MoSIS Assignment 3

Team

Name : Gauthier Le Compte
Student ID : s0172093
e-mail : gauthier.lecompte@student.uantwerpen.be

Name : Nick Wils
Student ID : s0162945
e-mail : nick.wils@student.uantwerpen.be

Time spent on this assignment (in hours): 36
Link to code: https://github.com/NickWils98/Mosis3

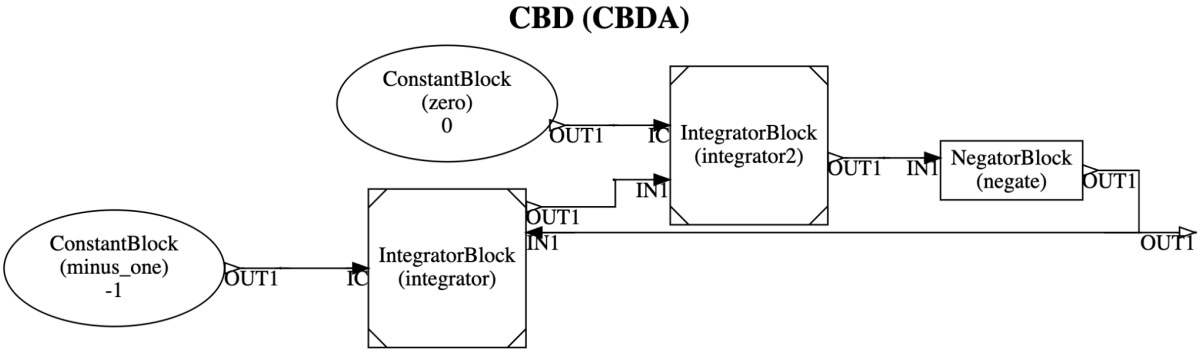
Note

We have not always included all models in our report and they are not all equally clear as they vary in size, but all the images of plots and models can be found in the resc folder.

Harmonic Oscillator

CBDA

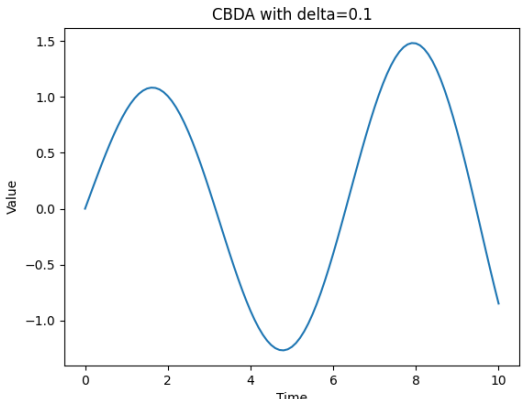
Model

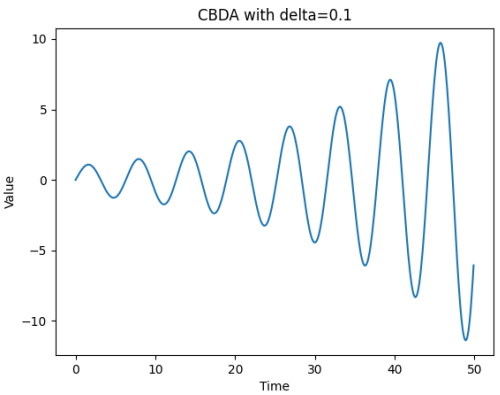


Since we need to solve a second order ODE, we need to use 2 integration blocks. As we need to take the second order primitive. For the first one we need an initial condition -1, this is because we take the primitive of -x and it was stated that dx / dt = 1. For the second order primitive we need to use an initial condition of 0, as it is stated that x(0) = 0. The input of the Integrals is -x so we just negate the solution. The result is then outputted and also used as the input of the first integral to make this circular.

Plot

When running the continuous-time, coupled CBD that makes use of IntegratorBlocks for 10 and 50 seconds, we got the following result:





In the beginning, this CBD copes well with the sinusoidal shape with values between -1 and 1. We also see that as time progresses, it goes more and more outside of this scope. That is why we also ran this simulation for 50 steps as you can see in the right plot. It is immediately noticeable that things are going badly with this plot and after step 50 it is approximately between -10 and 10 instead of -1 and 1. Things will only get worse the longer we run it. We can explain this behaviour, by noting that we work with integrals. So we work with higher order primitives and when time goes on, the error marge grows.

Validity

When running the CBDA validity check we get the following error message:

- RecursionError: maximum recursion depth exceeded while calling a Python object

This can be explained by the fact that with the CBDA we work with 2 integrators that are connected to each other and are therefore circular. When we run the validity check, the program simply doesn't know when to stop since this process keeps on going for an infinite amount of time, and thus throws the maximum recursion depth error.

We do however get an output until a certain point, which can be found below:
INITIAL SYSTEM:
integrator.OUT1(i) = integ(integrator.IN1(i), 0, (i))
integrator2.OUT1(i) = integ(integrator2.IN1(i), 0, (i))
zero.OUT1(i) = 0
one.OUT1(i) = 1
negate.OUT1(i) = (-negate.IN1(i))
integrator.IN1(i) = negate.OUT1(i)
integrator.IC(i) = zero.OUT1(i)
integrator2.IN1(i) = integrator.OUT1(i)
integrator2.IC(i) = one.OUT1(i)
negate.IN1(i) = integrator2.OUT1(i)
OUT1(i) = negate.OUT1(i)

STEP 1: substituted all connections and constant values
integrator.OUT1(i) = integ(negate.OUT1(i), 0, (i))
integrator2.OUT1(i) = integ(integrator.OUT1(i), 0, (i))
negate.OUT1(i) = (-integrator2.OUT1(i))
OUT1(i) = negate.OUT1(i)

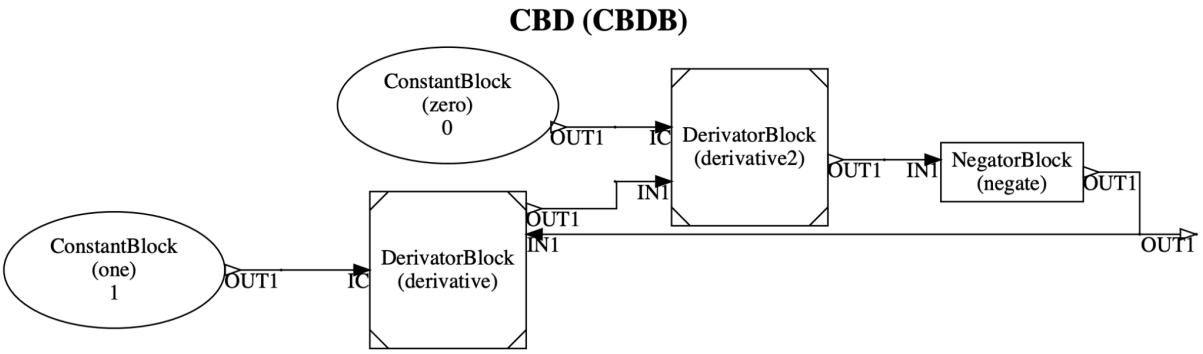
STEP 2:
integrator.OUT1(i) = integ(negate.OUT1(i), 0, (i))
integrator2.OUT1(i) = integ(integrator.OUT1(i), 0, (i))
negate.OUT1(i) = (-integrator2.OUT1(i))
OUT1(i) = (-integrator2.OUT1(i))

STEP 3:
integrator.OUT1(i) = integ(negate.OUT1(i), 0, (i))
negate.OUT1(i) = (-integ(integrator.OUT1(i), 0, (i)))
OUT1(i) = (-integ(integrator.OUT1(i), 0, (i)))

STEP 4:
negate.OUT1(i) = (-integ(integ(negate.OUT1(i), 0, (i)), 0, (i)))
OUT1(i) = (-integ(integ(negate.OUT1(i), 0, (i)), 0, (i)))

CBDB

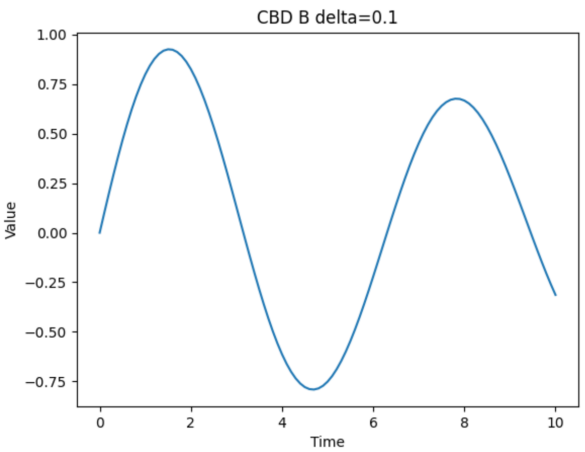
Model

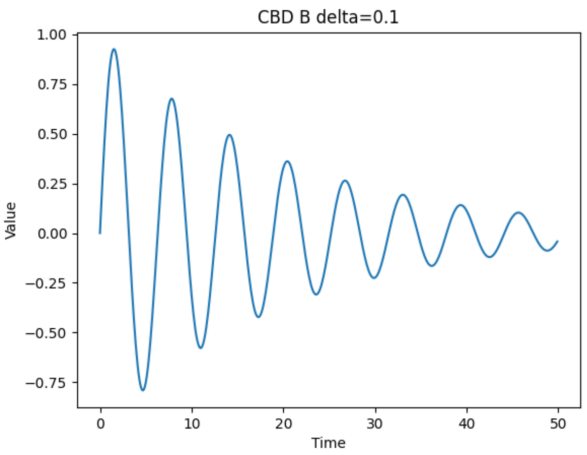


The CBDB Model looks exactly the same as the CBDA model discussed above, with the exception of the integrator blocks that have now become derivator blocks. Also the Initial conditions have been changed here. This is because derivators are the opposite of integrators so we changed the IC=-1 to IC=1 and the inverse of 0 stays zero of course.

Plot

The CBD B is very similar to CBD A discussed above, only here we use derivative blocks instead of integrator blocks. When running the continuous-time, coupled CBD for 10 and 50 seconds, we got the following result:





Again, here we see in the beginning that it closely maintains the sinusoidal shape within the limits of -1 and 1, but also after 10 seconds we see that it probably won't hold very well over time. In the right picture we have another simulation of 50 steps to see the effect of longer runtimes. Since derivatives are the opposite of integrators, it's no surprise that instead of overriding the limits of -1 and 1, it stays within those limits but converges to a much smaller number (to 0 after infinite time).

Validity

When running the CBDA validity check we get the following error message:

- RecursionError: maximum recursion depth exceeded while calling a Python object

For the CBDB we get the exact same error, which is also due to the same reason, except here we use 2 Derivator blocks instead of 2 Integrator blocks. Again, we also got output until a certain point where we received the error message.

INITIAL SYSTEM:
derivative.OUT1(i) = der(derivative.IN1(i), 0, (i))
derivative2.OUT1(i) = der(derivative2.IN1(i), 0, (i))
one.OUT1(i) = 1
zero.OUT1(i) = 0
negate.OUT1(i) = (-negate.IN1(i))
derivative.IN1(i) = negate.OUT1(i)
derivative.IC(i) = one.OUT1(i)
derivative2.IN1(i) = derivative.OUT1(i)
derivative2.IC(i) = zero.OUT1(i)
negate.IN1(i) = derivative2.OUT1(i)
OUT1(i) = negate.OUT1(i)

STEP 1: substituted all connections and constant values
derivative.OUT1(i) = der(negate.OUT1(i), 0, (i))
derivative2.OUT1(i) = der(derivative.OUT1(i), 0, (i))
negate.OUT1(i) = (-derivative2.OUT1(i))
OUT1(i) = negate.OUT1(i)

STEP 2:
derivative.OUT1(i) = der(negate.OUT1(i), 0, (i))
derivative2.OUT1(i) = der(derivative.OUT1(i), 0, (i))
negate.OUT1(i) = (-derivative2.OUT1(i))
OUT1(i) = (-derivative2.OUT1(i))

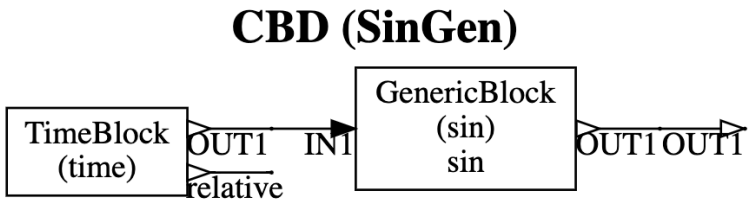
STEP 3:
derivative.OUT1(i) = der(negate.OUT1(i), 0, (i))
negate.OUT1(i) = (-der(derivative.OUT1(i), 0, (i)))
OUT1(i) = (-der(derivative.OUT1(i), 0, (i)))

STEP 4:
negate.OUT1(i) = (-der(der(negate.OUT1(i), 0, (i)), 0, (i)))
OUT1(i) = (-der(der(negate.OUT1(i), 0, (i)), 0, (i)))

SinGen

Model

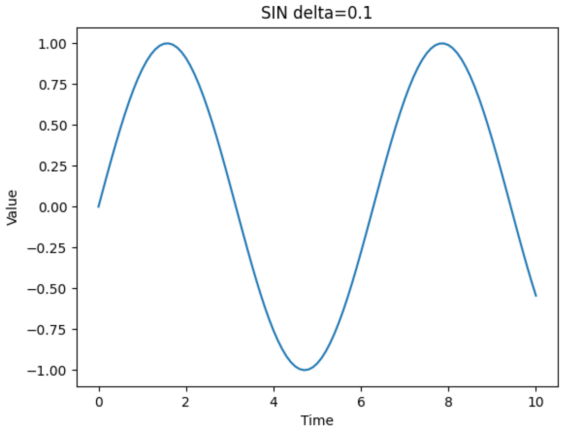
When we convert the simple sinus function into a CBD we get the following model:

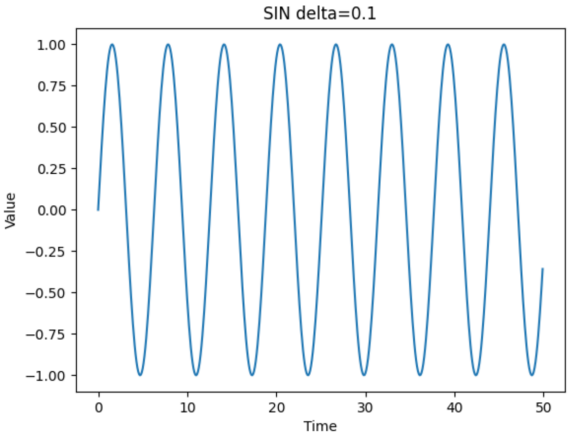


Here we just use a simple time block that simulates time. The output goes as input to a generic block with sine as operator. The output from this generic block then generates the plots.

Plot

In the plots below you can see the standard sine function. Both run for 10 and 50 steps. This is useful to see what we tried to approach in previous CBDs.





Validity - SinGen

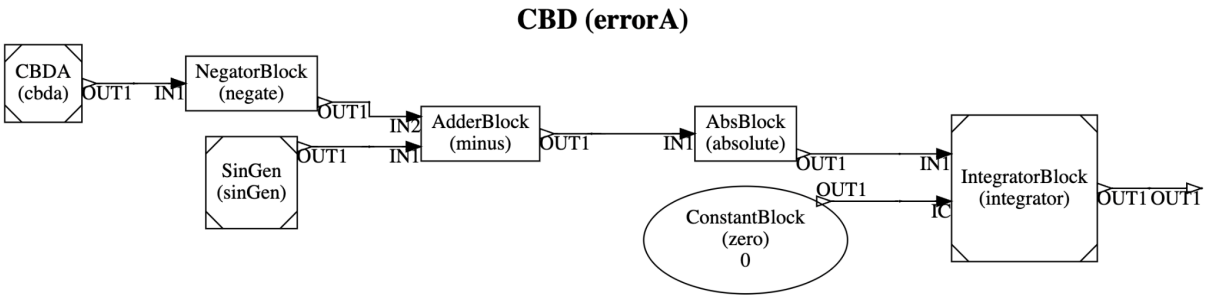
INITIAL SYSTEM:
time.OUT1(i) = time(i)
sin.OUT1(i) = sin(sin.IN1(i))
sin.IN1(i) = time.OUT1(i)
OUT1(i) = sin.OUT1(i)
STEP 1: substituted all connections and constant values
sin.OUT1(i) = sin(time(i))
OUT1(i) = sin.OUT1(i)
STEP 2:
OUT1(i) = sin(time(i))

RESULT IS:
OUT1(i) = sin(time(i))

ErrorA

Model

In the image below you can see how we found the error between our CBDA and the real sinus function.



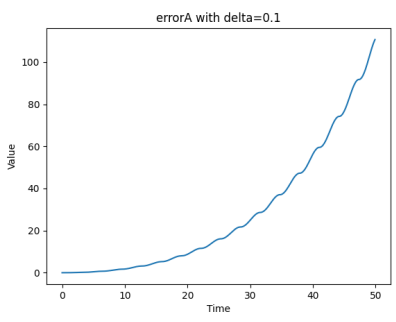
We took our previously generated SinGen model, added it to the first input of the adder. Then we took the also previously generated CBDA model, negated it because we have to subtract it from sin(t) and add the output Xa to the second input of the adder. Now we have Sin(t) + (-Xa), which corresponds to Sin(t) - Xa. The output of the adder is connected with an abs block such that we take the absolute value | Sin(t) - Xa | and the output of this we connect with the integrator function.

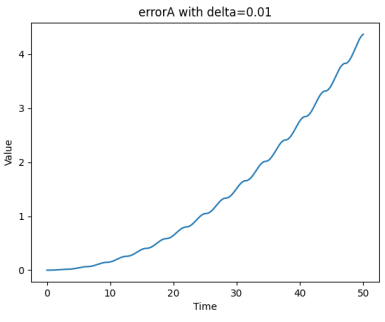
Plot

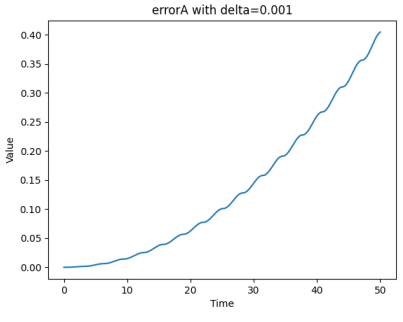
Error A is a coupled CBD that calculates the error between the output of CBDA (Xa) and the output of our Sinus Generator (Sin(t)). When we run this CBD for 50 seconds with

- Δt = 0.1
- Δt = 0.01
- Δt = 0.001

we got the following plots:







The rate of the error increases as we take more steps, which is normal because if we look back at the plot of CBD A, we see that it mimics sinus function very well at first, but the longer we run the CBD, the more it gets blown out of proportion. Hence we see an upward trend here and the rate of the error continues to increase as the difference between Sin(t) and CBD A gets bigger and bigger.

The rate of error (the angle of the line) is lower when the delta is lower. The impact of the step size is rather big. The CBDA models the sinus function a lot closer when the step size is smaller. The error changed from >100 to <0.5 when the delta dropped from 0.1 to 0.001. When taking smaller steps, the error margin drops significantly.

Validity

When running the ErrorA validity check we get the following error message:

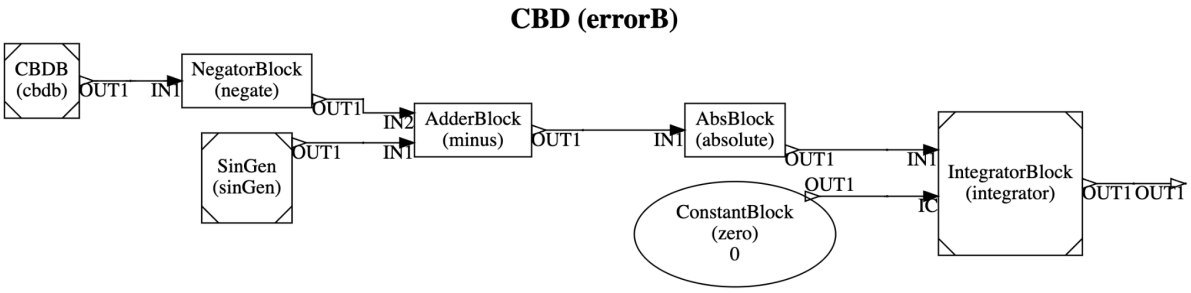
- TypeError: _cross_product_fncs() missing 2 required positional arguments: 'l1' and 'l2'

When trying to make the validity check for ErrorA we got this error. We weren’t able to solve this and suspect this is due to a bug in the library. Something additional is that we already get an error with the validity check of the CBDA, so it would also be very strange if we could run this. Unlike the CBDA and CBDB where we also got an error, but still got some output, here we don't get any output.

ErrorB

Model

In the image below you can see how we found the error between our CBDA and the real sinus function.



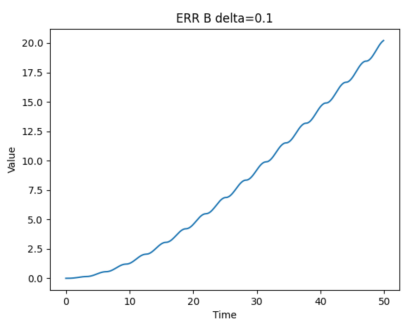
This is of course exactly the same as the ErrorA model explained above, but here we change the equation such that we calculate the error of sin(t) - Xb instead of Xa.

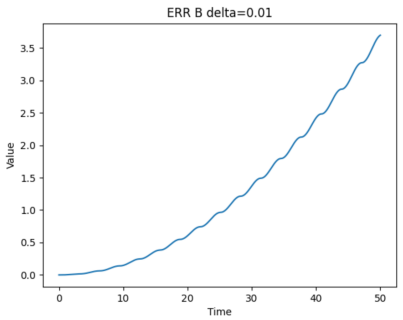
Plot

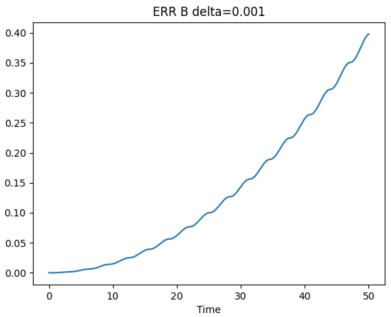
Error B is once again very similar to the ErrorA, except this is a coupled CBD that calculates the error between the output of CBDB (Xb) and the output of our Sinus Generator (Sin(t)). When we run this CBD for 50 seconds with

- Δt = 0.1
- Δt = 0.01
- Δt = 0.001

we got the following plots:







If we compare these values with the ErrorA values, we can immediately see that the error here is much lower. At Δt = 0.1 we had an error of 110 at ErrorA after 50 steps, while here we only are looking at 20. This trend also continues to an extent at Δt = 0.01. If we look at the plot of CBDB, this is actually not that surprising. CBDB starts well mimicking the sine function, but converges to a smaller value. However, it remains between 1 and -1, whilst CBDA on the other hand goes far above it. So the maximum error we can have is 1 (because we still follow the same period as the sinus function).

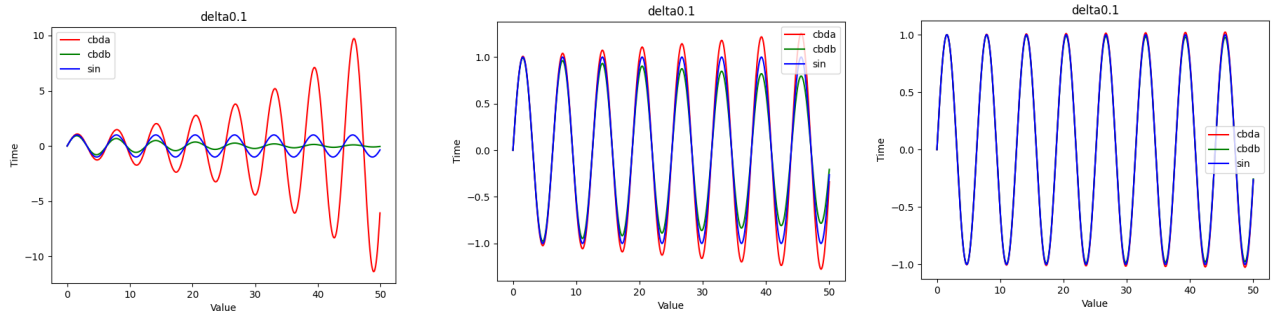
Here it is the same concept as with ErrorA: the lower the delta goes, the lower the error becomes.

Validity

Here again we take the same assumption we took with the errorA validity check. We also are not able to generate any sort of output.

Integrator vs Derivator

To get a good idea of why the error of CBDA is so much higher than that of CBDB and also why the derivator results in a better system than the integrator, we have plotted the Sin, CBDA and CBDB in 1 graph:



The blue line is the sinus function, the red line is CBDA and the green line is CBDB. You can clearly see that the CBDB line stays much better in range of the sinus function than that of the CBDA which gets way out of proportion.

RKF45

We implemented a converter with RKF45, but sadly we ran into a bug we could not solve. Due to this we were not able to do anything with the RKF45 CBD. The error we got was:

- AssertionError: assert itg.getBlockType() == "IntegratorBlock".

The code of the RKF45 is available on our github and in the provided source code in file “util.py”

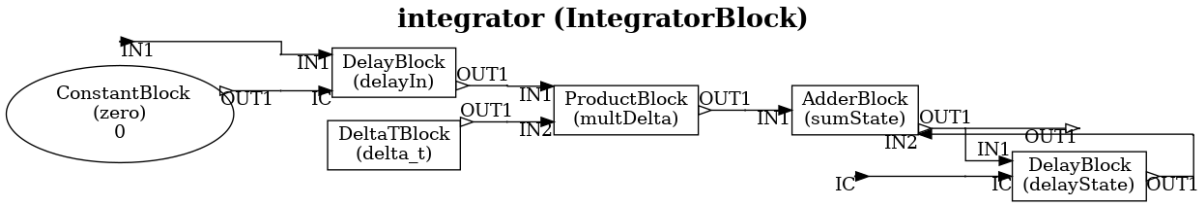
Integration Methods

We include plots of the integral of g(t), but we included this in the zip-file and in github.

Backward Euler Method

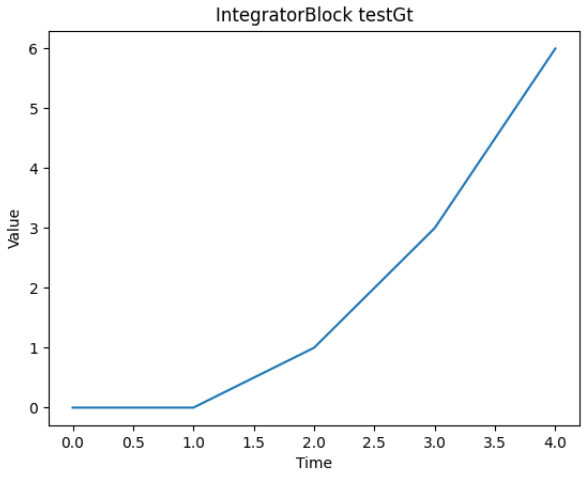
To implement the different integrators we started from the given code of the Backward Euler Method and adapted it. As a check that our implementation works as intended we ran a simulation that tested all the integration methods for 4 seconds with a delta of 1. This resulted in a small graph with easy numbers. So that we could manually check if it plotted what we intended.

Model



Test graph

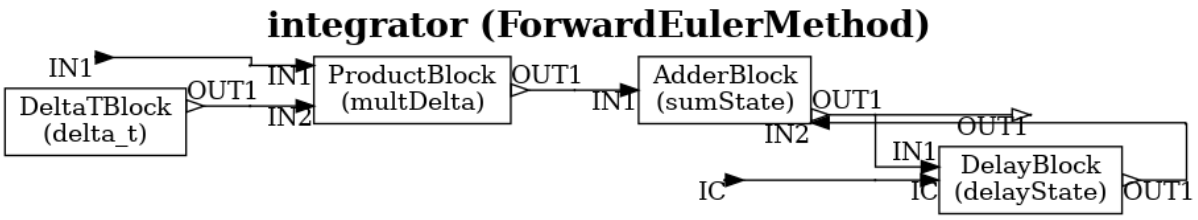
What we see here is that the angle of the graph increases with 1 every second. At time 4 we increase the value (0+1+2=3) with minus one the current time (3). So the total value at time 4 is 0+1+2+3=6.



Forward Euler Method

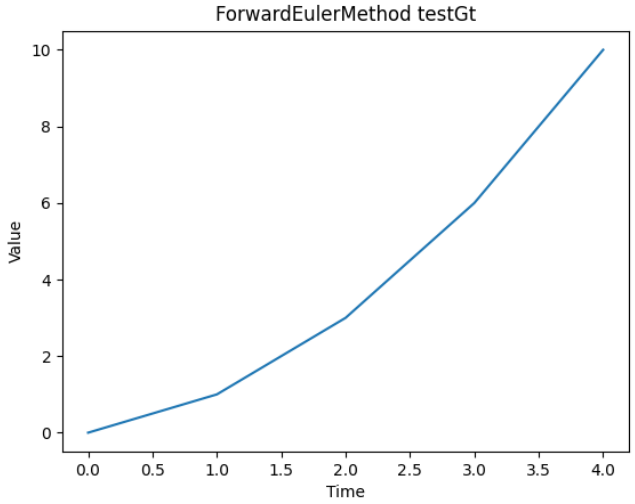
This method only differs on one point with the Backward Euler Method. With the Backward Euler Method we needed to use a delay block to get the input from the previous iteration to calculate the rectangle. Now we just use the current input. Practically this means that we just need to delete this delay block. You can see this when comparing the CBD images.

Model



Test graph

Similar to the Backward Euler Method we increase the value every second, but now with the time instead of the time minus 1. SO at time 4 we have 1+2+3+4 = 10. In other words the angle of the graph increases in the same way as the Backward Euler Method but now starts with 1 instead of 0.



Validity

INITIAL SYSTEM:
delta_t.OUT1(i) = i
multDelta.OUT1(i) = (multDelta.IN1(i) * multDelta.IN2(i))
delayState.OUT1(0) = delayState.IC(0)
delayState.OUT1(i) = delayState.IN1(i - 1 * i)
sumState.OUT1(i) = (sumState.IN1(i) + sumState.IN2(i))
multDelta.IN1(i) = IN1(i)
multDelta.IN2(i) = delta_t.OUT1(i)
delayState.IN1(i) = sumState.OUT1(i)
delayState.IC(i) = IC(i)
sumState.IN1(i) = multDelta.OUT1(i)
sumState.IN2(i) = delayState.OUT1(i)
OUT1(i) = sumState.OUT1(i)

STEP 1: substituted all connections and constant values
multDelta.OUT1(i) = (IN1(i) * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = sumState.OUT1(i - 1 * i)
sumState.OUT1(i) = (multDelta.OUT1(i) + delayState.OUT1(i))
OUT1(i) = sumState.OUT1(i)

STEP 2:
multDelta.OUT1(i) = (IN1(i) * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = (multDelta.OUT1(i - 1 * i) + delayState.OUT1(i - 1 * i))
OUT1(i) = (multDelta.OUT1(i) + delayState.OUT1(i))

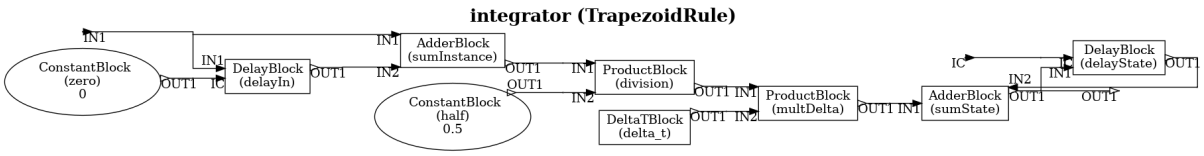
STEP 3:
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = ((IN1(i - 1 * i) * i) + delayState.OUT1(i - 1 * i))
OUT1(i) = ((IN1(i) * i) + delayState.OUT1(i))

RESULT IS:
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = ((IN1(i - 1 * i) * i) + delayState.OUT1(i - 1 * i))
OUT1(i) = ((IN1(i) * i) + delayState.OUT1(i))

Trapezoid Rule

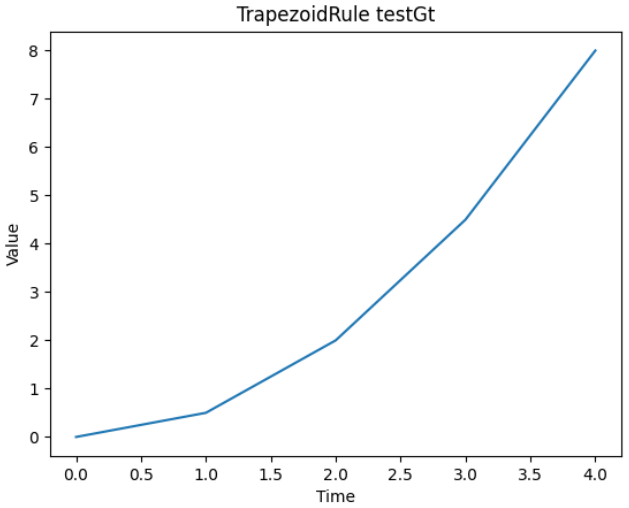
This method combines both the Backward and Forward Euler Method. So here we keep the delay block and use an Adderblock to sum both the original IN1 input and the input of the previous iteration which can be found in the output of the DelayBlock. The only other addition is the ProductBlock which multiplice this sum with a ConstantBlock 0.5.

Model



Test graph

Logically the graph is the average of both the Backward and Forward Euler Method. So at second 4 this is: (6+10)/2 = 8



Validity

INITIAL SYSTEM:
zero.OUT1(i) = 0
half.OUT1(i) = 0.5
delayIn.OUT1(0) = delayIn.IC(0)
delayIn.OUT1(i) = delayIn.IN1(i - 1 * i)
delta_t.OUT1(i) = i
multDelta.OUT1(i) = (multDelta.IN1(i) * multDelta.IN2(i))
division.OUT1(i) = (division.IN1(i) * division.IN2(i))
delayState.OUT1(0) = delayState.IC(0)
delayState.OUT1(i) = delayState.IN1(i - 1 * i)
sumState.OUT1(i) = (sumState.IN1(i) + sumState.IN2(i))
sumInstance.OUT1(i) = (sumInstance.IN1(i) + sumInstance.IN2(i))
delayIn.IN1(i) = IN1(i)
delayIn.IC(i) = zero.OUT1(i)
multDelta.IN1(i) = division.OUT1(i)
multDelta.IN2(i) = delta_t.OUT1(i)
division.IN1(i) = sumInstance.OUT1(i)
division.IN2(i) = half.OUT1(i)
delayState.IN1(i) = sumState.OUT1(i)
delayState.IC(i) = IC(i)
sumState.IN1(i) = multDelta.OUT1(i)
sumState.IN2(i) = delayState.OUT1(i)
sumInstance.IN1(i) = IN1(i)
sumInstance.IN2(i) = delayIn.OUT1(i)
OUT1(i) = sumState.OUT1(i)

STEP 1: substituted all connections and constant values
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
multDelta.OUT1(i) = (division.OUT1(i) * i)
division.OUT1(i) = (sumInstance.OUT1(i) * 0.5)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = sumState.OUT1(i - 1 * i)
sumState.OUT1(i) = (multDelta.OUT1(i) + delayState.OUT1(i))
sumInstance.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
OUT1(i) = sumState.OUT1(i)

STEP 2:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
multDelta.OUT1(i) = (division.OUT1(i) * i)
division.OUT1(i) = (sumInstance.OUT1(i) * 0.5)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = (multDelta.OUT1(i - 1 * i) + delayState.OUT1(i - 1 * i))
sumInstance.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
OUT1(i) = (multDelta.OUT1(i) + delayState.OUT1(i))

STEP 3:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = (((sumInstance.OUT1(i - 1 * i) * 0.5) * i) + delayState.OUT1(i - 1 * i))
sumInstance.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
OUT1(i) = (((sumInstance.OUT1(i) * 0.5) * i) + delayState.OUT1(i))

STEP 4:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = ((((IN1(i - 1 * i) + delayIn.OUT1(i - 1 * i)) * 0.5) * i) + delayState.OUT1(i - 1 * i))
OUT1(i) = ((((IN1(i) + delayIn.OUT1(i)) * 0.5) * i) + delayState.OUT1(i))

RESULT IS:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = ((((IN1(i - 1 * i) + delayIn.OUT1(i - 1 * i)) * 0.5) * i) + delayState.OUT1(i - 1 * i))
OUT1(i) = ((((IN1(i) + delayIn.OUT1(i)) * 0.5) * i) + delayState.OUT1(i))

Simpson's 1/3 Rule

This SImpsons method is the most complicated one, we will go over the main idea but not over all the different blocks. We can split the complexity of this method up in 4 parts: getting 3 time points every iteration, calculation I, Only calculate every other iteration, when there are only 2 points available use the Trapezoid Rule.

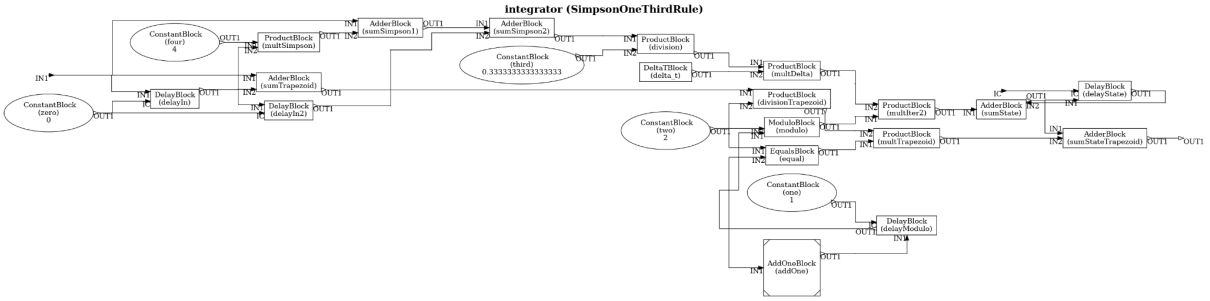
1. To get all 3 time points we simply added a delay block after the first delay block. This way we have the current time (IN1), the previous time (OUT1 of the first delay block), and the time before that (OUT1 of the second delay block).

2. Calculating the I, is just a couple of AdderBlocks and ProductBlocks connected with each other, using the previous inputs and ConstantBlocks.

3. To calculate every other iteration we used a delay block together with the AddOneBlock to count all the iterations. So that we could use a ModuloBlock with divider 2. This means that the moduloblock alternates its output each iteration with 0 and 1. So we multiply the output of the simpson ⅓ rule this iteration with the modulo. This means that it will either be 0, so nothing happens, or just the normal output.

4. We can also use this combination of the DelayBlock and AddOneBlock to see if this is the second time point. We can do this by using an EqualBlock and a ConstantBlock of 2. We then calculated the trapezoid rule and multiplied it with the output of the EqualBlock. Because the normal output will definitely be 0 because of the previous point, we can just add this to the solution. Important to note that we did not add this to the big summation as this is not part of the Simpson's ⅓ Rule and therefore should be dismissed in the next iterations.

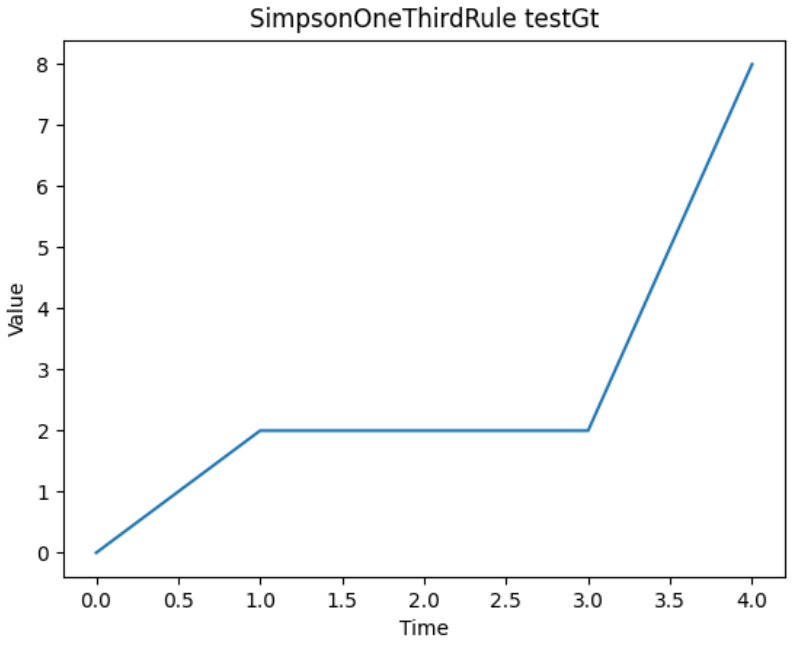
Model



A more clearer version of this model where you can zoom in with better quality can be found in the resource map.

Test graph

In the graph we see at second 1 that the output of the trapezoid rule is applied. At second 2 we should have (0+4*1+2)/3=2. This is the case and it also shows that the previous output of the Trapezoid rule has no impact on the following results. In the graph we also see that at second 3 nothing happens, which indicates that the “computed over 2 iterations” was also implemented correctly.



Validity

INITIAL SYSTEM:
zero.OUT1(i) = 0
third.OUT1(i) = 0.3333333333333333
one.OUT1(i) = 1
two.OUT1(i) = 2
four.OUT1(i) = 4
delayIn.OUT1(0) = delayIn.IC(0)
delayIn.OUT1(i) = delayIn.IN1(i - 1 * i)
delayIn2.OUT1(0) = delayIn2.IC(0)
delayIn2.OUT1(i) = delayIn2.IN1(i - 1 * i)
delayState.OUT1(0) = delayState.IC(0)
delayState.OUT1(i) = delayState.IN1(i - 1 * i)
delayModulo.OUT1(0) = delayModulo.IC(0)
delayModulo.OUT1(i) = delayModulo.IN1(i - 1 * i)
delta_t.OUT1(i) = i
multDelta.OUT1(i) = (multDelta.IN1(i) * multDelta.IN2(i))
multSimpson.OUT1(i) = (multSimpson.IN1(i) * multSimpson.IN2(i))
multIter2.OUT1(i) = (multIter2.IN1(i) * multIter2.IN2(i))
multTrapezoid.OUT1(i) = (multTrapezoid.IN1(i) * multTrapezoid.IN2(i))
division.OUT1(i) = (division.IN1(i) * division.IN2(i))
divisionTrapezoid.OUT1(i) = (divisionTrapezoid.IN1(i) * divisionTrapezoid.IN2(i))
sumState.OUT1(i) = (sumState.IN1(i) + sumState.IN2(i))
sumSimpson1.OUT1(i) = (sumSimpson1.IN1(i) + sumSimpson1.IN2(i))
sumSimpson2.OUT1(i) = (sumSimpson2.IN1(i) + sumSimpson2.IN2(i))
sumTrapezoid.OUT1(i) = (sumTrapezoid.IN1(i) + sumTrapezoid.IN2(i))
sumStateTrapezoid.OUT1(i) = (sumStateTrapezoid.IN1(i) + sumStateTrapezoid.IN2(i))
addOne.OUT1(i) = addone(addOne.IN1(i))
modulo.OUT1(i) = (modulo.IN1(i) mod modulo.IN2(i))
equal.OUT1(i) = (==)
delayIn.IN1(i) = IN1(i)
delayIn.IC(i) = zero.OUT1(i)
delayIn2.IN1(i) = delayIn.OUT1(i)
delayIn2.IC(i) = zero.OUT1(i)
delayState.IN1(i) = sumState.OUT1(i)
delayState.IC(i) = IC(i)
delayModulo.IN1(i) = addOne.OUT1(i)
delayModulo.IC(i) = one.OUT1(i)
multDelta.IN1(i) = division.OUT1(i)
multDelta.IN2(i) = delta_t.OUT1(i)
multSimpson.IN1(i) = four.OUT1(i)
multSimpson.IN2(i) = delayIn.OUT1(i)
multIter2.IN1(i) = modulo.OUT1(i)
multIter2.IN2(i) = multDelta.OUT1(i)
multTrapezoid.IN1(i) = equal.OUT1(i)
multTrapezoid.IN2(i) = divisionTrapezoid.OUT1(i)
division.IN1(i) = sumSimpson2.OUT1(i)

division.IN2(i) = third.OUT1(i)
divisionTrapezoid.IN1(i) = sumTrapezoid.OUT1(i)
divisionTrapezoid.IN2(i) = two.OUT1(i)
sumState.IN1(i) = multIter2.OUT1(i)
sumState.IN2(i) = delayState.OUT1(i)
sumSimpson1.IN1(i) = IN1(i)
sumSimpson1.IN2(i) = multSimpson.OUT1(i)
sumSimpson2.IN1(i) = sumSimpson1.OUT1(i)
sumSimpson2.IN2(i) = delayIn2.OUT1(i)
sumTrapezoid.IN1(i) = IN1(i)
sumTrapezoid.IN2(i) = delayIn.OUT1(i)
sumStateTrapezoid.IN1(i) = sumState.OUT1(i)
sumStateTrapezoid.IN2(i) = multTrapezoid.OUT1(i)
addOne.IN1(i) = delayModulo.OUT1(i)
modulo.IN1(i) = delayModulo.OUT1(i)
modulo.IN2(i) = two.OUT1(i)
equal.IN1(i) = two.OUT1(i)
equal.IN2(i) = delayModulo.OUT1(i)
OUT1(i) = sumStateTrapezoid.OUT1(i)

STEP 1: substituted all connections and constant values
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = sumState.OUT1(i - 1 * i)
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addOne.OUT1(i - 1 * i)
multDelta.OUT1(i) = (division.OUT1(i) * i)
multSimpson.OUT1(i) = (4 * delayIn.OUT1(i))
multIter2.OUT1(i) = (modulo.OUT1(i) * multDelta.OUT1(i))
multTrapezoid.OUT1(i) = (equal.OUT1(i) * divisionTrapezoid.OUT1(i))
division.OUT1(i) = (sumSimpson2.OUT1(i) * 0.3333333333333333)
divisionTrapezoid.OUT1(i) = (sumTrapezoid.OUT1(i) * 2)
sumState.OUT1(i) = (multIter2.OUT1(i) + delayState.OUT1(i))
sumSimpson1.OUT1(i) = (IN1(i) + multSimpson.OUT1(i))
sumSimpson2.OUT1(i) = (sumSimpson1.OUT1(i) + delayIn2.OUT1(i))
sumTrapezoid.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
sumStateTrapezoid.OUT1(i) = (sumState.OUT1(i) + multTrapezoid.OUT1(i))
addOne.OUT1(i) = addone(delayModulo.OUT1(i))
modulo.OUT1(i) = (delayModulo.OUT1(i) mod 2)
equal.OUT1(i) = (==)
OUT1(i) = sumStateTrapezoid.OUT1(i)

STEP 2:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)

delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = (multIter2.OUT1(i - 1 * i) + delayState.OUT1(i - 1 * i))
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addone(delayModulo.OUT1(i - 1 * i))
multDelta.OUT1(i) = (division.OUT1(i) * i)
multSimpson.OUT1(i) = (delayIn.OUT1(i) * 4.0)
multIter2.OUT1(i) = (modulo.OUT1(i) * multDelta.OUT1(i))
multTrapezoid.OUT1(i) = (equal.OUT1(i) * divisionTrapezoid.OUT1(i))
division.OUT1(i) = (sumSimpson2.OUT1(i) * 0.3333333333333333)
divisionTrapezoid.OUT1(i) = (sumTrapezoid.OUT1(i) * 2.0)
sumState.OUT1(i) = (multIter2.OUT1(i) + delayState.OUT1(i))
sumSimpson1.OUT1(i) = (IN1(i) + multSimpson.OUT1(i))
sumSimpson2.OUT1(i) = (sumSimpson1.OUT1(i) + delayIn2.OUT1(i))
sumTrapezoid.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
modulo.OUT1(i) = (delayModulo.OUT1(i) mod 2)
equal.OUT1(i) = (==)
OUT1(i) = (sumState.OUT1(i) + multTrapezoid.OUT1(i))

STEP 3:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = ((modulo.OUT1(i - 1 * i) * multDelta.OUT1(i - 1 * i)) + delayState.OUT1(i - 1 * i))
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addone(delayModulo.OUT1(i - 1 * i))
multDelta.OUT1(i) = (division.OUT1(i) * i)
multSimpson.OUT1(i) = (delayIn.OUT1(i) * 4.0)
division.OUT1(i) = (sumSimpson2.OUT1(i) * 0.3333333333333333)
divisionTrapezoid.OUT1(i) = (sumTrapezoid.OUT1(i) * 2.0)
sumSimpson1.OUT1(i) = (IN1(i) + multSimpson.OUT1(i))
sumSimpson2.OUT1(i) = (sumSimpson1.OUT1(i) + delayIn2.OUT1(i))
sumTrapezoid.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
modulo.OUT1(i) = (delayModulo.OUT1(i) mod 2)
equal.OUT1(i) = (==)
OUT1(i) = (((modulo.OUT1(i) * multDelta.OUT1(i)) + delayState.OUT1(i)) + (equal.OUT1(i) * divisionTrapezoid.OUT1(i)))

STEP 4:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)

delayState.OUT1(i) = (((delayModulo.OUT1(i - 1 * i) mod 2) * ((sumSimpson2.OUT1(i - 1 * i) * 0.3333333333333333) * i)) + delayState.OUT1(i - 1 * i))
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addone(delayModulo.OUT1(i - 1 * i))
multSimpson.OUT1(i) = (delayIn.OUT1(i) * 4.0)
sumSimpson1.OUT1(i) = (IN1(i) + multSimpson.OUT1(i))
sumSimpson2.OUT1(i) = (sumSimpson1.OUT1(i) + delayIn2.OUT1(i))
sumTrapezoid.OUT1(i) = (IN1(i) + delayIn.OUT1(i))
OUT1(i) = ((((delayModulo.OUT1(i) mod 2) * ((sumSimpson2.OUT1(i) * 0.3333333333333333) * i)) + delayState.OUT1(i)) + ((==) * (sumTrapezoid.OUT1(i) * 2.0)))

STEP 5:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = (((delayModulo.OUT1(i - 1 * i) mod 2) * ((((IN1(i - 1 * i) + multSimpson.OUT1(i - 1 * i)) + delayIn2.OUT1(i - 1 * i)) * 0.3333333333333333) * i)) + delayState.OUT1(i - 1 * i))
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addone(delayModulo.OUT1(i - 1 * i))
multSimpson.OUT1(i) = (delayIn.OUT1(i) * 4.0)
OUT1(i) = ((((delayModulo.OUT1(i) mod 2) * ((((IN1(i) + multSimpson.OUT1(i)) + delayIn2.OUT1(i)) * 0.3333333333333333) * i)) + delayState.OUT1(i)) + ((==) * ((IN1(i) + delayIn.OUT1(i)) * 2.0)))

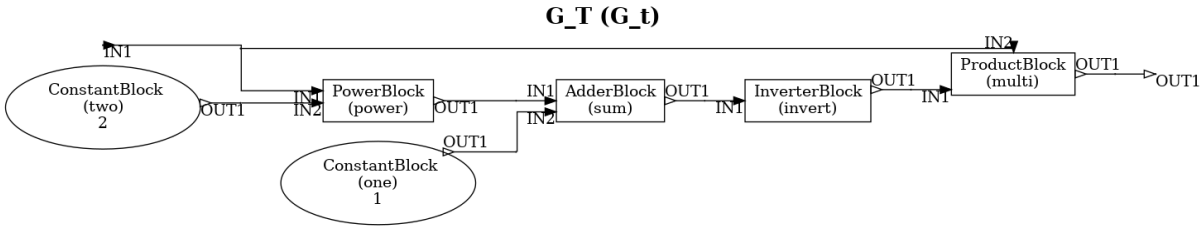
STEP 6:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)
delayState.OUT1(i) = (((delayModulo.OUT1(i - 1 * i) mod 2) * ((((IN1(i - 1 * i) + (delayIn.OUT1(i - 1 * i) * 4.0)) + delayIn2.OUT1(i - 1 * i)) * 0.3333333333333333) * i)) + delayState.OUT1(i - 1 * i))
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addone(delayModulo.OUT1(i - 1 * i))
OUT1(i) = ((((delayModulo.OUT1(i) mod 2) * ((((IN1(i) + (delayIn.OUT1(i) * 4.0)) + delayIn2.OUT1(i)) * 0.3333333333333333) * i)) + delayState.OUT1(i)) + ((==) * ((IN1(i) + delayIn.OUT1(i)) * 2.0)))

RESULT IS:
delayIn.OUT1(0) = 0
delayIn.OUT1(i) = IN1(i - 1 * i)
delayIn2.OUT1(0) = 0
delayIn2.OUT1(i) = delayIn.OUT1(i - 1 * i)
delayState.OUT1(0) = IC(0)

delayState.OUT1(i) = (((delayModulo.OUT1(i - 1 * i) mod 2) * ((((IN1(i - 1 * i) + (delayIn.OUT1(i - 1 * i) * 4.0)) + delayIn2.OUT1(i - 1 * i)) * 0.3333333333333333) * i)) + delayState.OUT1(i - 1 * i))
delayModulo.OUT1(0) = 1
delayModulo.OUT1(i) = addone(delayModulo.OUT1(i - 1 * i))
OUT1(i) = ((((delayModulo.OUT1(i) mod 2) * ((((IN1(i) + (delayIn.OUT1(i) * 4.0)) + delayIn2.OUT1(i)) * 0.3333333333333333) * i)) + delayState.OUT1(i)) + ((==) * ((IN1(i) + delayIn.OUT1(i)) * 2.0)))

Function g(t)

This is a rather simple implementation of Product/Adder/Power/Inverter/ConstantBlocks. The result is visible in the following image.



Integral of g(x)

In the following table all the results will be outputted, the analytic solution is 4.60522018.

	Delta=0.1	Delta=0.01	Delta=0.001
Backward EM	4.60388597	4.60506185	4.60520510
Forward EM	4.60488587	4.60516185	4.60521510
Trapezoid	4.60438592	4.60511185	4.60521010
Simpson’s ⅓	4.60522359	4.60502018	4.60520018

Comparison

It is no surprise that the lower the delta, the higher the accuracy is. This means more data points and smaller steps. The span of the step size is the estimation, so an infinitely small step size will result in the right analytic solution. So when we make the step size smaller we get closer to this.
The Forward EM is bigger than EM, this is to be suspected with graphs that have an upwards evolution like this one. The Forward Euler Method uses data points that are one time point further away. So when the graph grows, like this one, it uses bigger numbers.
It is to be expected that the Trapezoid rule will lie between the Backward and Forward Euler Methods.
We see that the Simpson method gets it almost right with the 0.1 delta, but varies when we take a different step size. The lower stepsizes both have a 0 on the 4th or 5th decimal where there should be a 2. Otherwise this is a very good approximation. When taking delta 0.005 it gets it totally right. The only way we can explain that the output is not getting better, is that either our implementation is somewhere not 100% right or there is a bug in the output and those 0’s should in fact be a 2. Simpson works by estimating the curve. It does this by giving a big weight on the middle point and giving smaller weights to start and end points. So the Closer they are, the better the estimation should be.

The Simpson’s ⅓ rule is definitely the most complicated and slowest method, but also the most accurate. The lower the delta the better the result should be.

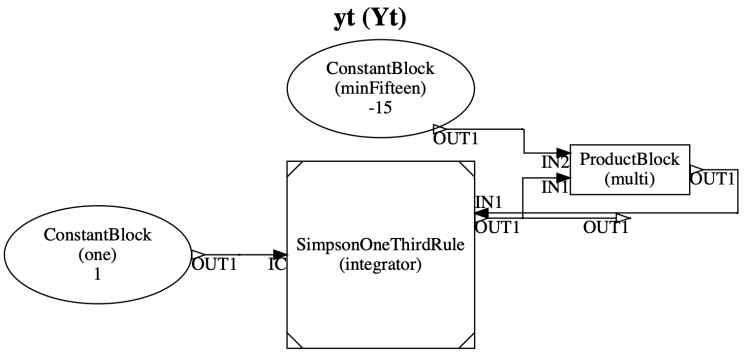
Stiff Systems of Equations

Model

The idea behind the model lies with the following statement:
The integral of the derivative of y(t) is y(t). So we can rewrite the assignment as follows:

- y'(t) = -15y
- integral(y'(t)) = integral(-15y)
- y(t) = integral(-15y)

(the added constants that come with primitives can be ignored for this purpose.)
So we just need to take the integral of "t" times -15. Which is described in the following model.



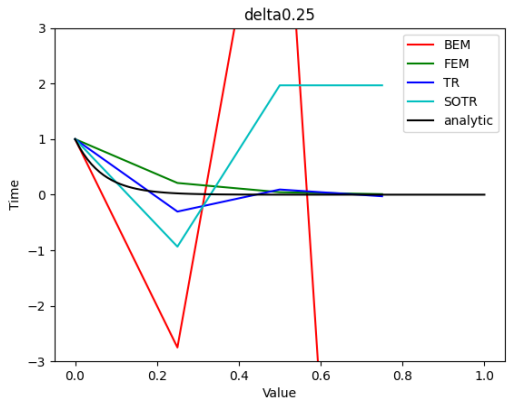
We only made the model for 1 integral method, as it is the same model with just other Integrator blocks.

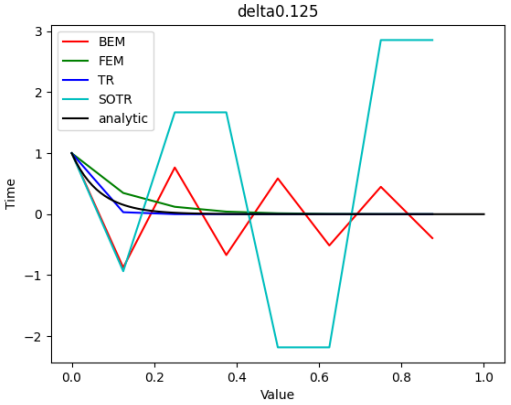
Plot

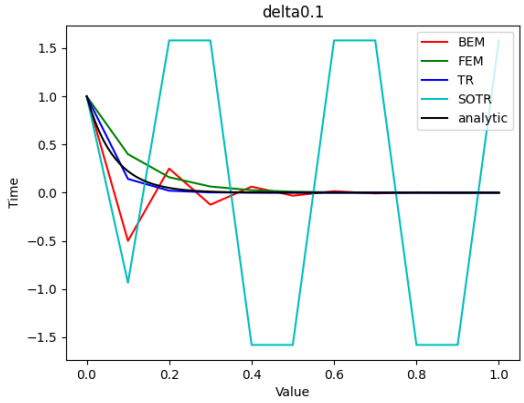
Below we simulated each of the 4 integrator methods and compared them to the real analytical solution (which can be seen in black) for every delta. In the provided ZIP-file you can find all the plots separately. But in an effort to keep the rapport from being cluttered we combined these plots for every delta.

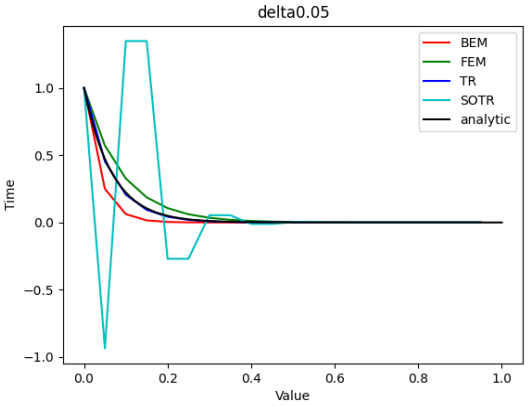
For clarification:

- BEM = Backward Euler Method
- FEM = Forward Euler Method
- TR = Trapezoid Rule
- SOTR = Simpson's ⅓ Rule









The best approximation was done with the Trapezoid rule. We suspect that this is because it follows the angle of the graph. The closer the datapoint the closer to the real graph we become. This idea is supported by the plots above. Because Simpson's ⅓ rule gives a high weight to one point and we work with rather unstable equations, it produces a bad output.

The Backwards Euler Method uses a datapoint to estimate the following stepsize part of the graph. So it uses the earlier point to estimate the following part. The Forward Euler method does the opposite, it uses a datapoint to estimate the previous step size part of the graph. So we would expect that the Backwards Euler Method has more variation because it uses bigger values. We know this because the graph is downwards. So when using earlier points to estimate later parts, you use bigger values.