Introduction

These Finite State Transducers (FSTs) normalize whole numbers 0 - 1000 in English and French. It was built with Pynini.

Methodology

This section explains the approach I used to build a rule-based number normalization system for both English and French using **Finite-State Transducers (FSTs)** with **Pynini**. Since English and French have different morphological rules for number formation, I implemented the two systems separately but followed the same overall workflow:

1. Define the linguistic rules of each language.
2. Convert each rule into a clean, modular FST component.
3. Combine all the components into one final FST normalizer per language.

1. Data Preparation: Defining Core Number Maps

English

For English, number construction is relatively straightforward. I created four major dictionaries:

- **units_map**: 0–9
- **teens_map**: 10–19 (irregular forms)
- **tens_digit_map**: 20, 30, 40, …
- **hundreds_map**: 100, 200, …, 900

These maps form the backbone of the English grammar. Each entry is later converted into an FST that maps the numeric input to the corresponding written-out English word.

French

French has more irregular patterns, especially from **70–99**, so I created:

- **units_map** (0–9)
- **teens_map** (10–19)
- **tens_digit_map** (20, 30, 40, …, but with exceptions)
- **tens_digits_upper_map_odds** for tricky ranges (70–79, 90–99)
- **hundreds_map** with pluralization rules (e.g., *cent* vs *cents*)

This separation allows me to handle French's compound and irregular constructions cleanly.

---

2. Building Basic FST Blocks

For both languages, the smallest meaningful FSTs are **unit FSTs**, which map digits "0"–"9" to their text equivalents.

```
- fst_units = pynini.union(*fst_units_list).optimize()
```

These basic blocks serve as the foundation for all larger number constructions.

I also created small helper FSTs:

Space Inserter

Used to combine words:

```
- fst_insert_space = I_O_FST("", " ")
```

English "and" Inserter

English uses *and* for numbers like:

- one hundred **and** twenty three

French does **not** use this rule, so this helper is only used on the English side.

3. Constructing Larger Number Components

A. English

3.1 Tens (20, 30, 40...)

English tens follow a simple pattern:

- `"2" + "0" → "twenty"`

I built this using:

- the tens map
- an FST that eats the trailing “0”

3.2 Compound Tens (21–29, etc.)

Example:

- `"2" + "1" → "twenty one"`

This required:

- tens FST
- space FST
- unit FST

3.3 Hundreds

Hundreds in English are consistent:

- 100 → "one hundred"
- 300 → "three hundred"

I built:

- direct hundreds (100, 200, …)
- compound hundreds (101–109, 111–119, 121–199, etc.)

These use:

- *hundreds FST*
- *space*
- *and*
- units or tens FSTs depending on the number.

3.4 One Thousand

A simple rule:

- `1000 → "one thousand"`

All English FSTs are combined using `pynini.union()` into the final normalizer.

B. French

French number-building rules are more complicated, especially:

- **70–79** (“soixante-dix” → literally “sixty ten”)
- **90–99** (“quatre-vingt-dix” → literally “four-twenty ten”)

3.1 Simple Tens (20, 30, 40...)

These are straightforward:

```
- 20 → vingt
- 30 → trente
```

3.2 Regular Compound Tens (21–69 except 71, 91, etc.)

Example:

```
- 21 → vingt un
- 59 → cinquante neuf
```

3.3 Upper Compounds (70–79, 90–99)

Here the base is *soixante* or *quatre-vingt*, joined with teens.

Example:

- 71 → soixante onze
- 95 → quatre-vingt quinze

I built special FSTs to generate:

- base tens (soixante / quatre-vingt)
- plus the corresponding teen forms

3.4 Hundreds

French hundreds sometimes add plural “s” (e.g., *deux cents*) unless followed by a number.

I implemented:

- single hundreds (100, 200, …, 900)
- compound hundreds (101–999)

3.5 One Thousand

```
- 1000 → mille
```

Finally, all French FSTs are unioned together and optimized.

4. Combining and Optimizing the FSTs

For both languages, I finished by creating a single top-level normalizer:

```
- fst = pynini.union(
-     fst_units,
-     fst_teens,
-     fst_exact_tens,
-     fst_compound_tens,
-     fst_hundred_units,
-     fst_hundred_teens,
-     fst_hundred_tens,
-     fst_hundreds_compound,
-     fst_hundreds_,
-     fst_one_thousand
- ).optimize()
```

This ensures:

- full coverage for 0–1000
- minimal number of states after optimization
- fast lookup and decoding

5. Summary of the Methodology

In short, the methodology for both English and French followed these steps:

1. **Define the grammar rules** of each language.
2. **Translate each rule into a small, independent FST.**
3. **Handle irregular ranges (teens, 70–99 in French, etc.).**
4. **Compose larger FSTs from smaller ones** (units → tens → hundreds → thousand).
5. **Union all components into one final normalizer** for each language.
6. **Optimize the final FST** for speed and compactness.

By modularizing everything, I was able to keep the logic clean and ensure that both language implementations remained easy to debug, extend, and analyze.

Compilation Time and Runtime Speed:

It took a total of 0.0011 seconds to compile. The runtime speed was 0.0065 seconds.

Findings:

- French was particularly challenging between compound numbers 71-79, 91-99, 171 - 179, 191 - 199, 271 - 279, 291 - 299, etc. This was because of the irregular compounds, like 71 as soixante-dix, 95 as quatre vingt quinze, etc
  The workaround for this was to create a special FST for numbers within 71 - 79, and 91 - 99 to fix errors like 71 translating to ‘soixante-dix un.`
- English FSTs were more straightforward than French rules, mainly due to the distinct rules for 71-79 and 91 - 99.

Experiments:

I tested these FSTs on numbers such as 105, 115, 995, 850, 400, and 75.
I also tested on full sentences. The src/normalize.py script from the repo can be called to test these normalisers. The full description of the function is available on the GitHub repo.

Conclusion:
This FST works as expected with numbers between 0 and 1000 in both French and English.

The next steps include generating FSTs for numbers that include commas and full stops.