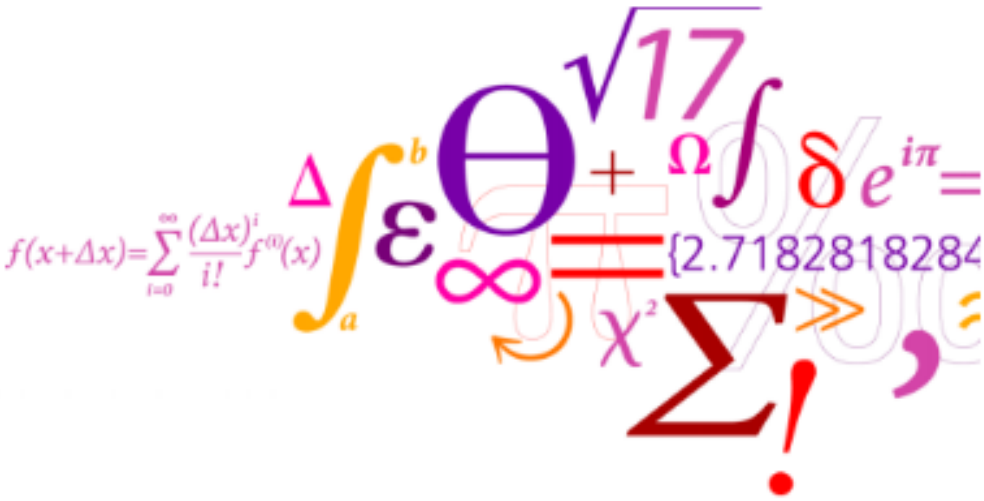
Technical University of Denmark

Implementation of an FSMD simulator in Python

Simon Tobias Lund - s184459
Sara Maria Guijarro Heeb - s184484
Victor Tram - s181104

September 30, 2019



Technical University of Denmark
Course: 02135 - Introduction to Cyber Systems

Introduction

FSMD stands for Finite State Machine with Datapath. It is a computation model built on the transition between possible states. An example of common uses of the finite state machine are regular expression, used for parsing text. A finite state machine is defined by a number of states, with associated operations and transitions to other states. In our FSMD, the operations are performed during the transition between states. The Datapath in FSMD, means that in addition to boolean operations and data types, we also use variables. This allows us to implement complex algorithms as a FSMD.

The finite state machine is just a model of computing, not a specific language or program. We have therefore defined a language to write an arbitrary FSMD in, and a compiler to parse and run it. The language is written as a table in xml, and the compiler is written in python. In other words, our python program parses the xml file, and runs the FSMD defined by it.

1 Constructing an FSMD simulator in Python

Much of our simulator relies on a prewritten code given by the responsibles of the course *Introduction to Cyber Systems*. This code uses the module xmltodict to read the xml-table into a python dictionary. It then reads this data into the dictionary fsmd, which defines all states, operations, and transitions. There is also a function for transforming the condition strings into python code and returning their truth value, and a function for performing operations defined by strings. In our simulation we used this code extentially so we had to write very little code related to parsing, making our task just about going through each state.

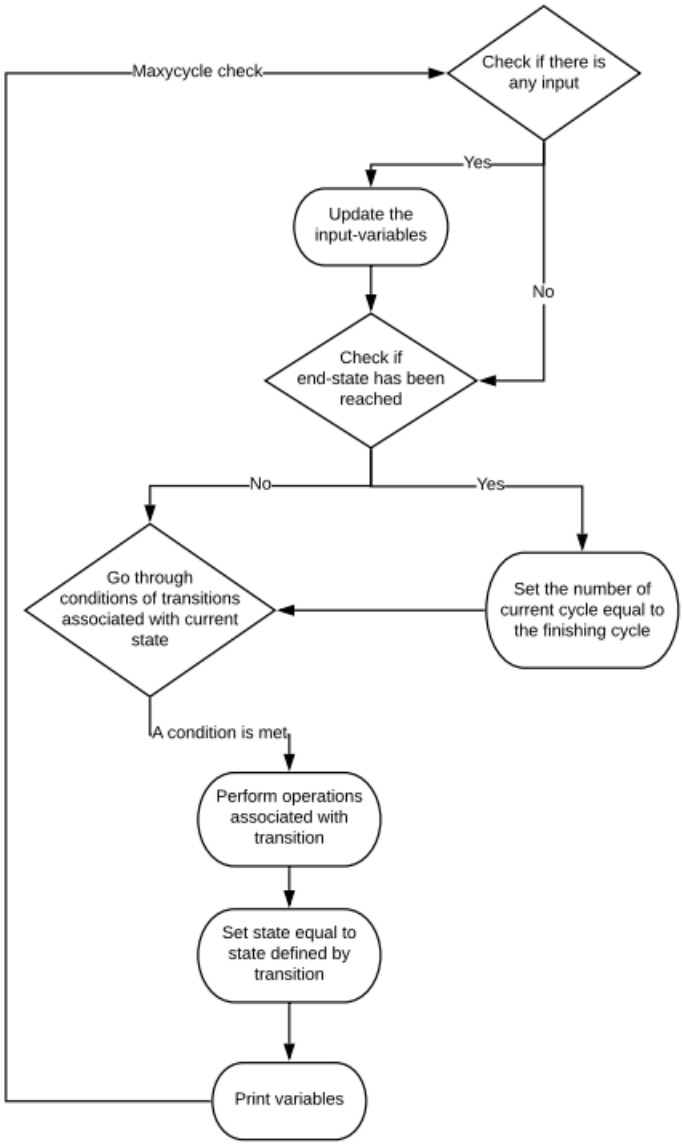


Figure 1: Executing the FSMD - A simple and well-defined cycle for each state.

First of all, to start a loop, the number of cycles is compared to the number of iterations to ensure that the program runs as many iterations as demanded, and prints step by step each cycle calculation. Consequently, checking if the end-state has been reached is the logical step that follows.

Nevertheless, it is also necessary to make sure that any input has been implemented and that it is being executed as demanded in the test files at disposition. As a matter of fact assuring that various inputs can be transcribed is also taken into consideration.

Then the program checks if the conditions, given by the tests at disposition, are met so that the program can run the right commands of the corresponding conditions. After which, the cycle is continued going back to the first step. As is represented in the Figure 1.

2 Developing an FSMD - Newton's Method

2.1 Challenges of the model implemented

For the third test, we decided to implement Newton's method as a FSMD. During development we also considered the cryptographic algorithms, Diffie–Hellman key exchange, but noticed that it did not have any real-world application in one FSMD, as it is an algorithm between two autonomous agents. Newton's method on the other hand seemed more exciting, as it could be used to find quite advanced roots with a simple algorithm.

Newton's method permits the finding of successive approximations to the roots of real-valued functions. The idea is to start by an initial reasonable guess, then used to estimate the function by it's tangent line to which we compute the x-intercept resulting to a better approximation to the original function's root. Subsequently, the method is iterated various times to improve the estimation.

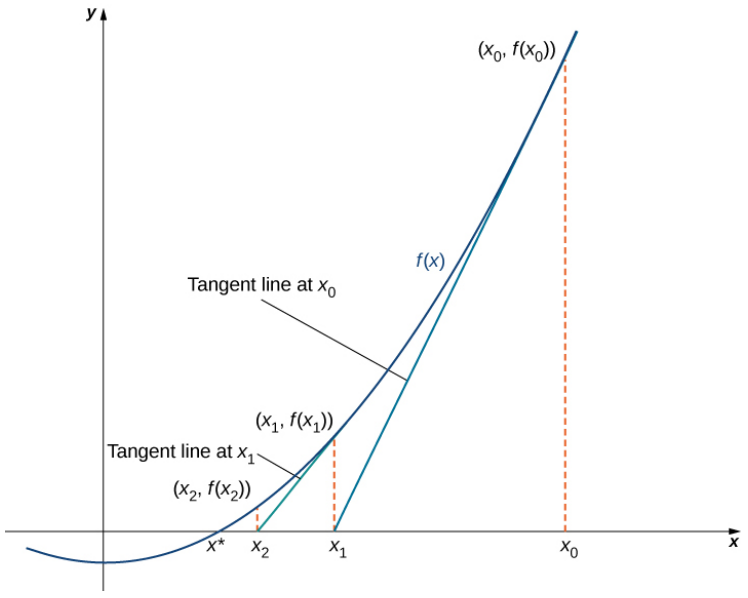


Figure 2: Graph representing the steps of Newton's Method

The x-intercept of the tangent line is taken as the next approximation x_{n+1} to the root studied so that $(x, y) = (x_{n+1}, 0)$ become the coordinates of the tangent line considered. Initially, the equation of a tangent line is the

following :

$$y = f'(x_n) \cdot (x - x_n) + f(x_n) \tag{1}$$

with $y = f(x)$ at $x = x_n$ being the coordinates at which the tangent line is on the curve of the function studied. Moreover, the tangent coordinates $(x, y) = (x_{n+1}, 0)$ are applied to the Eq.1 from which x_{n+1} is solved.

$$0 = f'(x_n) \cdot (x_{n+1} - x_n) + f(x_n) \tag{2}$$

$$\Rightarrow \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3}$$

Furthermore, being able to compute the derivative of the functions studied is a challenge that is thus given by the Eq.3. It was decided that to limit the number of predefined modules, the derivation should happen in the FSMD. Since this is a very complex task to do with exact values, the derivative is approximated. This was handled by using the expression of Newton's difference quotient :

$$f'(x) = \frac{f(x+h) - f(x)}{h} \tag{4}$$

Consequently, the code is ran faster as exact derivation would require many more states. Additionally, estimating the derivative will mildly affect the final result as Newton's method consists itself of an approximation. Despite, to compensate the imprecise computation, one would have to choose a very small value of h.

It was also necessary to take into account trigonometric, exponential and logarithmic functions as they are not included in the initial script. This was done by editing the code in the provided function `execute_operation()`, to incorporate functions from the module math.

```
variables[target] = eval(expression,{'sin':sin,'cos':cos,'log':log,'exp':exp,'pow':pow,'__builtins__': None},
                    merge_dicts(variables, inputs))
```

As shown above, the line includes a dictionary commanding the function `eval()` to read the string *"sin"* as the sinus function, and so on for the other functions.

2.2 Functionality of the FSMD developped

Much as in assembly languages we have chosen to include two general registries for storing values during calculation, var_A and var_B. This meant that x would be equal to the approximation of the root at all times, instead of having it be the root just at the first state, for instance. The downside of this is that the FSMD requires more memory to run.

Newton's algorithm becomes more precise the longer it runs, there is therefore no defined end-state. Moreover as it basically consists of doing the same operations on the data over and over again, no conditions had to be met. The program flow therefore became quite cyclical, running the steps : `INITIALIZE` → `STEP1` → `STEP2` → ... → `STEP6` → `STEP1`... Finally, since each step does not define any meaningful situation, we kept the naming scheme simple.

Regarding the implementation, most of it was very straight forward, just translating the design into python code, and a xml-table. We made a small change to the source code we had been handed, and made quite a few ad-hoc decisions regarding the dataflow of the algorithm for Newton's method.

For readability's sake, but at the cost of performance, we ended up printing information about almost everything for each state. Each state prints which cycle is currently running, the current state, which condition has been met, what operation has been executed, what the next state is, and the value of all variables.

The cycle of our Newton's Method FSMD is represented underneath to offer an idea of it's functionality.

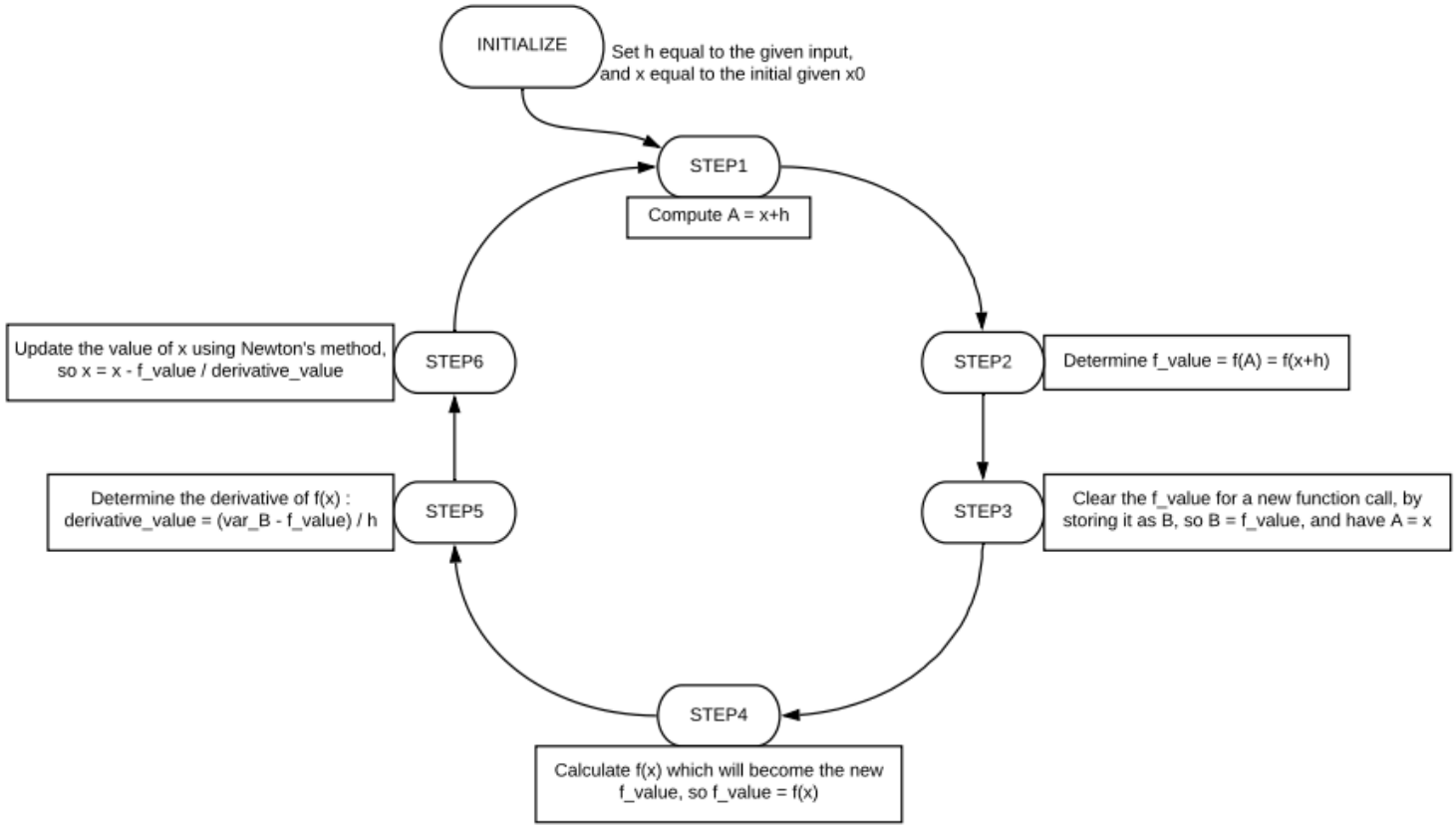


Figure 3: Cycle of Newton's Method FSMD

Finally, the simulation can parse and execute all three tests. We couldn't manage to have it parse multiple conditions for one state, but decided it wasn't too necesary as one can just write the conditions as one condition combined

by “and”. Newton’s method converges on the root with quadratic precision for single roots, and linear precision for roots with multiplicity. The limit for the precision of our algorithm was mostly the size of h, which can be changed. The number of cycles also has an effect, especially for roots with multiplicity.

The code was ran with the following function :

$$f(x) = (\log(sin(x)))^2 \tag{5}$$

From which we noticed that the error was around 0.023 after 100 cycles, 0.007 after 200 cycles, and 0.003 after 300 cycles. There was also a problem for some combinations of cycle-number and functions that $f'(x)$ became so close to zero that function crashed due to division by zero. This for example happened for our previous $f(x)$ with $h = 0.0000000001$ and $x_0 = 2.5$ at 295 cycles. By running without printing intermediate cycles the speed was greatly improved.

Conclusion

Our implementation worked fairly well overall, with all programs running as specified.

Something we discovered was that a FSMD isn’t a particularly good implementation of Newton’s algorithm. The positive sides of FSMD’s, with state-control and easy implementation of algorithms with many conditionals, wasn’t necessary to use for Newton’s algorithm, while the negative sides, like the complicated UI and forced conditional checks at every step, affected both ease of implementation and performance. If we were to do the project again we would have chosen an algorithm more fitted to the FSMD-model.

In future versions, the problem with division by zero could be worked around by having the end-state defined as an error state, giving current value of x and a warning to the user before stopping the simulator. Since there is only one end state, the downside of this is that this would prevent us from using the end state for anything else.

Showing the display at every cycle slowed down the program a great deal. We decided on not removing this, as it made debugging easier and also showed the user how the FSMD runs. A way of reconciling these considerations would have been to allow the user to turn on and off the display.