Project 2
Molecular Programming Language
CRN++

Roar Nind Steffensen (s144107)
Samy Haffoudhi (s222887)
Valeriu Seremet (s230096)

June 22, 2023

Abstract

This report describes our work for the CRN++ molecular programming language project. This project consists in writing an interpreter for the CRN++ language described in [1]. We first analyse the description of CRN++ in the article and study the grammar and syntactic restrictions. We then implement the parser using *FParsec*, next the type checker and the interpreter and finally we construct a simulator for chemical reactions. We also implemented AST visualisation and reaction graph visualisation using *Plotly.NET*. Finally we conclude with an evaluation and performance optimisation of the simulator as well as out final thoughts about the project. In the appendix there are extra error and runtime comparison plots as well as simulations of all program examples from the reference article.

In this report we are going to implement the molecular programming language termed *CRN++* based on the article *CRN++: Molecular programming language*[1]. The goal of this programming language is to provide a high level of abstraction for composing arithmetic modules which are simulated as chemical reaction networks (CRN). With such a tool it is possible to develop a set of chemical reactions which can compute a given program, before using time and resources to synthesise the CRN in the laboratory. The implementation is developed using *F#*, with the frameworks *FParsec* for parsing, *FsCheck* for testing and *Plotly.NET* for visualisations.

The grammar provided by the article[1] will be presented as well as a revised to a LL(1) grammar. Based on this, we present the abstract syntax tree (AST) of the language with a parser and non-crn interpreter as baseline. A CRN simulator is then implemented and evaluated wrt. correctness.

1 The CRN++ Language

1.1 Presented CRN grammar

The key goal of the CRN++ language is to embed familiar imperative programming concepts into a set of chemical reactions, allowing for a convenient higher level for abstractions when working with molecular computation.

As such, the proposed grammar for this language is presented in Listing 1.1 in the article [1]. On a high level, a CRN++ program consists in a sequence of *Conc* and *Step* statements. Where *Cocn* statements define initial concentrations of species and where each *Step* statement contains a list of commands (explicit reactions, arithmetic operations or conditionals) to be run in parallel.

1.2 Revised grammar

The proposed grammar is not an LL(1) grammar. Indeed, as, for example, shown by the production rule $\langle RootSList \rangle \coloneqq \langle RootS \rangle \mid \langle RootS \rangle ', ' \langle RootSList \rangle$, given in Listing 1.1 in the article [1], it cannot be inferred which branch to parse next by only looking at the next input symbol, thus making the grammar not LL(1).

However, being easy to parse, having an LL(1) grammar is of great interest. Fortunately, the proposed grammar can easily be revised into an LL(1) grammar by modifying the production rules for the *RootSList* and *CommandSList* non-terminal symbols:

$$\begin{aligned}
\langle RootSList \rangle &\coloneqq \langle RootS \rangle \, \langle RootSList' \rangle \\
\langle RootSList' \rangle &\coloneqq \text{`,'} \, \langle RootS \rangle \, \langle RootSList' \rangle \mid \epsilon \\
\langle CommandSList \rangle &\coloneqq \langle CommandS \rangle \, \langle CommandSList' \rangle \\
\langle CommandSList' \rangle &\coloneqq \text{`,'} \, \langle CommandS \rangle \, \langle CommandSList' \rangle \mid \epsilon
\end{aligned}$$

In addition, we noticed some differences between the grammar rules and example code snippets in the article [1] (lack of curly brackets and semicolon in the grammar), for these inconsistencies, we decided to follow the rules given in Listing 1.1.

1.3 Additional syntactic restrictions

Other precisely defined syntactic restrictions that are not entrusted by the grammar must be satisfied by CRN++ programs to be valid.

Firstly, commands within a step, which can be executed in parallel, cannot contain cyclic dependencies, which means that the output from one module is an input to another and vice versa. The example given in the article [1] of the presence of both an `add[c, d, a]` and an `mul[a, b, c]` modules in a same step is such a case of cyclic dependency. A special case of this restriction is the fact that a species cannot be both input and output to a module.

Another observed restriction of the CRN++ language, is that input species for a module should be defined either by a *conc* statement or as an output species of a previous module. These syntactic restrictions, together with the revised grammar, describe the requirements for well-formed CRN++ programs.

1.4 Abstract syntax tree

Our abstract syntax tree model for CRN++ programs consists in the F# types declarations seen in listing 1.

```
type species = S of string
type Command =
    | Load of species * species
    | Rxn of species list * species list * float
    // Arithmetic
    | Add of species * species * species
    | Sub of species * species * species
    | Mul of species * species * species
    | Div of species * species * species
    | Sqrt of species * species
    // Conditionnals
    | Cmp of species * species
    | Ifgt of Command list
    | Ifge of Command list
    | Iflt of Command list
    | Ifle of Command list
    | Ifeq of Command list
type Root =
     | Conc of species * float
     | Step of Command list
type Crn = Root list
```

Listing 1: AST types

Using the code from project 1 we can visualise such ASTs, for example, figure 1 shows the AST for the example GCD program given in Figure 3.a in the article [1].

1.5 Language parsing

The parser for CRN++, visible in `Compiler/Parser.fs` was implemented using the FParsec library following our refined grammar rules, discussed in 1.2.

It is able to produce abstract syntax trees for the article example programs after they've been adjusted to the refined syntax presented in 1.3.

Property-based testing was used to validate the parser implementation using the FsCheck library. Relying on our implementation of a custom generator for generating well-formed

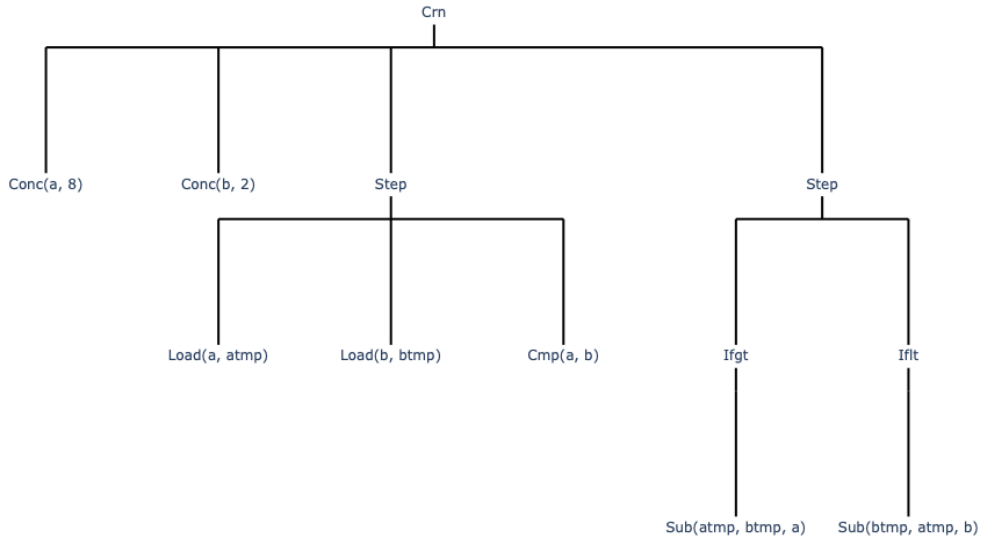


Figure 1: GCD Program AST.

CRN++ programs, the implemented property for the parser verifies that the string representation of an AST is parsed to the same AST, ensuring that the parser is consistent.

1.6 Type checker and basic interpreter

To verify that the parsed program complies with the syntax restrictions mentioned in section 1.3, a type checker is implemented. The type checker is visible in `Compiler/TypeChecker.fs` and implements the rules mentioned in 1.3.

During the execution of a CRN++ program, we are interested in keeping track of exactly two components:

- The state of the flag used for conditional blocks
- The concentration for each of the species

Thus, we define the `State` type as seen in listing 2.

```
type Flag = Greater | Equal | Less
type State = {
    variables : Map<string, float>
    flag: Flag
}
```

Listing 2: Basic interpreter state

Here the `Flag` type initially is set to `Equal` before the first call to `cmp` however the TypeChecker ensures that a call to `cmp` happens before any conditional block, setting the initial

signal values. This avoids having an invalid `Unset` state in the `Flag` type during all conditional evaluations.

Our interpreter implementation is visible in `Compiler/BasicInterpreter.fs`. Each module is evaluated using the corresponding mathematical operations, and the separation of steps is done explicitly by executing them in order. This interpreter serves as a baseline for the expected output of a crn program. Regarding property-based testing, no checks were implemented for ensuring that the order of execution does not matter in steps. In fact, for many CRN++ programs, order does have an impact on our interpreter output since modules are run consectively, one step at a time. It is for example the case with the step `step[ ld[a, b], ld[b, c] ]`, which despite not containing cycles, results in different species concentrations at a given step depending on the order the two load modules where run.

2 Chemical reactions

2.1 Reaction parser

A chemical reaction consists in reactants, products and a rate. Where reactants and products consist of species names, multiplicities and concentrations. We thus define the types for chemical reactions seen in listing 3.

```
type Name = string
type Multiplicity = int
type Concentration = float
type Rate = float
type Species = Name * Concentration
type ReactionComponent = Name * Multiplicity
type Reactants = ReactionComponent list
type Products = ReactionComponent list
type Reaction = Reactants * Rate * Products
type Solution = Map<Name, Species>
type CRN = Reaction list * Solution
```

Listing 3: Reaction parser types

To parse chemical reactions we can define this as a simplified grammar:

$$
\begin{aligned}
\langle ReactionS \rangle &:= \langle ComponentsS \rangle \ \langle RateS \rangle \ \text{`} \rightarrow \text{`} \ \langle ComponentsS \rangle \\
\langle RateS \rangle &:= \text{`(`} \ \langle number \rangle \ \text{`)`} \mid \epsilon \\
\langle ComponentsS \rangle &:= \langle species \rangle \ \langle ComponentsS' \rangle \mid \text{`}\emptyset\text{`} \\
\langle ComponentsS' \rangle &:= \text{`} + \text{`} \ \langle species \rangle \ \langle ComponentsS' \rangle \mid \epsilon
\end{aligned}
$$

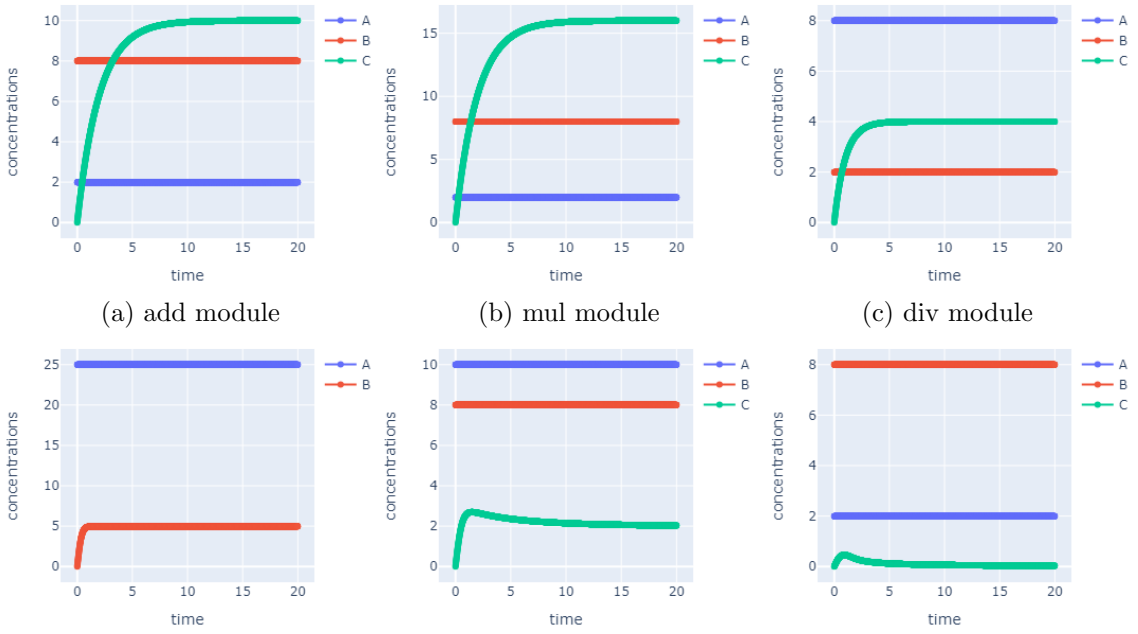
Where the terminal symbol $\langle species \rangle$ corresponds to a species name and $\langle number \rangle$ to a floating point number for its concentration. Specifying a rate is optional, the default value is set to a rate of 1.0, and concentrations are set to 0.0 unless set in a `conc` statement.

The reaction parser implementation is visible in `ChemicalEngine/Parser.fs`.

2.2 CRN reactions simulator

To simulate the reactions in a given CRN, the general reaction solution $\frac{dS}{dt}$ given by the article[1] is implemented and evaluated for all species/chemicals in the simulated CRN. Here it is necessary to choose the time resolution used for the discrete time intervals between simulation steps. If the time interval is too large, the simulation becomes unstable and will not converge to the expected output, and for small intervals, the simulation becomes too computationally intense. Here we settled on a time resolution of 0.01, such that we calculate 100 simulations per unit of time. This seems to be create stable outputs within a reasonable amount of time.

Each of the modules defined in the article[1] are implemented by the reactions provided. Example simulations can be seen in figure 2. Since the reactions operate on concentrations of species, it is not possible to evaluate negative values, and looking at the sub module: $A < B$ in figure 2f we see that it evaluates to 0.



(a) add module (b) mul module (c) div module

(d) sqrt module (e) sub module: $A > B$ (f) sub module: $A < B$

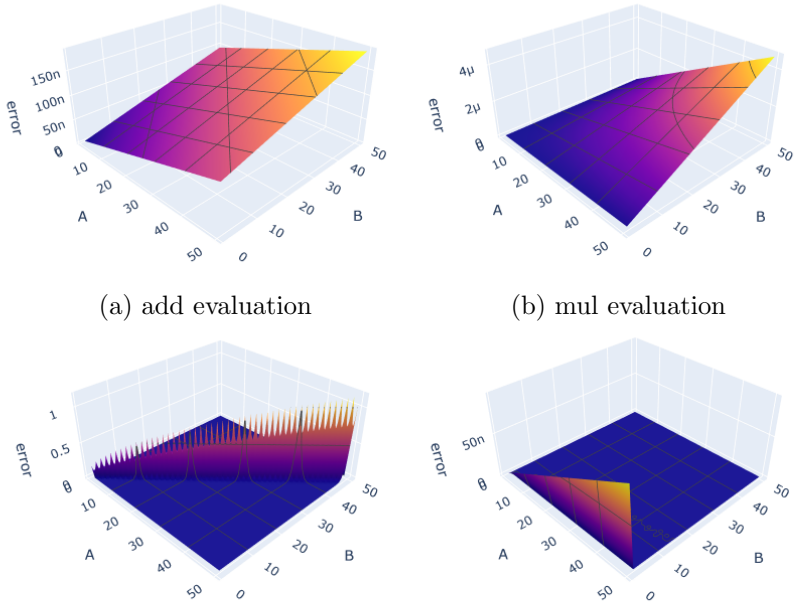
Figure 2: Arithmetic CRN module simulations.

The module reactions are implemented in `ChemicalEngine/Modules.fs` and simulated using `ChemicalEngine/Simulator.fs`.

2.3 Simulation evaluations

To test the correctness of the module simulations we replicated the error analysis plots of the article seen here in figure 3. Note that for the sub module, we had to restrict the domain to only include $A \geq B$, since for larger cases of $A < B$ the error grows a lot (see figure 4 in

appendix). In the article[1] the sub module errors is a known problem and they introduce techniques to minimise this error by the crn program structure. The output of each module is also tested with property based testing to verify that it calculates the expected result within small margin of error, which is generally true, however for larger inputs where A and B are close the sub module occasionally does not converge fast enough to pass. This growing error is visualised by the diagonal spike in figure 3c (also see figure 5 in appendix).



(a) add evaluation

(b) mul evaluation

(c) sub evaluation: $A \geq B$

(d) div evaluation

Figure 3: Simulation error for inputs A and B by comparing to expected output, replicating error results from the reference article[1].

Since our error analysis replicate the data shown in the article[1], together with the property-based tests for expected outputs, it gives strong evidence for the correctness of the module simulations. The *sqrt* module is also implemented and tested with property-based tests, however since it was not shown in the articles error analysis it is not shown here.

3 CRN compilations

By combining the list of reactions and the species we can combine multiple CRN's into one and simulate the compilation of reactions all in parallel. With the arithmatic modules defined in section 2.2, we now need to define the *cmp* module and the conditional branching modules: *ifGt*, *ifLt*, *ifEq*, *ifLe* and *ifGe*.

To implement these, we need sequential evaluation of steps, since *cmp* and the conditionals are not composable in the same step. For modules to be composable, they should not

deplete the reactants or have cyclic dependencies with products of the other modules. The sequential evaluation is implemented by clock species which then catalyse all reactions at offset phases, such that non-composable modules do not react simultaneously (or reacts minimally). The clock CRN from the article is implemented to do this, and phase species are added as catalysts to all reactions in each step. The article does not specify initial concentrations of the clock species, but consolidating their reference paper[2] shows that it creates a $p + 1$ fixed point oscillator for a p phase clock, where no signal should go to 0, since the clock would converge to a fixed point. By setting the initial concentrations of 2 contiguous phase-signals to almost 0.5 and the rest to almost 0 s.t. their accumulated concentration sums to 1, we get the clock as shown in the article[1], (see fig 6 in appendix).

With the clock implemented we can insert the 2 required phases of the *cmp* module to generate the comparison signals used to catalyse the conditional modules. For the equal-including conditions, the *cmp* module requires an ϵ offset to both of its inputs using the existing *add* modules with an ϵ value of 0.5 as mention in the article[1]. With these defined, the simulator now supports all the proposed modules. When the steps and modules have been catalysed by *clock* and *cmp* species, we combine the CRN's for each command in a step, and then for all steps to produce a single CRN, and this compilation is then simulated, relying only on the signals for ordering.

3.1 Evaluation of simulated compilations

Simulating all signals for the example programs (see figures 9 to 20 in appendix) easily becomes computationally intensive. To mitigate this, the simulation performance was improved by pre-calculating $k(rxn) \cdot netChange(S, rxn)$ for each specimen for all reactions, as well as using lookup arrays for concentrations used in the last product of the solution ODE from the article[1]. All reactions with a net change of 0 are removed for that specimen and by using index lookups we avoid the hash calculations of using maps. For the examples we generally see a 10-20x performance improvement, but more importantly, by not evaluating unrelated reactions it scales significantly better with larger CRN's with composable modules (see figures 7 and 8 in appendix).

All example programs from the article are replicated and seen in the appendices, and overall we are very satisfied with how our CRN++ interpreter turned out. Many implementation details were omitted in the report due to the given page limit.

References

[1] David Doty and Hendrik Dietz, eds. *DNA Computing and Molecular Programming*. Springer International Publishing, 2018. DOI: 10.1007/978-3-030-00030-1. URL: https://doi.org/10.1007%2F978-3-030-00030-1.

[2] Michael Lachmann and Guy Sella. "The computationally complete ant colony: Global coordination in a system with no hierarchy". In: *Advances in Artificial Life*. Ed. by Federico Morán et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 784–800. ISBN: 978-3-540-49286-3.

Appendix

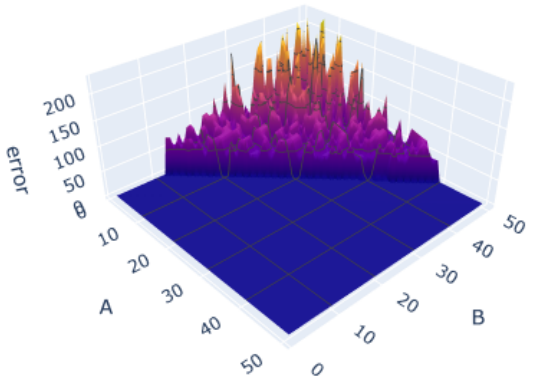
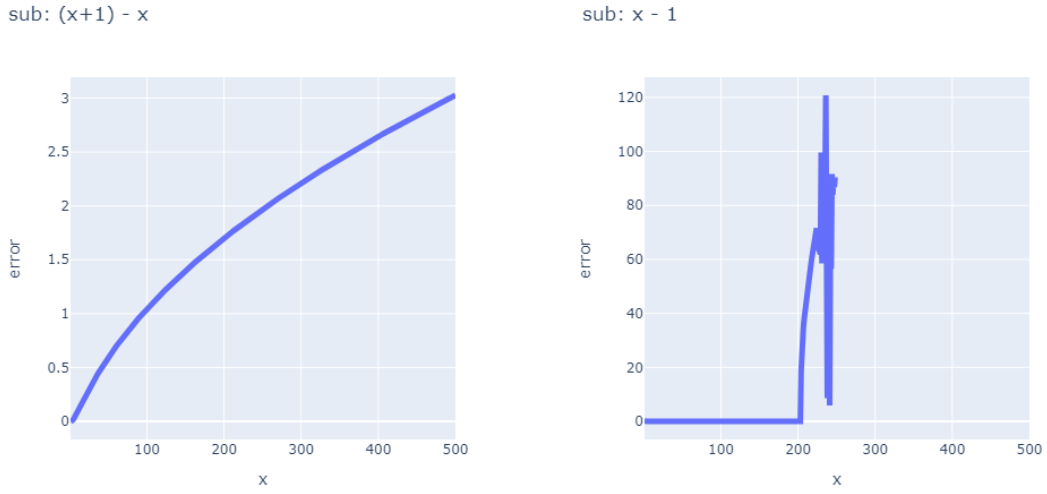


Figure 4: Evaluation of sub module, including domain $A < B$.



(a) Evaluation of sub module for $(x + 1) - x$.

(b) Evaluation of sub module for $x - 1$.

Figure 5: Showing growing error of sub module for larger inputs. Here it is clear that the $|A - B| \geq 1$ threshold stated by the article only applies for small input values, by the growing error seen in figure 5a. This is most likely caused by the convergence speed of the module for close inputs. In figure 5b for mitigating the sub error by subtracting 1 multiple times, we initially see that the error indeed is low initially, but suddenly spikes, which in this case is due to the time resolution of the simulation, causing the output to oscillate and not converge to the expected output.

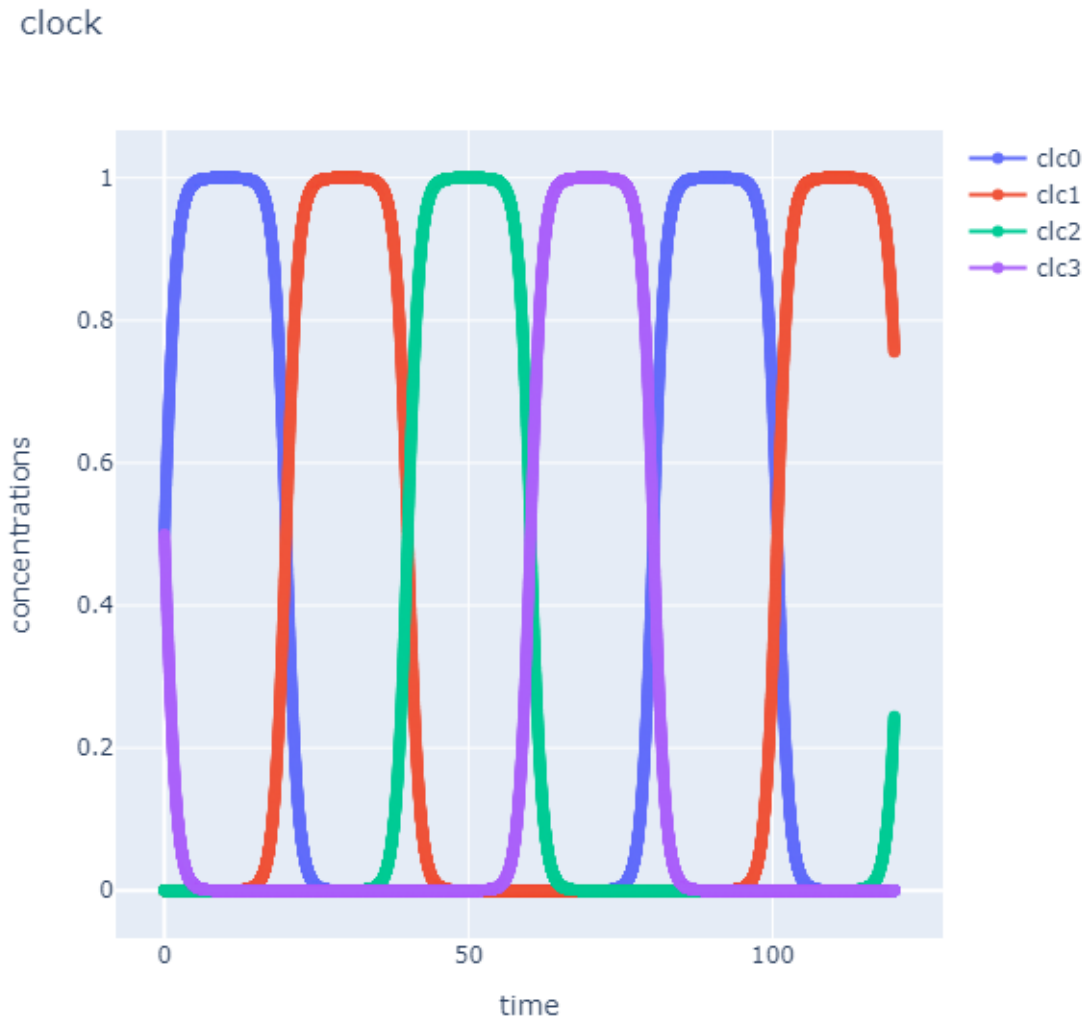


Figure 6: Simulation of 4 phase clock specimen oscillating.

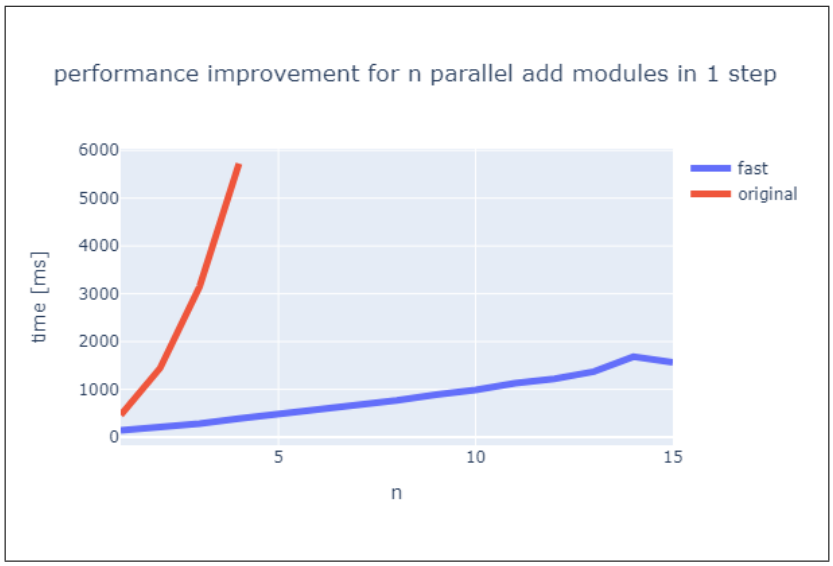


Figure 7: Performance comparison in simulating n parallel add modules, using the original and updated fast simulation described in 3.1. Each CRN compilation is simulated for 100.000 iterations equivalent to 1000 units of time.

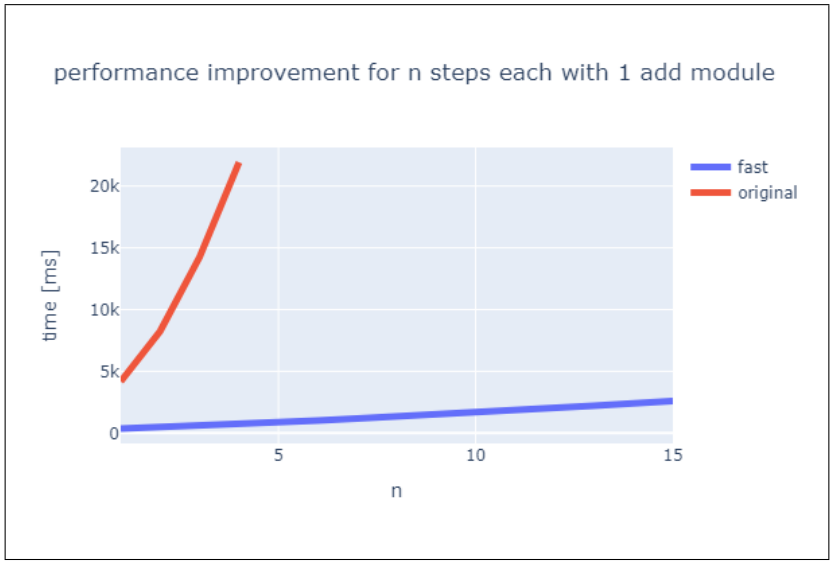


Figure 8: Performance comparison in simulating n steps each of 1 add module, using the original and updated fast simulation described in 3.1. Each CRN compilation is simulated for 100.000 iterations equivalent to 1000 units of time.

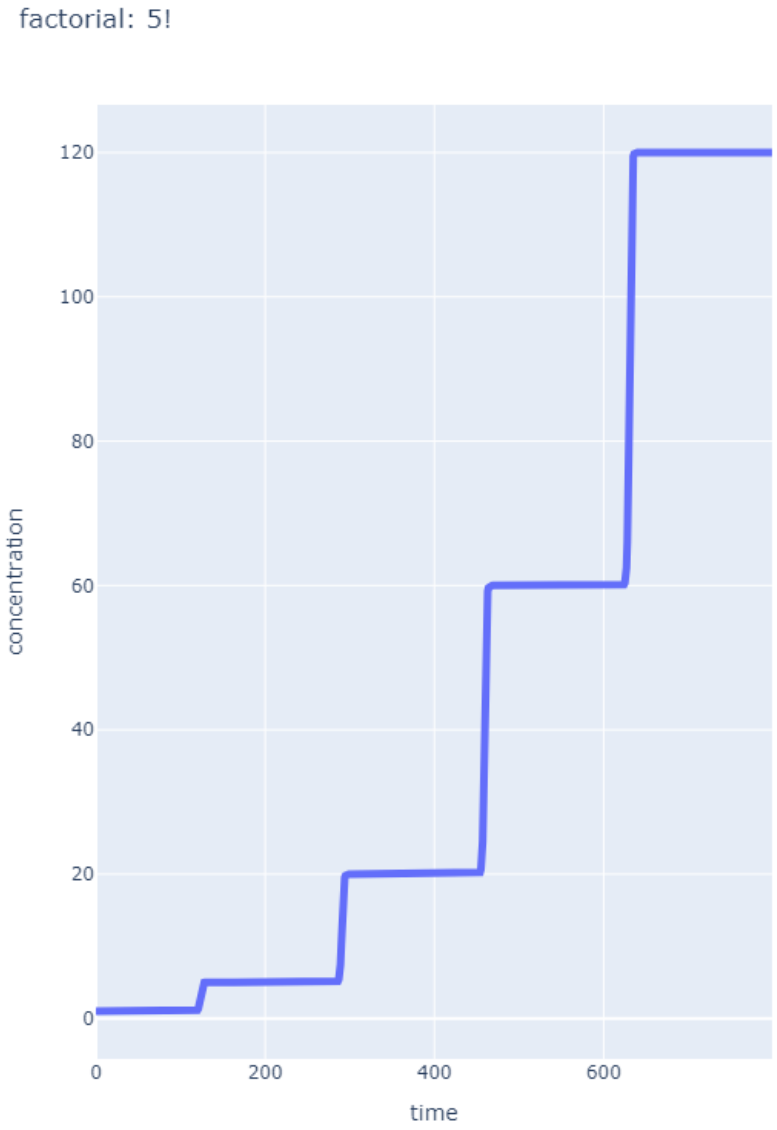


Figure 9: Simulation of 5 factorial program, only showing result signal.

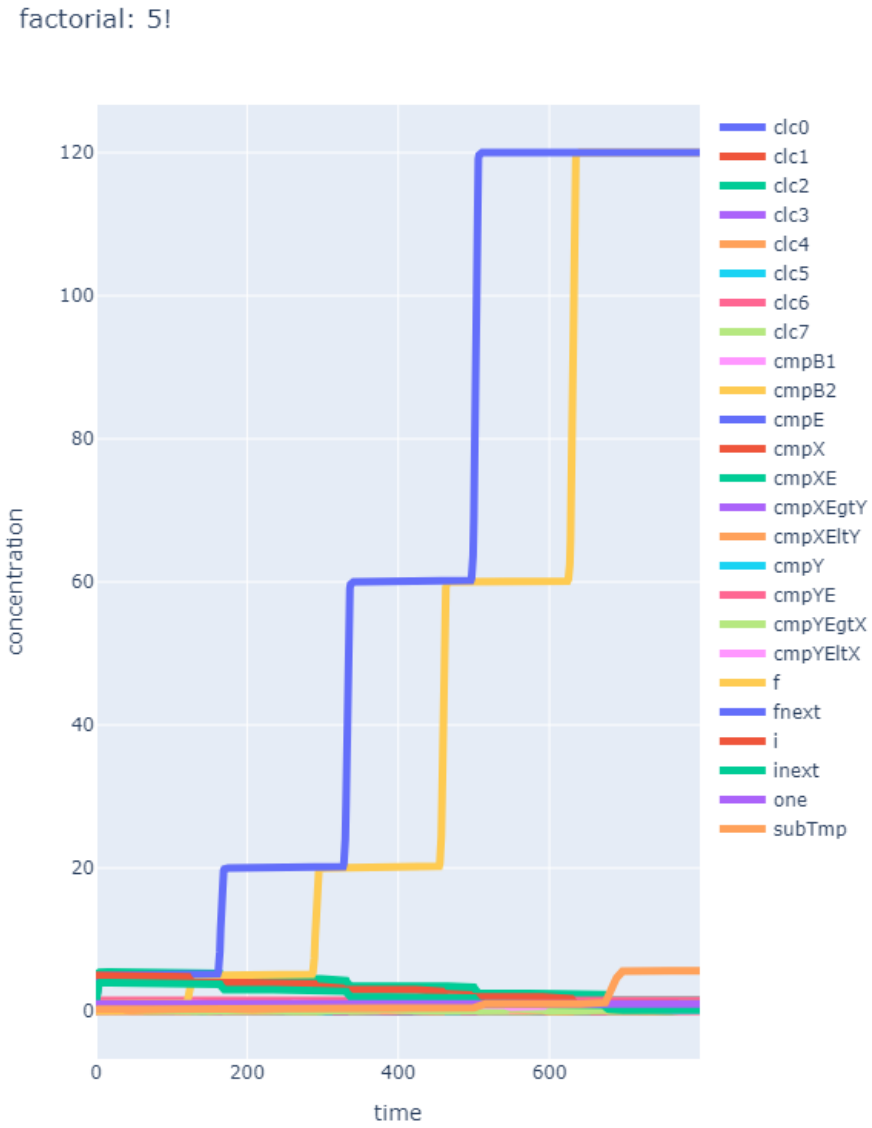


Figure 10: Simulation of 5 factorial program, showing all signals. This nicely showcases the extended *cmp* module signals, for the conditions used in the factorial program.

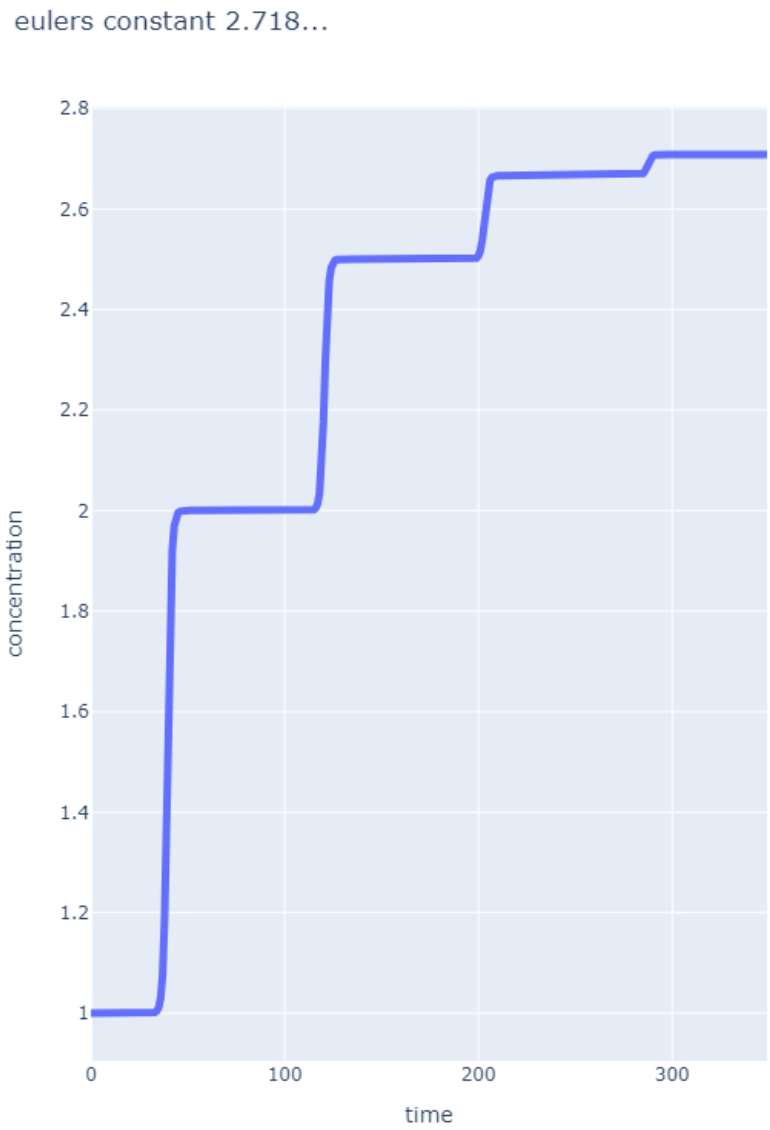


Figure 11: Simulation of eulers constant program, only showing result signal.

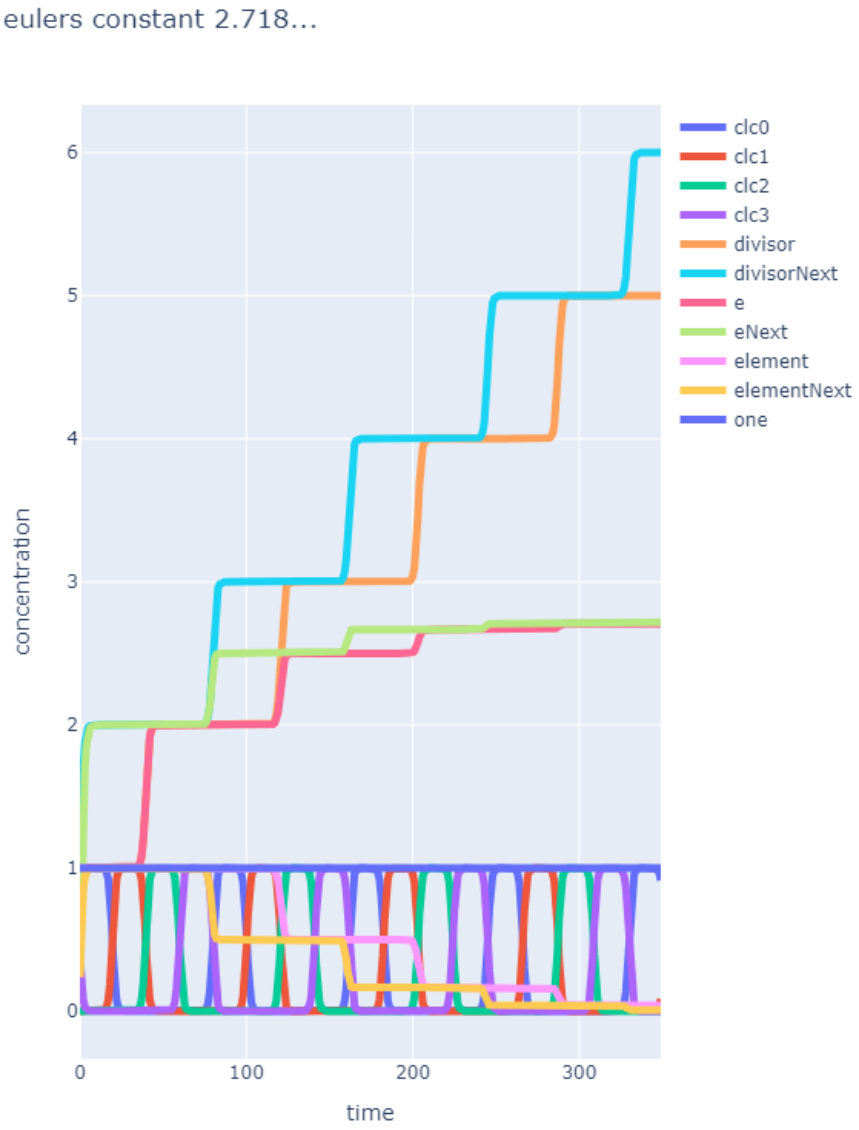


Figure 12: Simulation of eulers constant program, showing all signals.

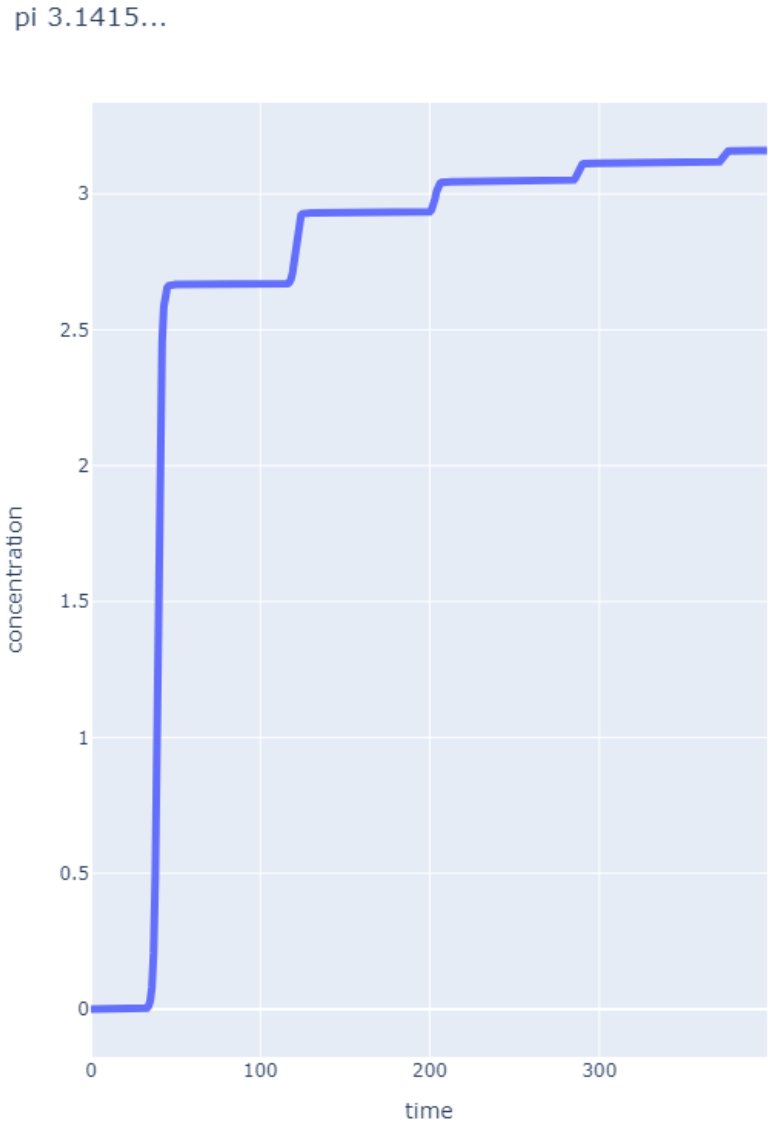


Figure 13: Simulation of pi program, only showing result signal.

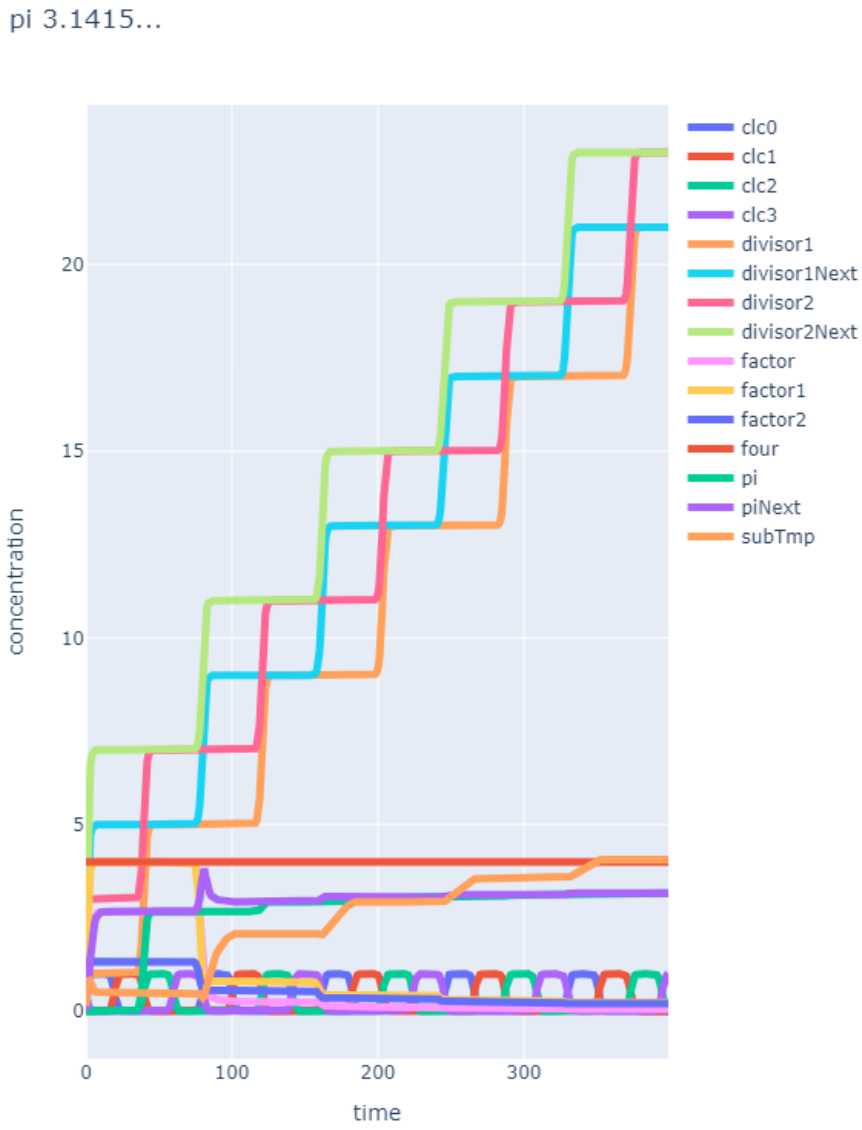


Figure 14: Simulation of pi program, showing all signals.

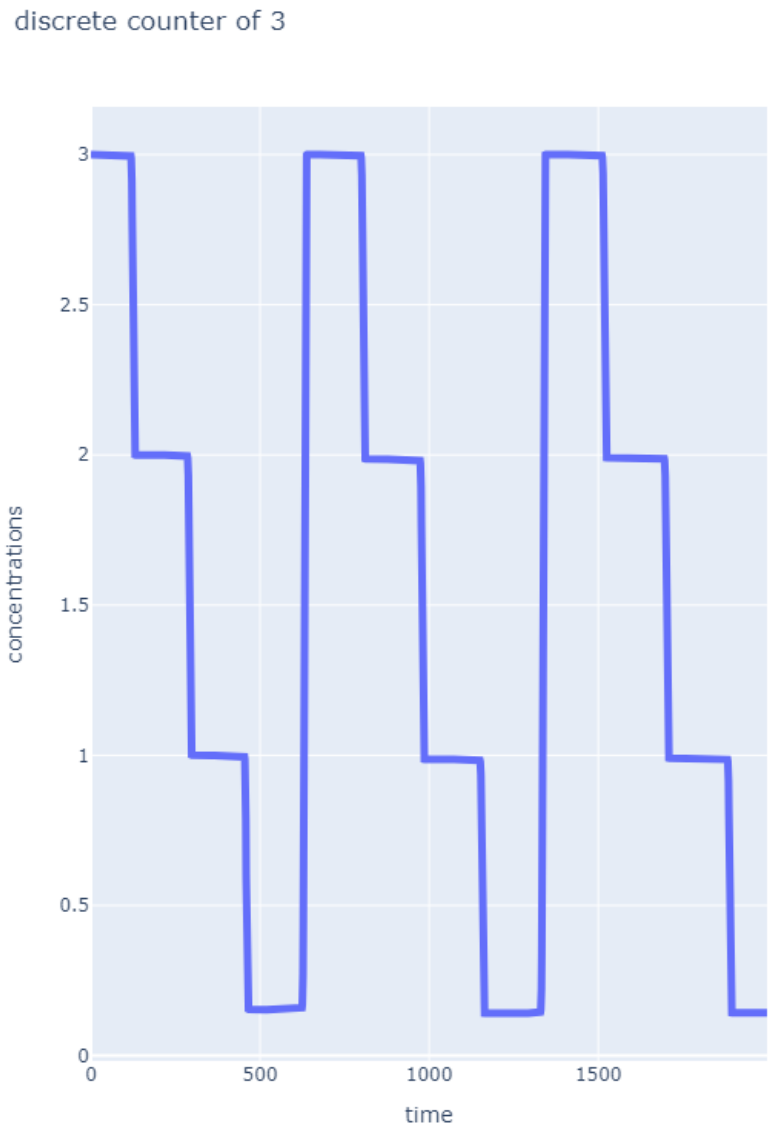


Figure 15: Simulation of discrete counter for 3 program, only showing result signal.

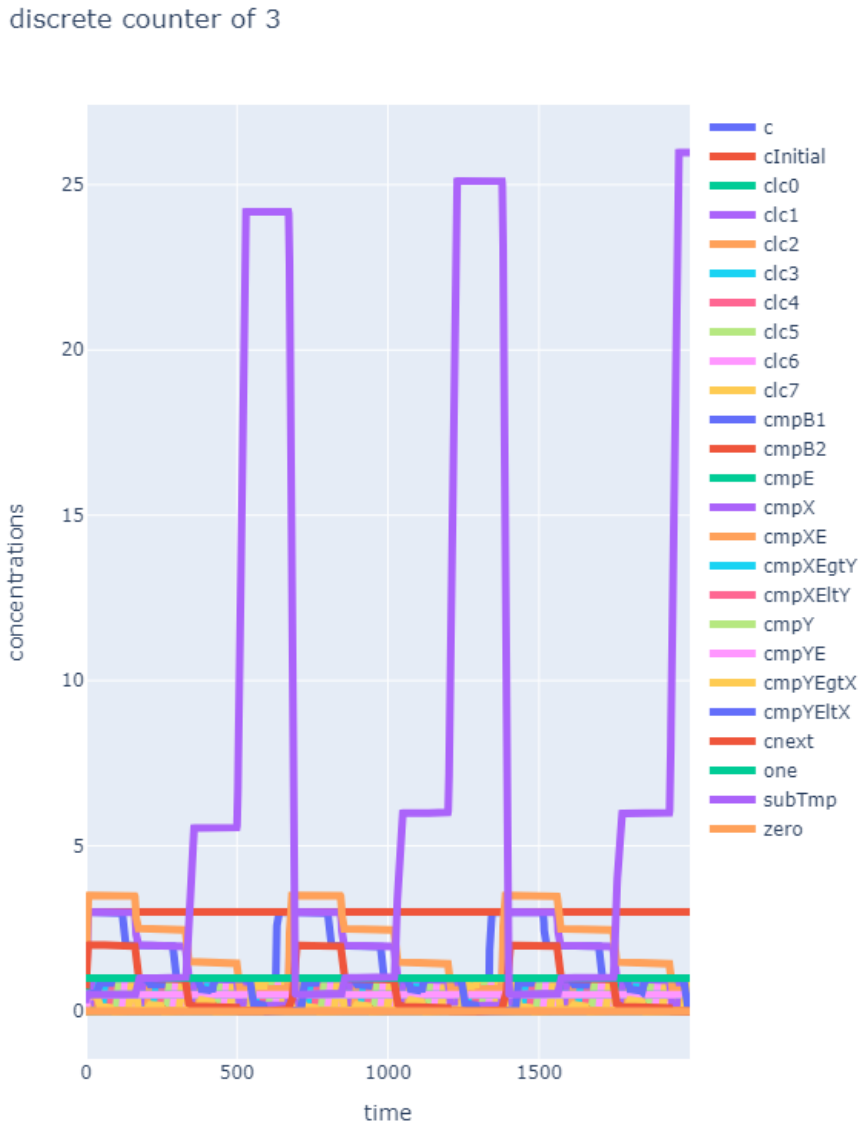


Figure 16: Simulation of discrete counter for 3 program, showing all signals.

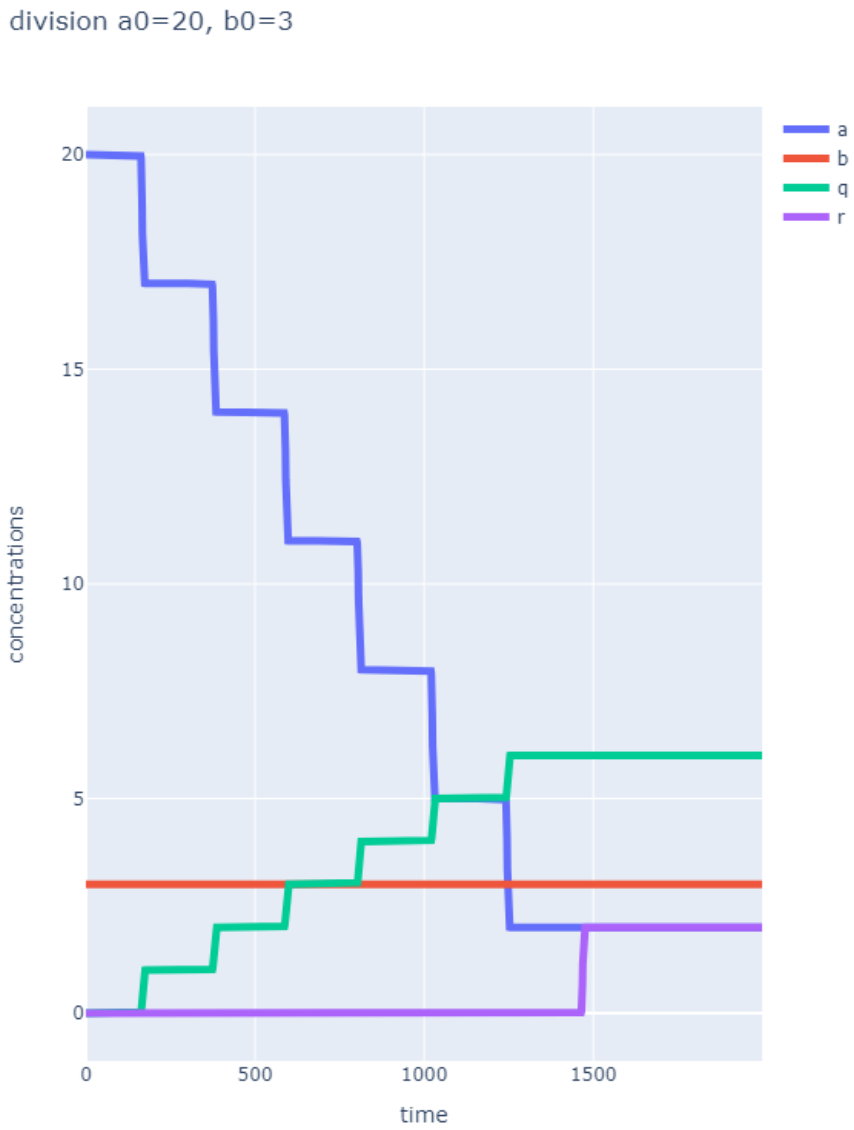


Figure 17: Simulation of division program for a0=20 and b0=3, only showing result signal.

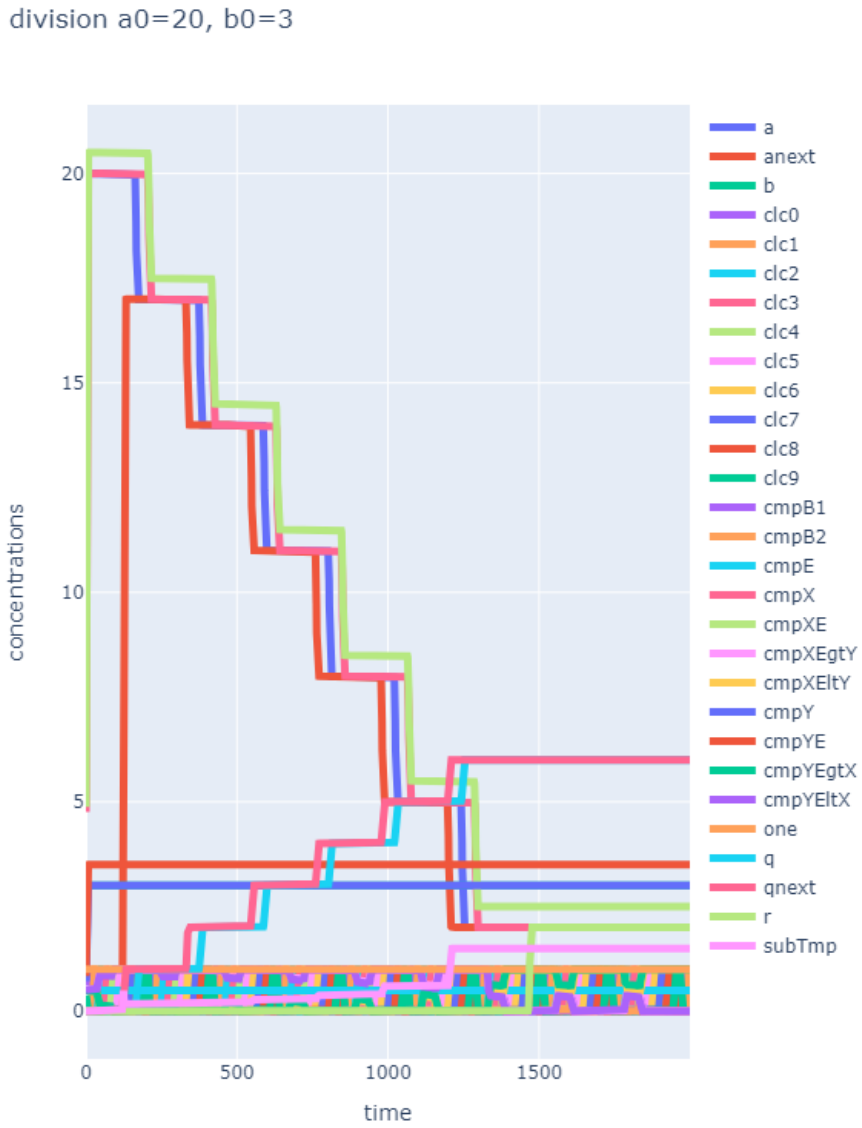


Figure 18: Simulation of division program for a0=20 and b0=3, showing all signals.

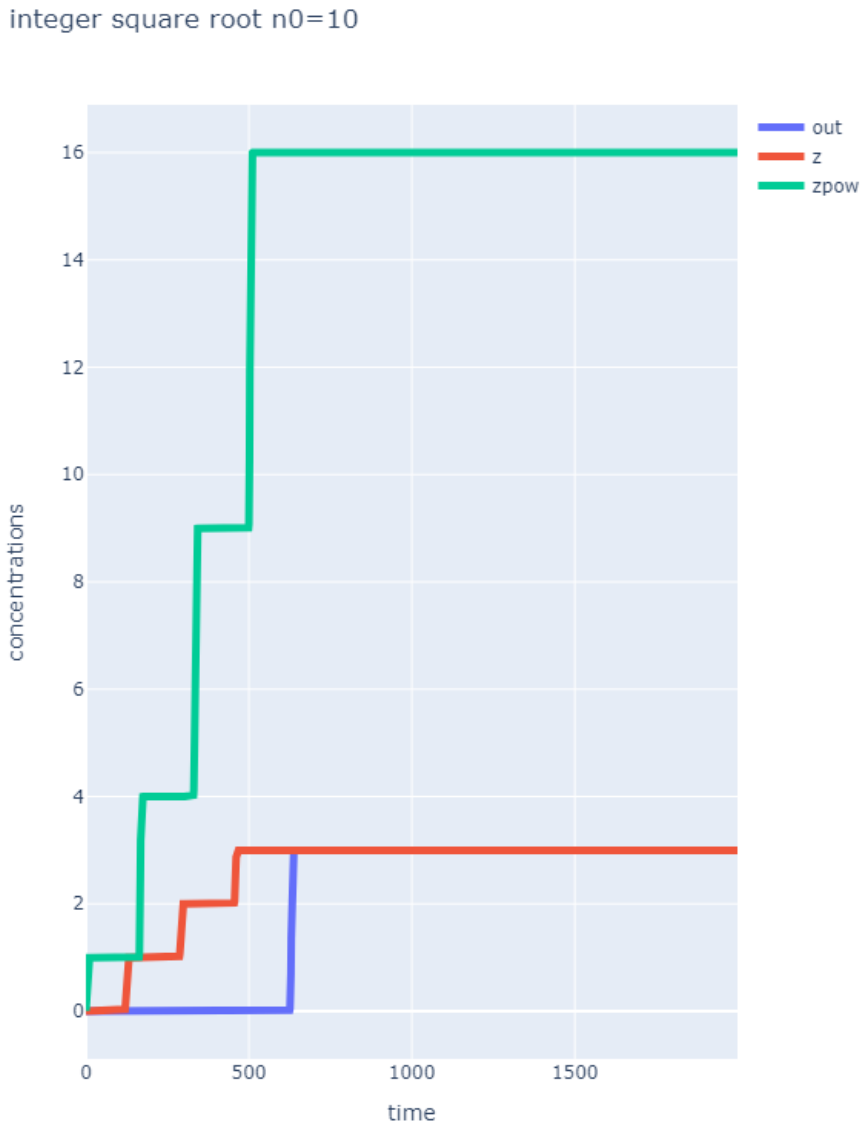


Figure 19: Simulation of integer square root program for n0=10, only showing result signal.

Figure 20: Simulation of integer square root program for n0=10, showing all signals.