CBD Assignment

By Arno Deceuninck (s0181217) and Dogukan Altay (s0211552)

Missing std parts

The code is in CBD.lib.std and all tests related to this passed. There were initially some problems with the getDependencies from DelayBlock, but this was fixed later.

Linear solver

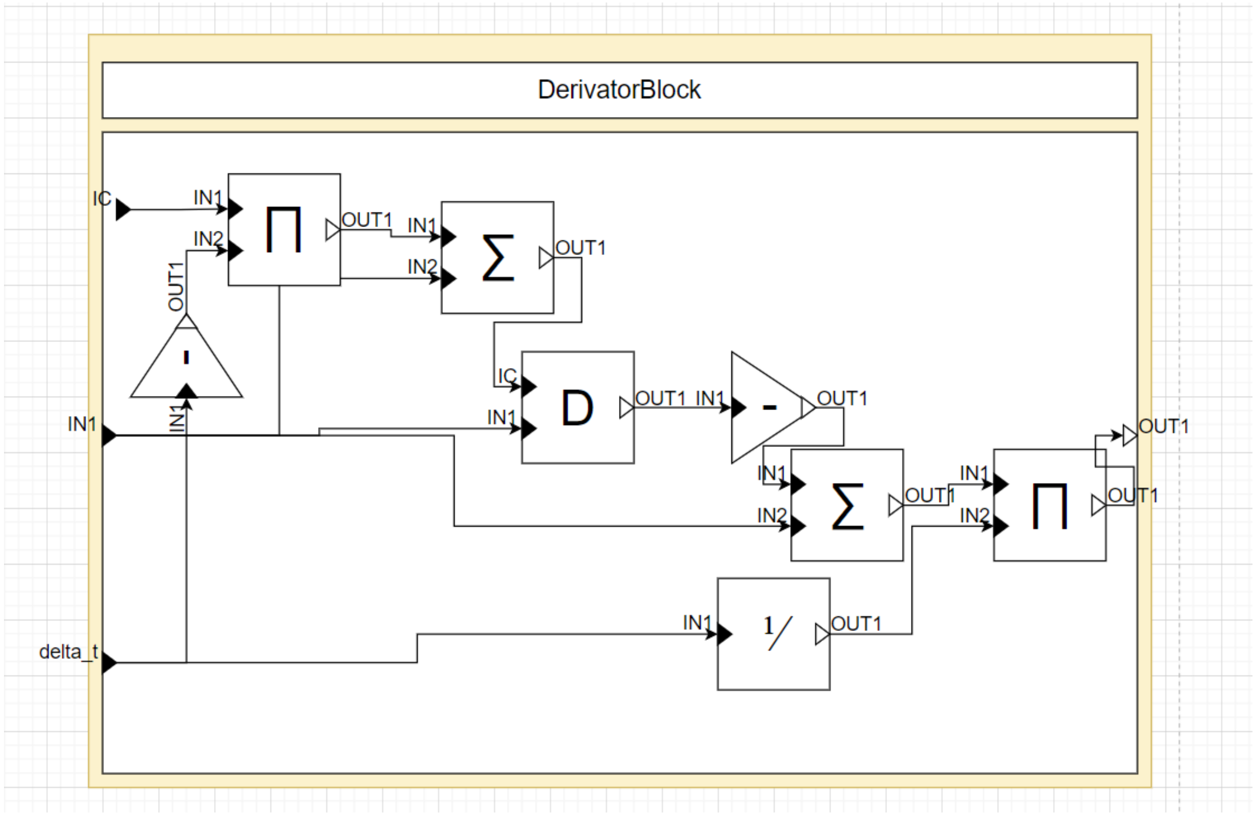
For this, we separated some different cases. An addition is never a problem, since variables added to each other stay linear. A multiplication is possible if there is only one variable involved in it. Logical operations are never possible. All other operations can exist in a linear equation as long as their value is known, meaning that the blocks connected to their input port are not part of the strong component. After implementing these cases in the __isLinear function, the tests related to this passed.

Derivator and Integratorblock

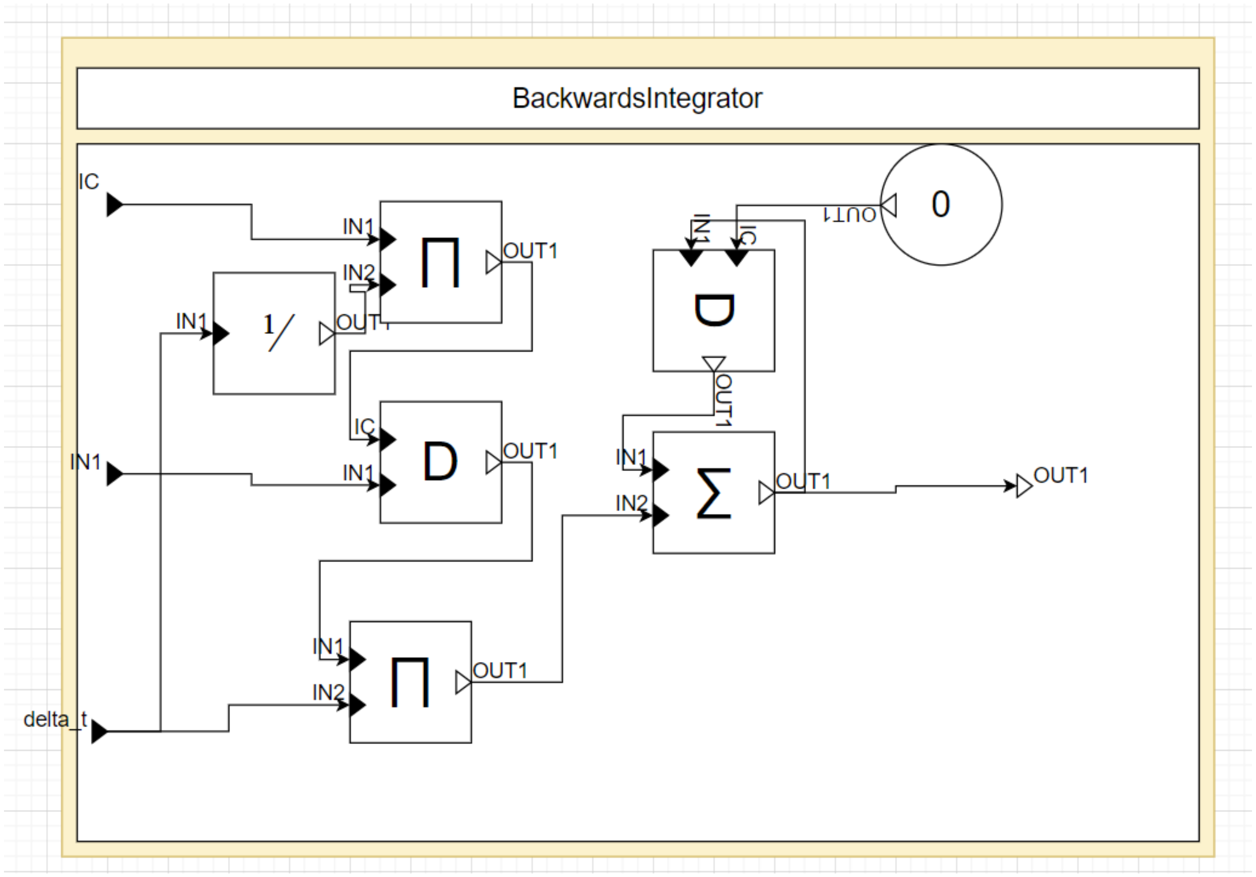
For these blocks, we based ourselves on the recorded lectures. The formula we implemented was:

$\widetilde{y}(t) = \frac{\widetilde{x}(t) - \widetilde{x}(t-\Delta t)}{\Delta t}$

This could be easily implemented with the basic blocks. $\widetilde{x}(t)$ was given as input. $\widetilde{x}(t - \Delta t)$ could be obtained with a delay block. The only problem is the value of this at iteration 0 .To solve this, the derivatorblock also has an IC input and based our IC for our DelayBlock on this by reforming the formula such that $IC_D(0) = \widetilde{x}(0) - IC_d(0)\Delta t$ where $IC_D$is the IC of the DelayBlock and IC_d the IC of our Derivatorblock.



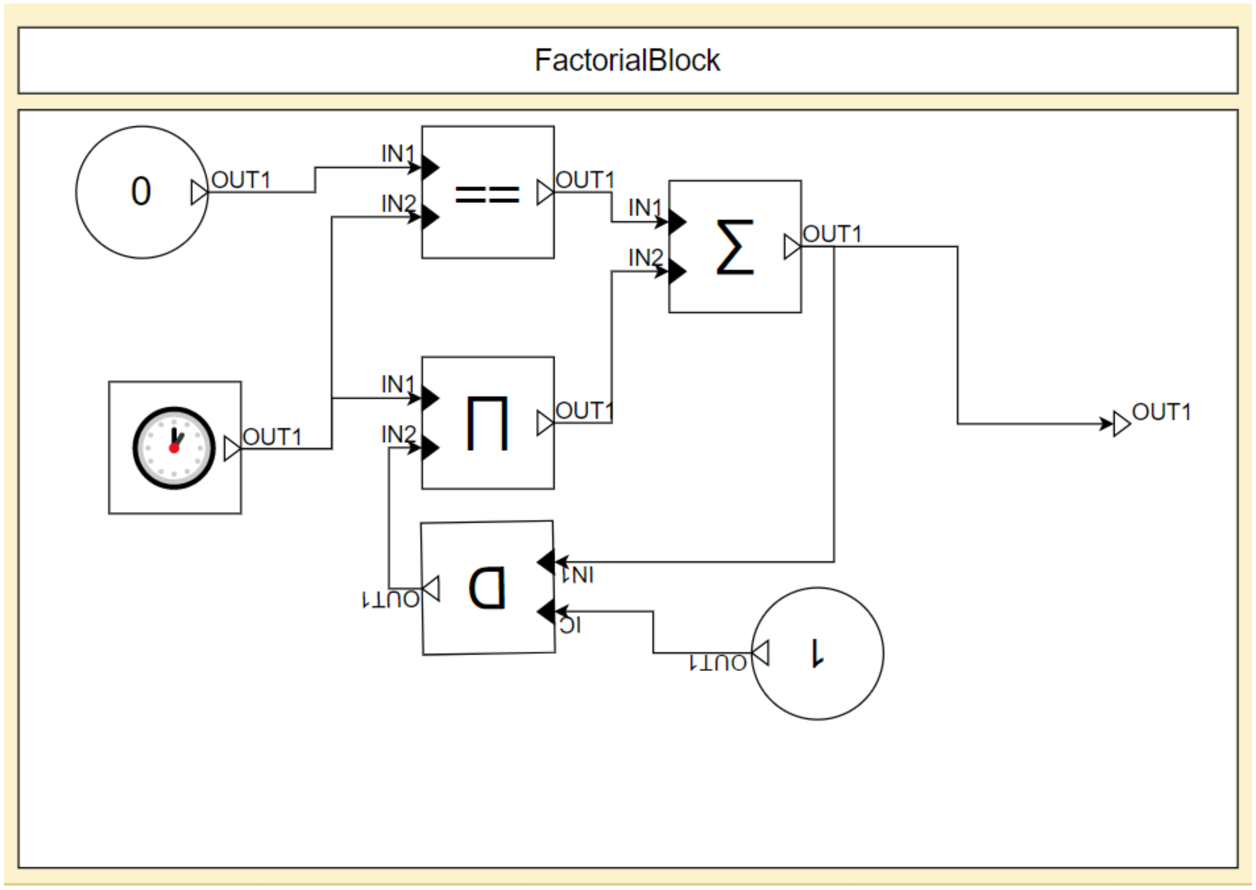
The IntegratorBlock was done in a similar way with the formula $x_n = x_{n-1} + f(x_{n-1})\Delta t$. Also here a DelayBlock could be used to remember x_{n-1} and $f(x_{n-1})$. The initial condition of $x_{n-1}$is 0 and for $f(x_{n-1})$ we based us on an IC value of the integratorblock by transforming the equation to $f(x_0) = \frac{x_0}{\Delta t}$. With this, everything could be implemented with the basic blocks and all tests passed.



For both blocks, their generated latex equations (with the variable names changed to an easier readable letter) can be found in latex.pdf.

Factorials

Factorials are defined as $f(i) = i * f(i-1)$ and $f(0) = 1$. This could be implemented with a DelayBlock and a ProducBlock. The only problem then is the first number, because you would get $f(0) = IC * 0 \neq 1$, so your initial condition doesn't matter. To fix this, we added the result of $i == 0$ to the output, since this is only 1 in the first iteration and 0 in all the others. The code for this can be found in CBD.lib.drawio. A test can be found in test.drawioTest.py. Also for this block the generated latex equations (with the variable names changed to an easier readable letter) can be found in latex.pdf.

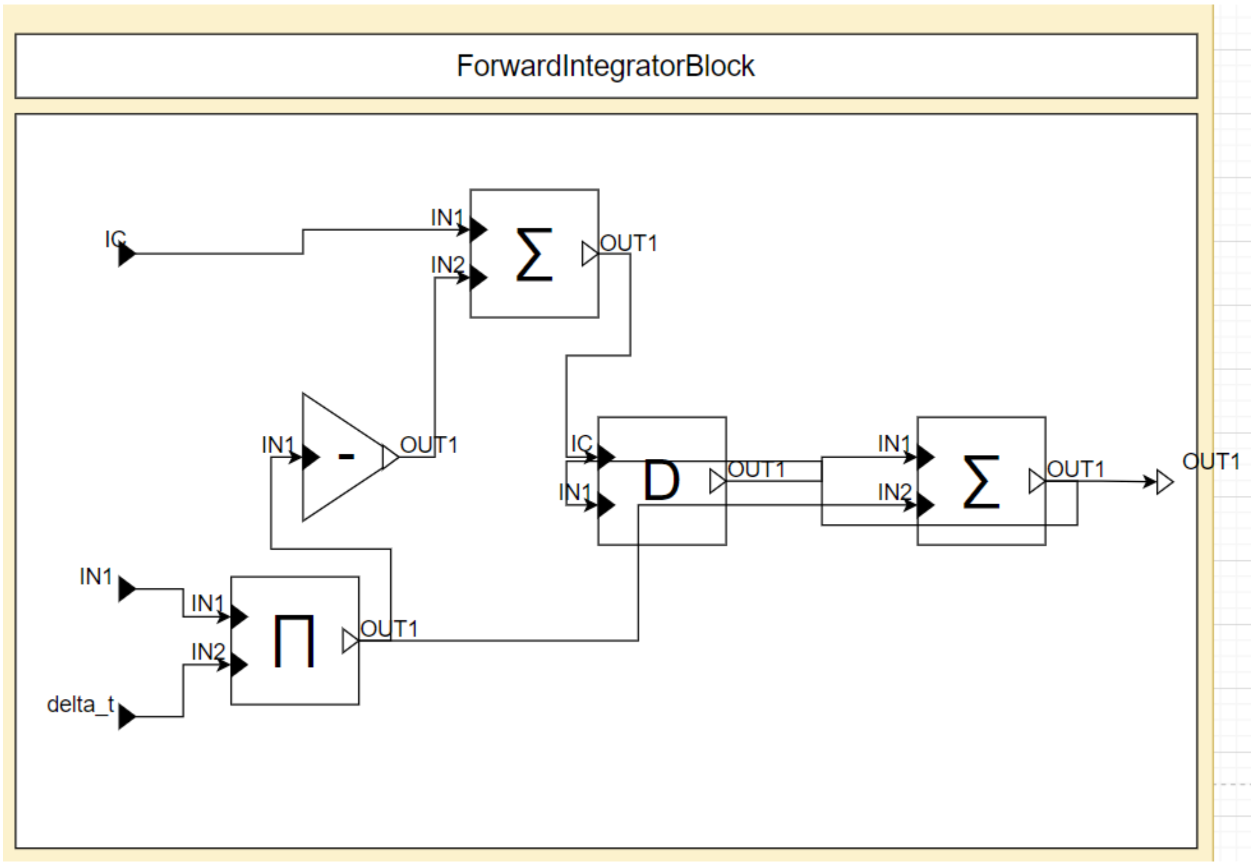


Other integrators

The code of the integrators can be found in CBD.lib.drawio.py. The tests for this in the test.drawioTests.py file. Also for these blocks the generated latex equations (with the variable names changed to an easier readable letter) can be found in latex.pdf.

Forwards Euler

The forwards euler rule was similar to the backwards euler rule, but with the formula $x_n = x_{n-1} + f(x)\Delta t$, so always looking forward at the value instead of back. The initial value for x_{n-1} could be found by subtracting $f(x)\Delta t$ from the initial value given to the integrator block.

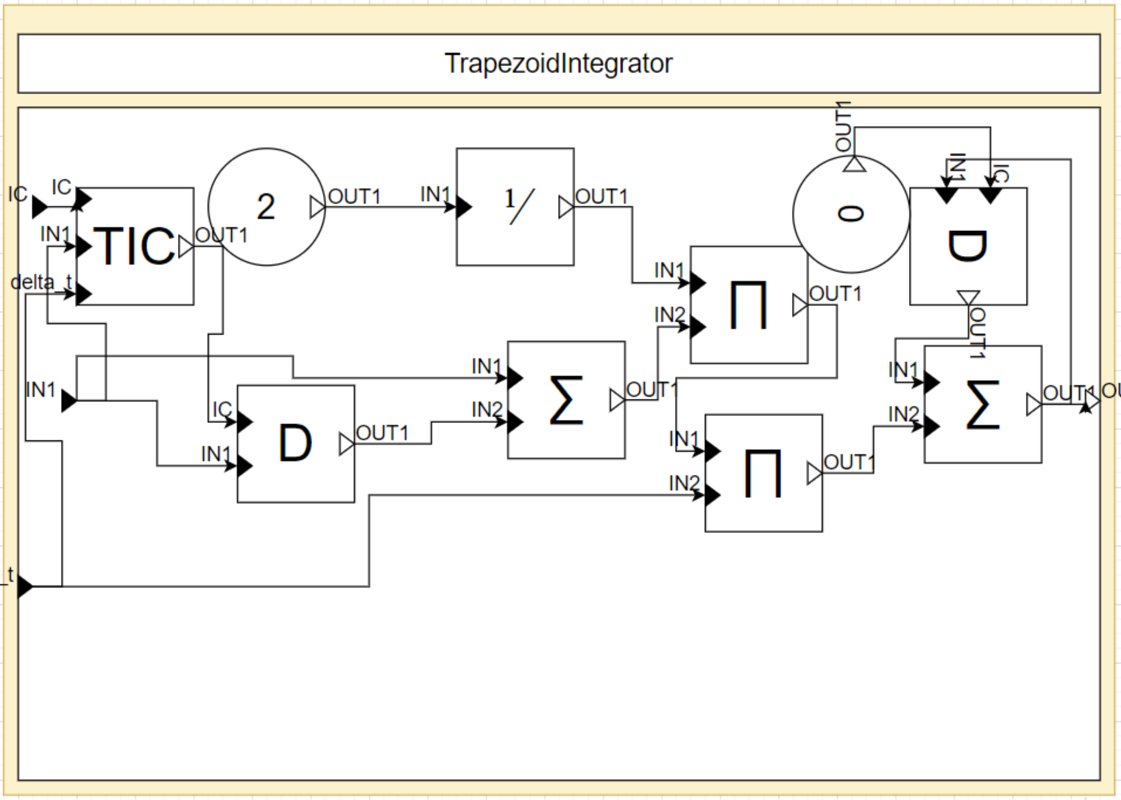
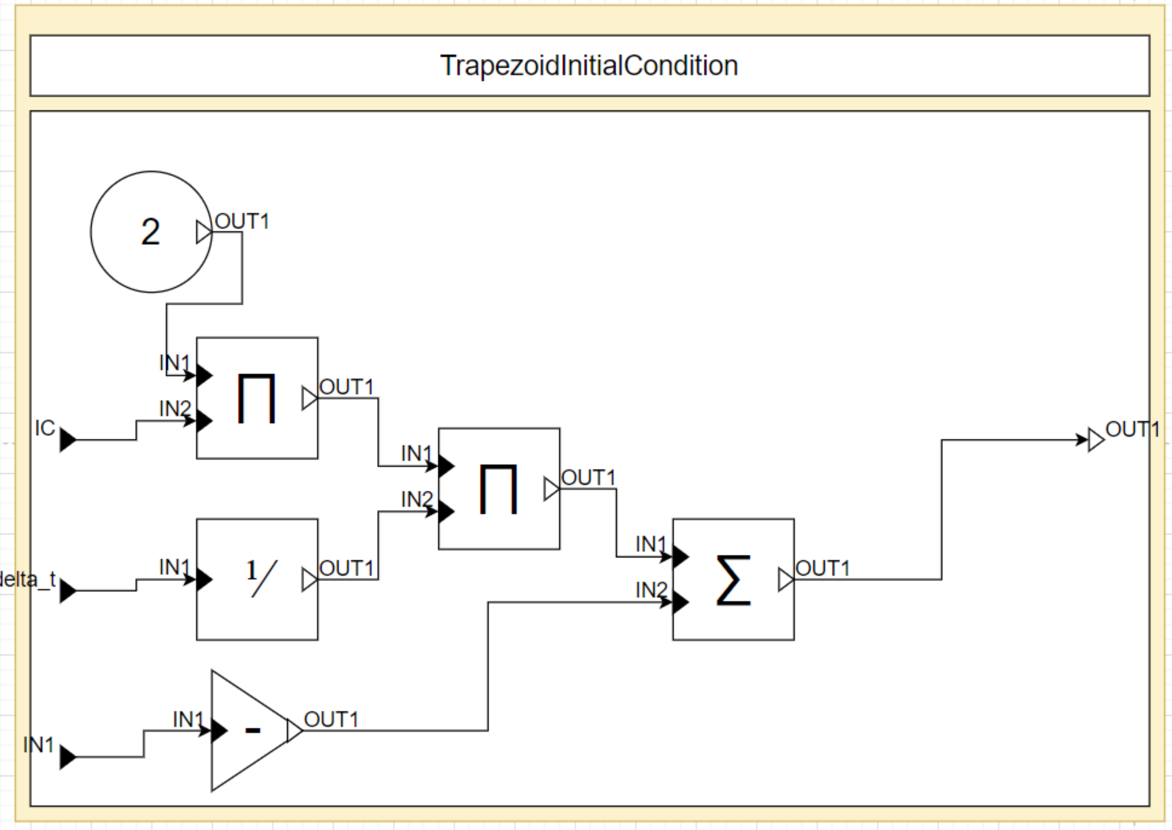


Trapezoid

For this, we based us on the formula we found on the Wikipedia page (https://en.wikipedia.org/wiki/Trapezoidal_rule).

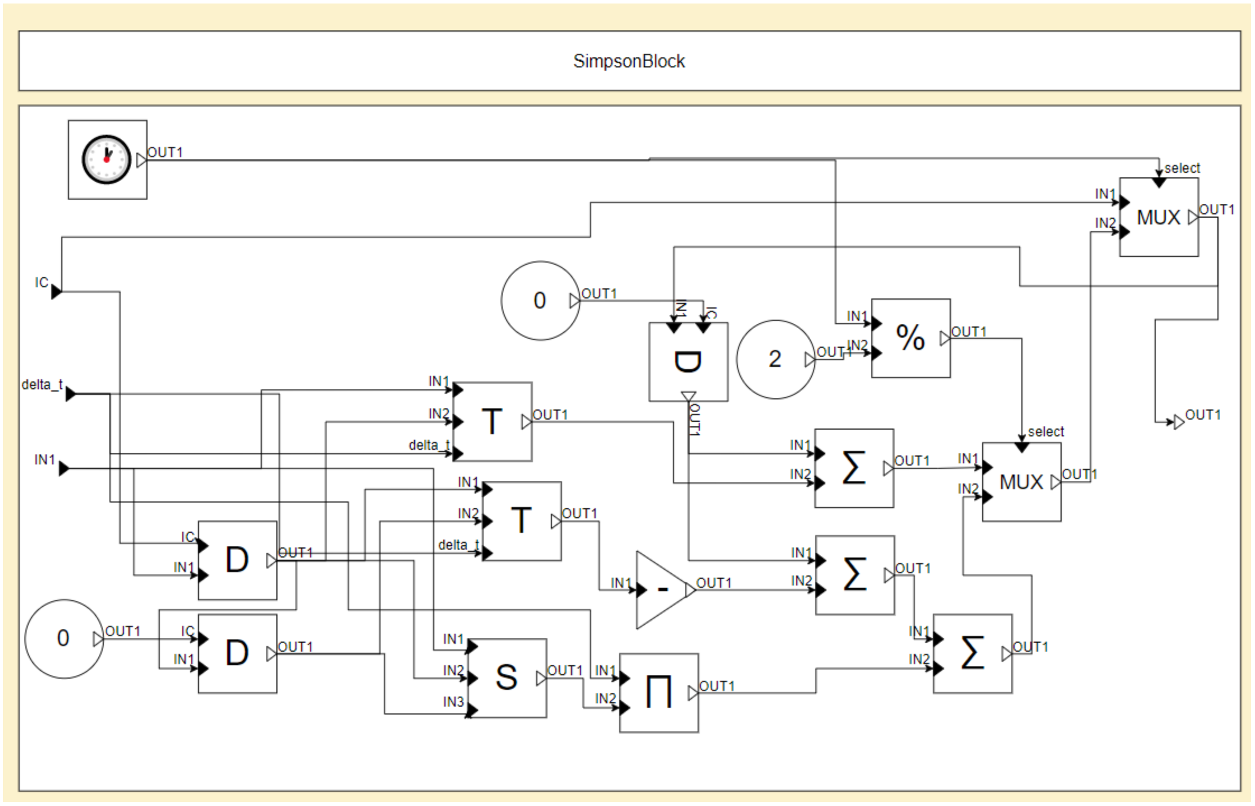
$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k$$

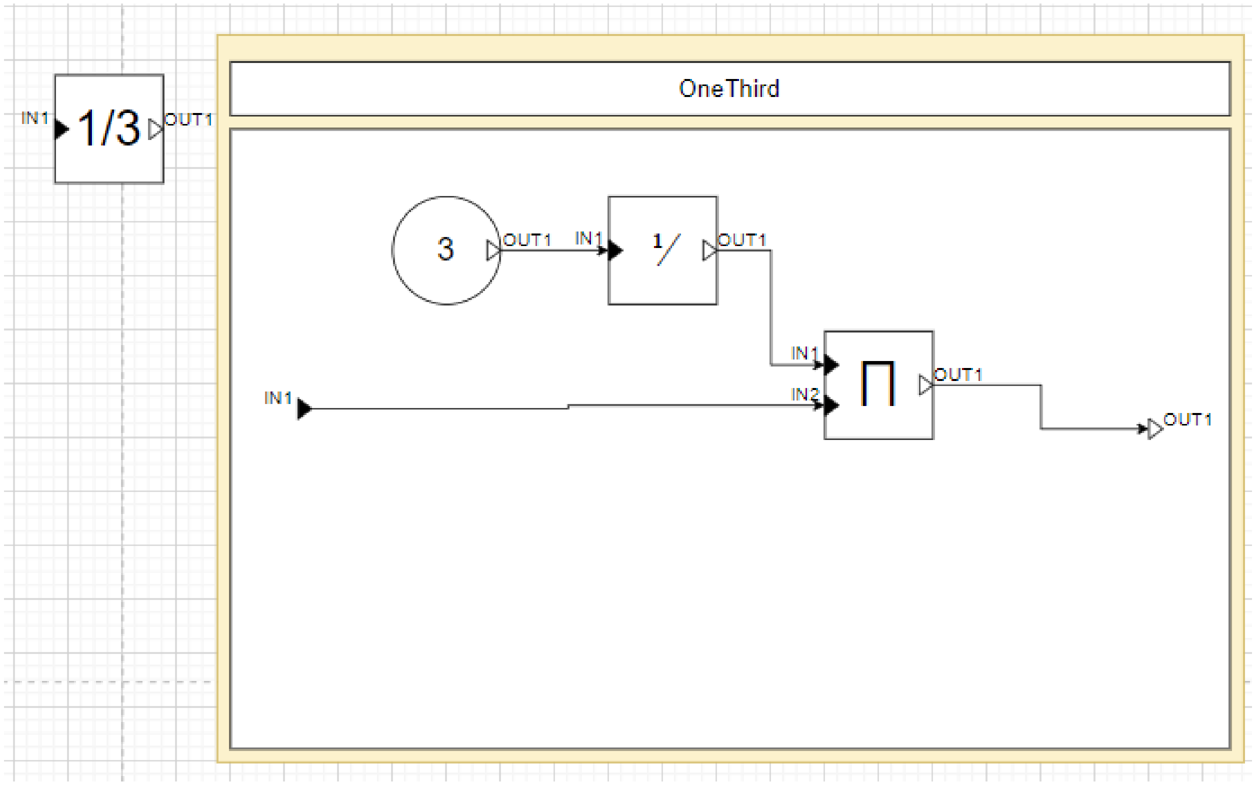
So we added $\frac{f(x_{k-1}) + f(x_k)}{2}\Delta x_k$ each iteration to the result of the previous iteration. For finding the initial value of $f(x_{k-1})$, we used a separate block (to keep everything more readable). The formula for this initial value could be found based on the IC value given to the TrapezoidIntegrator by transforming the formula to $IC_D(0) \; = \; \frac{2\,IC_i(0)}{\Delta t} - \; f(x_0)$.


TrapezoidIntegrator
IC
TIC
IN1
delta_t
OUT1
2
1/
D
Σ
Π
0
IN2
delta_t

TrapezoidInitialCondition
2
OUT1
IC
delta_t
IN1
Π
1/
-
Σ
IN2

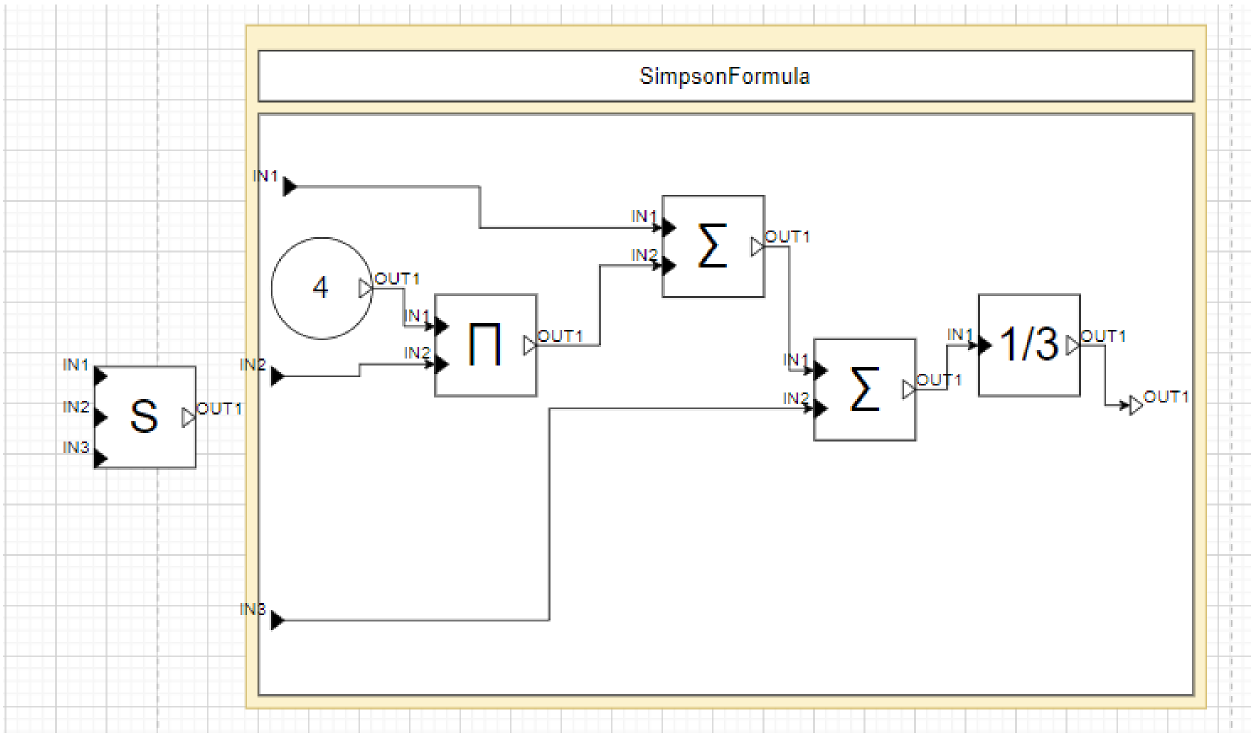
Simpson

To use Simpson’s rule, we must have 3 points, so we remember the two previous values always with two DelayBlocks. When we have 3 points, we can just apply the formula. When we have only 2 points (e.g. 2 remaining points after grouping all other points in groups of 3), we used the Trapezoid rule. We couldn’t know wether it was currently the last point (so whether we had to use the Simpson rule or Trapezoid rule), but this problem was solved by always applying the Trapezoid rule when a new point enters, and on the next point subtracting this added value and adding the Simpsons rule instead (by using a Multiplexer with a ModuloBlock this could be done alternating). Last special case is in iteration 0, where we have only 1 point, but then our initial value could be immediately sent to the output (also with a multiplexer block). To improve readability, this was also splitted over multiple blocks.

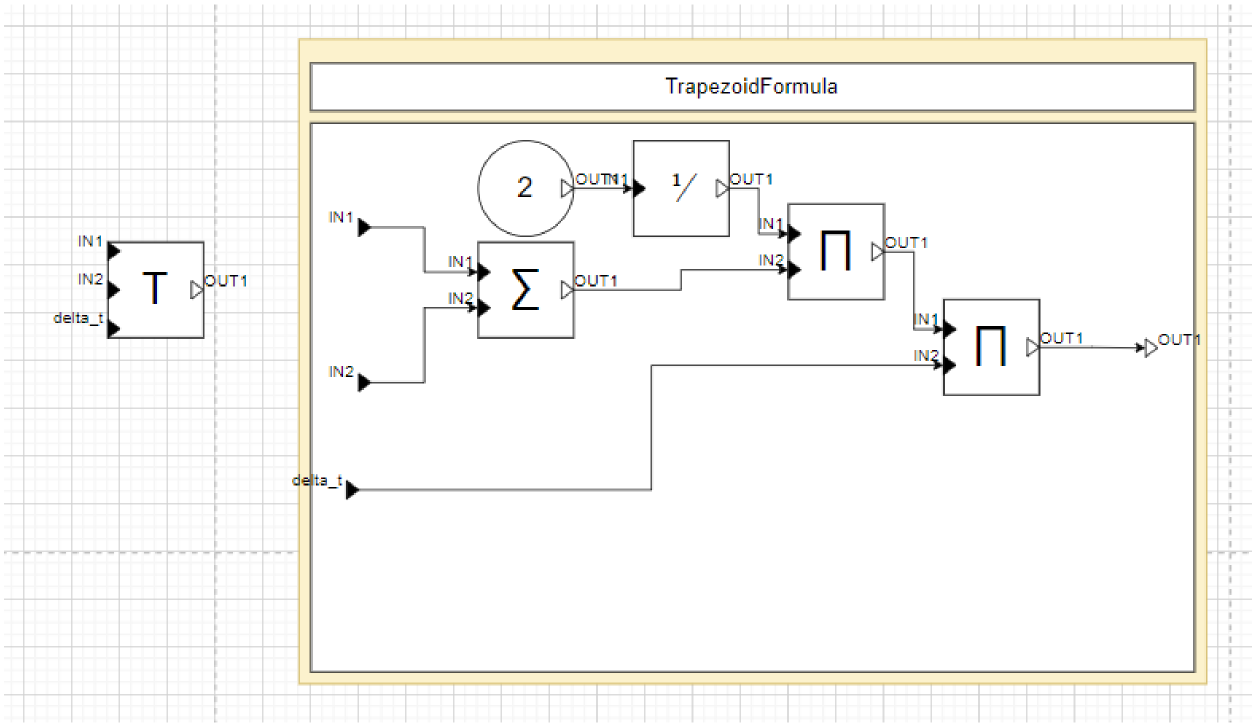




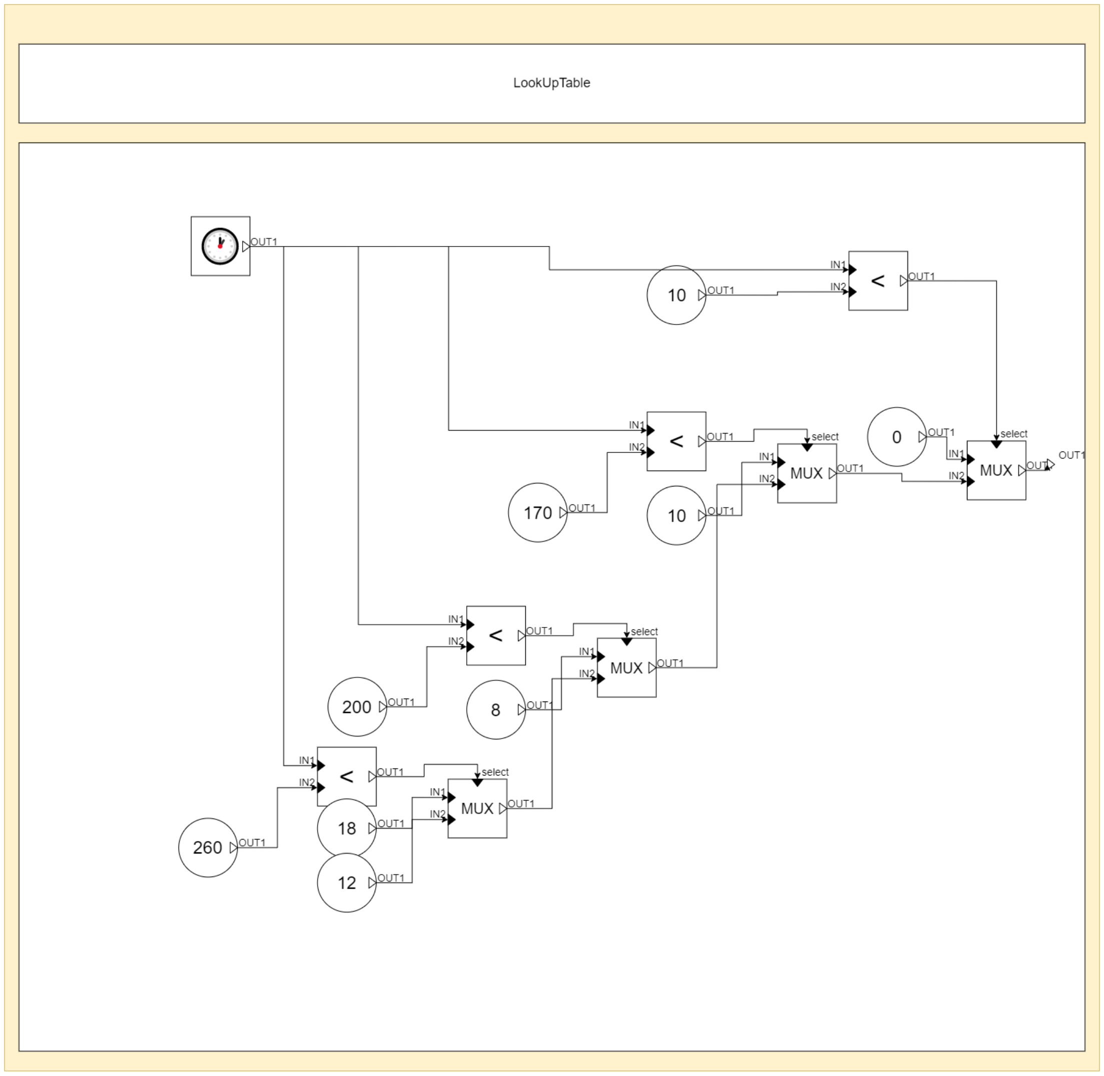
OneThird
IN1
1/3
OUT1
3
OUT1
IN1
1/
OUT1
IN1
IN2
Π
OUT1
IN1
OUT1



SimpsonFormula
IN1
IN2
IN3
S
OUT1
IN1
IN1
IN2
Σ
OUT1
4
OUT1
IN1
IN2
Π
OUT1
IN2
IN1
IN2
Σ
OUT1
IN1
1/3
OUT1
OUT1
IN3


IN1
IN2
delta_t
T
OUT1
TrapezoidFormula
2
OUT1
IN1
1/
OUT1
IN1
IN1
IN2
∑
OUT1
IN1
IN2
∏
OUT1
IN1
IN2
∏
OUT1
OUT1
IN2
delta_t

LookUpTable Block



Above diagram shows the CBD model of the lookup table that has been used in the previous assignment. We used multiplexer blocks to achieve the desired behaviour.
The code of the LookUpTable can be found in CBD.lib.drawio.py. The tests for this in the test.drawioTests.py file. Also for these blocks the generated latex equations (with the variable names changed to an easier readable letter) can be found in latex.pdf.