

    context = build_context(user_query, retrieved)
    print("🧠 Sending context to LLM...\n")
    response = agent.run(context)
    print("🎯 Final Answer:\n")
    print(response.content)

# ---------------- USAGE ----------------
if __name__ == "__main__":
    user_query = "An industrial plant is polluting a nearby river affecting local c
ommunities. Can I file a Public Interest Litigation (PIL) against the company?
Which Supreme Court judgments discuss environmental protection, citizens' ri
ghts, and PIL procedure?"
    run_query_pipeline(user_query)
```

1. Imports & Config

```
import faiss
import json
import numpy as np
from sentence_transformers import SentenceTransformer
from phi.agent import Agent
from phi.model.google import Gemini
```

- `faiss` → for similarity search in vector database.
- `json` → load metadata describing chunks.
- `numpy` → handle numerical vectors for embedding & search.
- `SentenceTransformer` → embedding model (turns text into vector).
- `Agent` & `Gemini` → PhiData agent wrapping Google Gemini LLM for query expansion + final answer.

```
FAISS_INDEX = "supreme_court.faiss"
METADATA_FILE = "metadata.json"
EMBED_MODEL = "sentence-transformers/all-MiniLM-L6-v2"
TOP_K = 5
SHOW_PREVIEW_CHARS = 500
```

- **Paths & constants**
  - `.faiss` file → stores compressed vectors of legal case chunks.
  - `.json` metadata → maps each vector to original text + file info.
  - Embedding model → pre-trained transformer for semantic similarity.
  - `TOP_K` → number of most relevant chunks to retrieve.
  - `SHOW_PREVIEW_CHARS` → how much of each chunk to preview in logs.

---

2. Loading Index & Metadata

```
print("📂 Loading FAISS index & metadata...")
try:
    index = faiss.read_index(FAISS_INDEX)
except Exception as e:
    raise RuntimeError(f"❌ Could not load FAISS index: {FAISS_INDEX}\n{e}")
```

- Reads the FAISS index from file.
- If corrupted or missing → raises error.

```
try:
    with open(METADATA_FILE, "r", encoding="utf-8") as f:
        metadata = json.load(f)
except FileNotFoundError:
    raise RuntimeError(f"❌ Metadata file not found: {METADATA_FILE}")
```

- Loads metadata JSON that contains:
  - `file` (filename of judgment)
  - `year` (case year)
  - `chunk_index` (position in doc)
  - `text` (actual excerpt)

```
print(f"✅ Loaded FAISS index with {index.ntotal} vectors and metadata for {le
n(metadata)} chunks.")
```

- Confirms successful load and prints counts.

---

3. **Load Embedding Model & Agent**

```
print("⚡ Loading embedding model...")
embed_model = SentenceTransformer(EMBED_MODEL)
```

- Loads the embedding model into memory for converting queries → vectors.

```
agent = Agent(
    model=Gemini(id="gemini-1.5-flash"),
    markdown=True,
)
```

- Creates a PhiData agent with **Gemini-1.5-flash**.
- Will be used for **query expansion** and **final response generation**.

---

4. Functions

(a) Expand Query

```
def expand_query(original_query: str) → str:
    expansion_prompt = f"""
    Expand the following query strictly in the context of the Constitution of Indi
a
    and Supreme Court of India case law. Do NOT include foreign cases or US a
mendments.

    Return only the expanded query as a single line of text.

    Query: {original_query}
    """
    run = agent.run(expansion_prompt)
    return run.content.strip()
```

- Uses Gemini to **rewrite query** →
  - More formal
  - Legally precise
  - Limited strictly to Indian SC law
- Ensures context is clean and avoids irrelevant sources.

(b) **Retrieve Chunks**

```
def retrieve_chunks(query: str, k: int = 5):
    query_vector = embed_model.encode([query], normalize_embeddings=Tru
e)
    D, I = index.search(np.array(query_vector, dtype="float32"), k)
    retrieved = [metadata[i] for i in I[0]]
    return retrieved
```

- Converts query → vector using embeddings.
- Searches FAISS for **k nearest vectors**.
- Retrieves corresponding metadata entries.

(c) **Build Context**

```
def build_context(original_query: str, retrieved_chunks):
    context = "You are an expert legal assistant.\n"
    context += "Answer the query STRICTLY using ONLY the following excerpts
from Supreme Court of India judgments.\n"
    context += "If no relevant information is found, say exactly: 'No relevant inf
ormation found in the provided judgments.'\n"
```

- Sets rules → restrict model to Indian SC judgments only.

```
    for i, chunk in enumerate(retrieved_chunks, 1):
        context += f"\n[Case {i} | File: {chunk['file']} | Year: {chunk['year']} | Chu
nk: {chunk['chunk_index']}]\n"
        context += chunk["text"] + "\n"
```

- Appends retrieved case excerpts with citation labels `[Case 1]`, `[Case 2]` etc.

```
    context += f"\nUser Query: {original_query}\n"
    context += "Provide a concise, legally accurate answer, and cite case numb
```

```
ers [Case 1], [Case 2], etc. where applicable."
    return context
```

- Adds user's original query.
- Instructs LLM to use **case references in response**.

(d) **Pipeline Runner**

```
def run_query_pipeline(user_query: str):
    print(f"\n🔎 Original Query: {user_query}")
    expanded_query = expand_query(user_query)
    print(f"📖 Expanded Query: {expanded_query}\n")
```

- Prints original & expanded query.

```
    retrieved = retrieve_chunks(expanded_query, TOP_K)
    if not retrieved:
        print("❌ No relevant chunks found in FAISS index.")
        return
```

- Gets top-k relevant case chunks.
- If none found → aborts.

```
    print(f"✅ Retrieved {len(retrieved)} chunks:\n")
    for i, chunk in enumerate(retrieved, 1):
        preview = chunk["text"][:SHOW_PREVIEW_CHARS].replace("\n", " ")
        print(f"--- Chunk {i} ---")
        print(f"File: {chunk['file']} | Year: {chunk['year']} | Chunk Index: {chunk['c
hunk_index']}")
        print(f"Preview: {preview}...\n")
```

- Prints retrieved chunks (file, year, preview).

```
    context = build_context(user_query, retrieved)
    print("🧠 Sending context to LLM...\n")
    response = agent.run(context)
    print("🎯 Final Answer:\n")
    print(response.content)
```

- Builds context + retrieved case law.
- Sends to Gemini for final **legal reasoning answer**.
- Prints result.

---

5. **Main Execution**

```
if __name__ == "__main__":
    user_query = "I am a journalist..."
    run_query_pipeline(user_query)
```

- Defines a test query:
  - Journalistic freedom vs Sedition (IPC 124A) & promoting enmity (IPC 153A).
  - Asks if protection under Article 19(1)(a) applies.
- Runs the pipeline:
  1. Expand query
  2. Retrieve SC cases
  3. Build context
  4. Get Gemini answer with case citations.

⚖️ **Overall Flow**

1. **User query** → expanded by Gemini for legal precision.

2. **Embedding model** → converts query → vector.
3. **FAISS search** → finds most relevant case excerpts.
4. **Build context** → joins query + cases into a structured prompt.
5. **Gemini agent** → answers with legal reasoning & citations.

📄 Sample User Queries (Detailed)

1️⃣ Freedom of Speech / Journalism

- **Query:** I am a journalist who published an article criticizing a state government policy. The police have filed an FIR against me under IPC Sections 124A (sedition) and 153A (promoting enmity). I did not incite violence — I only expressed my opinion with supporting facts. Can I seek protection under Article 19(1)(a) of the Constitution, and which Supreme Court judgments can help me defend my right to freedom of speech?

2️⃣ Data Privacy / Digital Rights

- **Query:** The government is collecting personal data including my social media activity and financial records for public welfare schemes. Do I need to provide consent under Indian law? Which Supreme Court rulings clarify the right to privacy and data protection in India?

4️⃣ Criminal Law / Bail

- **Query:** I am accused of a non-violent financial crime and have applied for bail. How does the Supreme Court guide granting bail in such cases? Which judgments provide precedence for bail in economic offense cases?

5️⃣ Environmental Law / Public Interest Litigation

- **Query:** An industrial plant is polluting a nearby river affecting local communities. Can I file a Public Interest Litigation (PIL) against the company? Which Supreme Court judgments discuss environmental protection, citizens' rights, and PIL procedure?

6️⃣ Censorship / Publication Rights

- **Query:** Can the government restrict the publication of a book or academic research under the Constitution? Which Supreme Court judgments define the limits of freedom of expression in publications?

7️⃣ Online Platforms / Liability

- **Query:** Are social media platforms or online portals liable for user-generated content under Indian law? Which Supreme Court judgments clarify platform responsibility and content moderation rules?

8️⃣ Right to Privacy in Digital Communications

- **Query:** How does the Supreme Court define the right to privacy for digital communications, including emails and messages? Which rulings protect citizens against unauthorized surveillance?

9️⃣ Political Activism / Freedom of Expression

- **Query:** I am a political activist expressing dissent. How does the Supreme Court define freedom of speech in protests and political activities? Which judgments safeguard lawful political expression?

🔟 Tenancy / Property Rights

- **Query:** I am a tenant facing unlawful eviction by my landlord. Which Supreme Court judgments define tenants' rights and limits of eviction under Indian tenancy laws?

Police accessed my WhatsApp chats without a proper court warrant during investigation. Can I challenge this under the Right to Privacy guaranteed by the Supreme Court in Puttaswamy judgment, and how does Article 21 protect me in this case?

https://youtu.be/WS1uVMGhlWQ?si=oBubkbhTh3jKRDaX

https://www.youtube.com/playlist?list=PLdF3rLdF4ICScQkCs5SKFO9zijhSpS8EN

https://www.youtube.com/watch?v=eVLiKPcCN2k

https://www.youtube.com/playlist?list=PLAMHV77MSKJ4Z4OXqao1gRdfQK7VQYAXb

https://www.youtube.com/watch?v=aSx0jg9ZILo

https://www.youtube.com/watch?v=acxqoltilME

https://www.youtube.com/watch?v=Po4Tf_UWSu8&t=1s