CRN++: Molecular programming language

Marko Vasić[1] · David Soloveichik[1] · Sarfraz Khurshid[1]

Abstract

Synthetic biology is a rapidly emerging research area, with expected wide-ranging impact in biology, nanofabrication, and medicine. A key technical challenge lies in embedding computation in molecular contexts where electronic micro-controllers cannot be inserted. This necessitates effective representation of computation using molecular components. While previous work established the Turing-completeness of chemical reactions, defining representations that are faithful, efficient, and practical remains challenging. This paper introduces *CRN++*, a new language for programming deterministic (mass-action) chemical kinetics to perform computation. We present its syntax and semantics, and build a compiler translating *CRN++* programs into chemical reactions, thereby laying the foundation of a comprehensive framework for molecular programming. Our language addresses the key challenge of embedding familiar imperative constructs into a set of chemical reactions happening simultaneously and manipulating real-valued concentrations. Although some deviation from ideal output value cannot be avoided, we develop methods to minimize the error, and implement error analysis tools. We demonstrate the feasibility of using *CRN++*on a suite of well-known algorithms for discrete and real-valued computation. *CRN++* can be easily extended to support new commands or chemical reaction implementations, and thus provides a foundation for developing more robust and practical molecular programs.

1 Introduction

A highly desired goal of synthetic biology is realizing a programmable chemical controller that can operate in molecular contexts incompatible with traditional electronics. In the same way that programming electronic computers is more convenient at a higher level of abstraction than that of individual flip-flops and logic circuits, we similarly expect molecular computation to admit specification via programming languages sufficiently abstracted from the hardware. This paper focuses on developing a compiler for a natural imperative programming language to a deterministic (mass-action) chemical reaction network implementing the desired algorithm. We do not directly make assumptions on how the resulting reactions would be implemented in chemistry. This could in principle be

achieved by DNA strand displacement cascades (Soloveichik et al. 2010), or other programmable chemical technologies such as the PEN toolbox (Baccouche et al. 2014).

Deterministic (mass-action) chemical kinetics is Turing universal (Fages et al. 2017), thus in principle allowing the implementation of arbitrary programs in chemistry. Turing universality was demonstrated by showing that arbitrary computation can be embedded in a class of polynomial ODEs (Bournez et al. 2017), and then implementing these polynomial ODEs with mass-action chemical kinetics. While these results establish a sound theoretical foundation and show the power of chemistry for handling computation tasks in general, translating and performing specific computational tasks can lead to infeasibly large and complex sets of chemical reactions.

In this work we develop a programming paradigm for chemistry, based on the familiar imperative programming languages, with the aim of making molecular programming more intuitive, and efficient. Most commonly used programming languages such as C, Java and Python, are imperative in that they use statements that change a program's state, with typical branching constructs such as if/ else, loops, etc. Note that although CRNs are sometimes talked about as a programming language (Chen et al. 2013), they are difficult to program directly (it is even

✉ Marko Vasić
vasic@utexas.edu

David Soloveichik
david.soloveichik@utexas.edu

Sarfraz Khurshid
khurshid@utexas.edu

[1] The University of Texas at Austin, Austin, USA

unfair to equate them with assembly language). In contrast, *CRN++* operates at a much higher level.

We introduce the syntax and semantics of *CRN++*, an imperative programming language that compiles to deterministic (mass-action) chemical reaction networks. *CRN++* has an extensible toolset including a simulation framework and error analysis functionality. A user specifies a *sequence* of statements, termed commands, to execute. Assignment, comparison, loops, conditional execution, and arithmetic operations are supported. The generated reactions are logically grouped into modules performing the corresponding command. Each module transforms initial species concentrations to their steady-state values which are the output of the module. We ensure that such modules are composable by preserving the input concentrations at the steady-state.

A mapping of imperative program logic to chemical reactions manipulating continuous concentrations poses various challenges that we must address. All reactions happen concurrently, making it difficult to represent sequential computation where, for example, the result of one operation is first computed and then used in another operation. Similarly, all branches of the program execution (i.e., if/else) are followed simultaneously to some degree. To mimic sequential execution in mass-action chemistry, we ensure that the reactions corresponding to the current command happen quickly, while other reactions are slow. For this we rely on a chemical oscillator in which the *clock* species oscillate between low and high concentrations, and catalyzing reactions with different clock species. To achieve conditional execution, we further need to ensure that the reactions corresponding to the correct execution branch happen quickly, while those corresponding to other branches are inhibited. Our *cmp* module sets *flag* species to reflect the result of comparison, and these species catalyze the correct branch reactions.

A further source of error is that the set of basic modules, such as addition, converge to the correct value only in the limit, thus computing approximately in finite time. To mitigate this source of error, we choose a set of modules to exhibit exponential (fast) convergence. We further provide a toolkit for error analysis and detection, which can help a user to identify and mitigate the source of error, guiding the design of more accurate *CRN++* programs.

We demonstrate the expressiveness of our language by implementing and simulating common discrete algorithms such as greatest common divisor, integer division, finding integer square root, as well as real-valued (analog) algorithms such as computing Euler's number and computing π. We implement the *CRN++* compiler which generates the reactions implementing a high level imperative algorithm, and use the *CRNSimulator* package (http://users.ece.utexas.edu/~soloveichik/crnsimulator.html) to manipulate

and simulate chemical reactions using Mathematica. *CRN++* is an extensible programming language allowing for easy addition of new modules; we release the open-source version (https://github.com/marko-vasic/crnPlusPlus) of the tool to enable others make use of it, and extend it further.

2 Examples

In this section we discuss the characteristics of chemical reaction networks (CRNs) through examples. First, the overall idea of computation in CRNs is presented, followed by example programs in *CRN++*. The focus is to give a high level idea of our technique, while later sections discuss internal details.

Although historically the focus of the study of CRNs was on understanding the behavior of naturally occurring biological reaction networks, recent advancements in DNA synthesis coupled with general methods for realizing arbitrary CRNs with DNA strand displacement cascades (Soloveichik et al. 2010) opened the path to engineering with chemical reactions. In this work we are not interested in a way to engineer the molecules implementing a reaction but focus on reaction behavior and dynamics. We abstract away molecule implementation information and denote molecular species with letters (e.g. A).

Molecular systems exhibit complex behaviors governed by chemical reactions. To give a formal notation of chemical reaction networks, consider the CRN 1 (Buisman et al. 2009):

CRN 1 Example chemical reaction network

$$A + B \xrightarrow{1} A + B + C \tag{1}$$

$$C \xrightarrow{1} \emptyset \tag{2}$$

The CRN 1 consists of two reactions. A chemical reaction is defined with *reactants* (left side), *products* (right side), and *rate constant* which quantifies the rate at which reactants interact to produce products. To illustrate this, reaction 1 is composed of *reactants* $= \{A, B\}$, *products* $= \{A, B, C\}$, and rate constant $k = 1$. Since most reactions in *CRN++* have the rate constant equal to 1, from now on we drop the rate constant when writing reactions, unless it is different than 1. Note that multiple molecules of same species can be in a list of reactants (analogously for products); to support this we use the multiset notation. As an example, to describe reaction: $A + A \longrightarrow B$ we write

$reactants = \{A^2\}$, where the upper index (2) represents multiplicity (number of occurrences).

It may seem that a molecule of C is produced out of nothing in reaction 1, since the multiset of reactants is a submultiset of the products. This represents a level of abstraction where *fuel* species that drive the reaction are abstracted away (i.e., the first reaction corresponds to $F + A + B \longrightarrow A + B + C$). Making this assumption allows us to focus on the computationally relevant species. The choice to use general (non-mass/energy preserving) CRNs is an established convention for DNA strand displacement cascades (Soloveichik et al. 2010).

When the molecular counts of all species are large, and the solution is "well-mixed", the dynamics of the system can be described by ordinary differential equations (mass-action kinetics). Molecular concentrations are quantified by a system of ODEs, where the concentration of each species is characterized by the following ODE:

$$\frac{d[S]}{dt} = \sum_{\forall rxn \in CRN} k(rxn) \cdot netChange(S, rxn) \cdot \prod_{\forall R \in reactants(rxn)} [R]^{m_{rxn}(R)}(t)$$

The right side is a sum over reactions in the CRN, where $k(rxn)$ is a rate of reaction rxn, $netChange(S, rxn)$ is a net change of molecules of S upon triggering of rxn (can be negative), and $m_{rxn}(R)$ is the multiplicity of reactant R in reaction rxn. To illustrate the general formula, the set of ODEs characterizing CRN 1 is:

$$\frac{d[A]}{dt} = 0, \quad \frac{d[B]}{dt} = 0, \quad \frac{d[C]}{dt} = [A](t) \cdot [B](t) - [C](t)$$

Since the concentrations $[A]$ and $[B]$ are constant (derivatives zero), we have $\frac{d[C]}{dt} = [A](0) \cdot [B](0) - [C](t)$. Thus $[C](t)$ is increasing when smaller than $[A](0) \cdot [B](0)$, decreasing in the opposite case, and does not change when $[C](t) = [A](0) \cdot [B](0)$. Therefore system has a global stable steady-state $[C] = [A](0) \cdot [B](0)$. We say that this module computes multiplication, due to the relation between initial concentrations and concentrations at the steady state.

We simulate and plot the dynamics of the multiplication CRN, as shown in Fig. 1. Initial concentrations of A and B are 6 and 2, respectively, while the concentration of C approaches value 12. Note that the exact value defined by the steady state ($[C](t) = 12$) is reached only at the limit of time going to infinity. Since the computation has to be done in finite time, the presence of error is unavoidable. This error raises challenging issues with programming in chemistry, and necessitates techniques for controlling it. One crucial property that determines the error is the convergence speed of the module. The multiplication command in *CRN++* is implemented through the above module, following the design principles of *convergence*
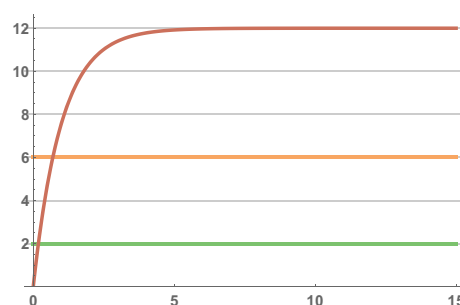

Fig. 1 Multiplication CRN. $[A]$ shown in orange, $[B]$ in green, and $[C]$ in red. (Color figure online)

speed and *composability* described in Sect. 3. Chemical reactions are abstracted away from a user who can simply write $mul[a, b, c]$ to multiply.

CRN++ is an imperative language, and as such supports sequential execution. Note that even a simple operation of multiplying and storing into the same variable, e.g. $A := A * B$, requires support for sequential execution. We use operator ":=" to relate input and output concentrations; $A := A * B$ denotes that $[A](t)$ converges to $[A](0) * [B](0)$. The above implementation of the *mul* module necessarily assumes that the output species is different from the input species. Otherwise, $mul[a, b, a]$ goes to infinity or 0 depending on the value of B. To implement $A := A * B$, we split the computation into two sequential steps: (1) $C := A * B$, (2) $A := C$. To multiply we use the *mul* module described above. For the assignment we use the load module (*ld*). To ensure the assignment executes after the multiplication, we catalyze the two modules with the clock species that reach their high values in different phases of the oscillator. Importantly, the chemical oscillator and clock species are abstracted away from a user, who simply uses the *step* construct to order reactions: $step[\{mul[a, b, c]\}]$, $step[\{ld[c, a]\}]$.

One of the basic blocks of programming languages are conditional branches, executing upon success of a precondition. Similarly to implementing sequential operations, we implement conditional execution by activating (through catalysis) some reactions and deactivating others, depending on a result of condition. Since no species can be driven to 0 in finite time,[1] all branches of condition will be active to some extent, which makes this an interesting source of errors without direct analogy in digital electronics. Analogous to clock species in sequential execution, reactions are catalyzed by *flag* species to support branching. The flag species have high and low values that reflect the result of the comparison. Our *cmp* module sets the flag species to

---

[1] Although certain pathological CRNs can drive concentrations to infinity in finite time (e.g., $2A \rightarrow 3A$), and thereby drive certain other species to 0 in finite time (e.g., with an additional $B + A \rightarrow A$), these cases cannot be implemented with any reasonable chemistry.

reflect the result of the comparison. In the following example we demonstrate the usage of *cmp* module and conditional execution.

To demonstrate the expressiveness of our language we showcase the implementation of Euclid's algorithm (Fig. 2) to compute the greatest common divisor (GCD) of a two numbers. The GCD is computed by subtracting the smaller of the values from the larger one until they become equal.

Figure 3a shows the implementation of Euclid's algorithm in *CRN++*. Lines 2–3 define the initial concentrations of species *a* and *b*, where constants *a*0 and *b*0 are values for which GCD is computed. To order the execution, the *step* construct is used. Multiple instructions that do not conflict with each other can be part of the same step and they are executed in parallel. In the first step *a* and *b* are stored into temporary variables and compared, setting the flag species to reflect the result of the comparison. The second step uses the result of the previous comparison, and effectively stores $a - b$ into *a* if $a > b$, and vice versa. Since the same species cannot be used as both input and output to *sub* module, temporary variables are used (*atmp* and *btmp*). Steps repeatedly execute due to the oscillatory behavior of the clock species, thus implementing looping behavior by default; the steps can be viewed as being inside of the 'forever' loop. *CRN++*, in addition to the language and compiler to chemical reactions, is connected to the simulation backend that enables convenient testing for correctness.

We show a simulation of the GCD program in Fig. 3b where GCD(32,12) is computed. Although not visible in the plot we can identify a number of non-idealities in the *CRN++*implementation. First, modules converge to correct values only in limit of time going to infinity; thus for example the first subtraction (second step) will set *a* to value close to but not exactly 20. Second, modules that should not be executing cannot be completely turned off, and so when the first step executes the second does so as well (of course, in much smaller extent). Analogously the two subtraction operations in the second step are supposed to be mutually exclusive, and yet they co-occur to some extend. More discussion on the sources of error is provided in Sect. 3.4.

In addition, we implement a set of algorithms in (a) discrete space—counter, factorial, integer division, integer square root, as well as in (b) continuous space—by implementing *CRN++*programs that approximate value of *Euler's* constant and π. These examples are shown in Sect. 4.

3 Technique

This section explains *CRN++*, both the underlying constructs used to build it, as well as high level primitives that represent the language itself. We start by presenting high-level modules that are at the core of *CRN++*(Sect. 3.1), followed by explanation of how the sequential behavior is achieved (Sect. 3.2), after which we give an overview of *CRN++*grammar (Sect. 3.3), and finally we discuss the error detection and analysis tools we provide (Sect. 3.4).

3.1 Modules

Modules represent the core of *CRN++*, and in their form are somewhat analogous to the instruction set architecture (ISA) in machine languages. Modules implement basic operations such as load, add, subtract, multiply, compare. We provide the exhaustive list of modules in Table 1. Importantly, *CRN++*is extensible, and supports easy addition of new modules.

There are multiple ways of computing addition and other operations in chemistry. As mentioned in the previous section, our implementation choice is led by two basic principles: (a) convergence speed, and (b) composability.

3.1.1 Convergence speed

To provide intuition about the convergence speed of modules we analyze the following CRN:

CRN 2 Simple CRN

$A + B \longrightarrow \emptyset$
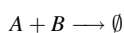
The ODEs describing the above CRN are: $\frac{d[A]}{dt} = -[A](t) * [B](t)$ and $\frac{d[B]}{dt} = -[A](t) * [B](t)$. Since the amount of *B* decreases with the same speed as *A*, we can express the dynamics of the system in terms of $D_0 = [B](0) - [A](0)$:

$$\frac{d[A]}{dt} = -[A](t) * ([A](t) + D_0)$$

If $D_0 \neq 0$, the solution is:

$$[A](t) = \frac{[A](0)D_0}{-[A](0) + [A](0)e^{D_0 t} + D_0 e^{D_0 t}}$$

To consider the convergence speed we look at the non-constant part of the equation. Due to the factor e^{-t} the decrease of the non-constant part is exponential, and we say that the convergence speed is *exponential*. If $D_0 > 0$

```
1: procedure GCD(a, b)
2:     while a ≠ b do
3:         if a > b then
4:             a ← a − b
5:         else
6:             b ← b − a
7:         end if
8:     end while
9:     return a
10: end procedure
```

Fig. 2 Euclid's algorithm for computing GCD

$([B](0) > [A](0))$ terms with exponential factors tend to infinity, and $[A]$ to zero. Conversely, when $D_0 < 0$, exponential factors converge to zero, and $[A]$ to $-D_0$.

When $D_0 \neq 0$ so far; by solving the ODE for $D_0 = 0$ we get a following:

$$[A](t) = \frac{[A](0)}{1 + [A](0)t}$$

In this equation, non-constant part is not anymore exponential, but linear; thus we say that the convergence speed is *linear*. To summarize, CRN 2 exhibits exponential convergence speed when $[A](0) \neq [B](0)$, and linear otherwise.
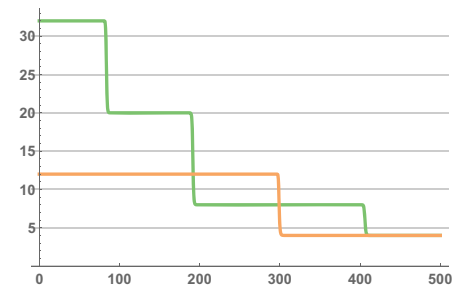
Fig. 3 Implementation of Euclid's algorithm for computing GCD in CRN++(left), simulation results of the implementation (right). (Color figure online)

```
1 crn = {
2   conc[a,a0],
3   conc[b,b0],
4   step[{
5     ld[a, atmp],
6     ld[b, btmp],
7     cmp[a,b]
8   }],
9   step[{
10    ifGT[{ sub[atmp,btmp,a] }],
11    ifLT[{ sub[btmp,atmp,b] }]
12  }]
13 };
```

(a) GCD implementation

(b) Dynamic simulation of the GCD program for $a0 = 32$, $b0 = 12$. Concentrations of a (green), and b (orange) are shown in function of time.

Table 1 CRN++modules

Type	Restrictions	Output (steady state)	CRN
ld[A, B]	$B \not\equiv A$	$B := A$	$A \longrightarrow A + B$
			$B \longrightarrow \emptyset$
add[A, B, C]	$C \not\equiv A \wedge C \not\equiv B$	$C := A + B$	$A \longrightarrow A + C$
			$B \longrightarrow B + C$
			$C \longrightarrow \emptyset$
sub[A, B, C]	$C \not\equiv A \wedge C \not\equiv B$	$C := \begin{cases} A - B, & A > B \\ 0, & \text{otherwise} \end{cases}$	$A \longrightarrow A + C$
			$B \longrightarrow B + H$
			$C \longrightarrow \emptyset$
			$C + H \longrightarrow \emptyset$
mul[A, B, C]	$C \not\equiv A \wedge C \not\equiv B$	$C := A \cdot B$	$A + B \longrightarrow A + B + C$
			$C \longrightarrow \emptyset$
div[A, B, C]	$C \not\equiv A \wedge C \not\equiv B$	$C := A/B$	$A \longrightarrow A + C$
			$B + C \longrightarrow B$
sqrt[A,B]	$B \not\equiv A$	$B := \sqrt{A}$	$A \xrightarrow{1} A + B$
			$B + B \xrightarrow{\frac{1}{2}} \emptyset$
cmp[A,B]	$A \not\equiv B$	Sets flag species	* Two CRNs (mapping and AM) triggering in a two consecutive phases (as discussed in Sect. 3.1.7)

The first column denotes the type of the module. The restrictions column imposes compile-time restrictions for using modules, here $\not\equiv$ is used to mean different species (not values)

The output column shows the value of outputs at the steady state. Finally, the CRN column shows chemical reactions implementing the module

*CRN++*models computation in which an instruction executes in a step (clock phase), and computation in subsequent steps depends on values computed in previous steps. In this reason, it is of a crucial importance to achieve a high precision of computation until the end of a current step; and to achieve this goal it is necessary to ensure a high convergence speed. For this reason we want to ensure that all of our modules exhibit exponential convergence. So, for example, we do not want to use the reaction $A + A \rightarrow C$ to compute division by 2 since it exhibits only linear convergence (corresponding to the $A = B$ analysis above).

3.1.2 Composability

To explain composability we analyze two different CRNs that perform addition (i.e., compute $C := A + B$).

CRN 3 Addition CRN (preserves inputs). Inputs: A and B, output: C

$A \longrightarrow A + C$
$B \longrightarrow B + C$
$C \longrightarrow \emptyset$
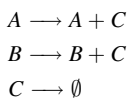
CRN 4 Addition CRN (destroys inputs). Inputs: A and B, output: C

$A \longrightarrow C$
$B \longrightarrow C$

Although both of these CRNs compute addition (this is evident for CRN 4; more detailed derivation is presented for CRN 3 in Sect. 3.1.4), they behave very differently when composed with other modules. For example, to compute $W := (X * Y) + Z$, we can combine the *mul* module (CRN 1) with X and Y as inputs and some species K as output, with an addition module with K and Z as inputs and W as output. If we use CRN 3 for addition, note that K is not changed by it (the input acts catalytically), and thus K approaches $X * Y$ as expected, allowing the addition module to correctly compute $(X * Y) + Z$ in the limit. In contrast, if we use CRN 4 to perform addition, then by consuming K the addition module will affect the equilibrium of the multiplication module, driving it lower than the desired value $X * Y$. In this way CRN 4 is not composable, while CRN 3 allows for the correct composition.

More generally, for any set of modules that: (1) use inputs catalytically (does not produce or consume inputs), (2) exhibit exponential convergence, and (3) have a unique stable steady state, a CRN composed of such modules also has the above three properties. A proof can be found in Buisman et al. (2009). From this it follows that the composed CRN of the *mul* module and CRN 3 exhibits the exponential convergence speed, and has a stable steady state defined by $W := (X * Y) + Z$.

Note that the above discussion of composability concerns reactions occurring within a single step construct (i.e., we discuss correct computations when reactions occur concurrently). Even non-composable reactions can be composed by separating them in time via the step construct. However, in order to keep the number of steps low and ensure faster computation we aim to make reactions composable when possible. With this, we have set up the two main design criteria (convergence speed and composability) for the modules, and we next describe the *CRN++*modules.

3.1.3 *Ld* module

Loads the value from source (first argument) into a destination (second argument). The CRN used for load operation is following:

CRN 5 load CRN

$A \longrightarrow A + B$
$B \longrightarrow \emptyset$
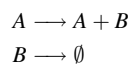
A is the input and B is the output species. This module, similar to *add* (see next section), has exponential convergence speed (Buisman et al. 2009). In addition, the concentration of input species is constant, thus ensuring composability.

3.1.4 *Add* module

Adds two values (first and second argument) and stores the result into destination (third argument). The Add CRN is shown in CRN 3.

By solving the ODEs that characterize $[C](t)$ we get the following equation: $[C](t) = [A](0) + [B](0) + ([C](0) - [A](0) - [B](0)) \cdot e^{-t}$. From the equation it follows that $[C]$ converges to $[A](0) + [B](0)$, and thus we say the CRN

performs addition. Moreover, the CRN exhibits exponential convergence.

3.1.5 *Sub* module

Subtracts the second input value from the first and stores into the destination (third argument).

CRN 6 subtraction CRN

$A \longrightarrow A + C$
$B \longrightarrow B + H$
$C \longrightarrow \emptyset$
$C + H \longrightarrow \emptyset$
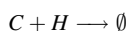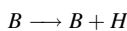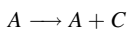
The above CRN was generated via evolutionary algorithms (Buisman et al. 2009); by analyzing its system of ODEs, the network computes truncated subtraction:

$$[C] = \begin{cases} [A] - [B], & \text{if } [A] > [B] \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

Input species A and B are not affected and the property of composability is satisfied. Neither we nor Buisman et al. found the analytical solution; however, our simulation results show that the module converges exponentially quickly unless $A = B$ (see the Sect. 3.1.1 for an analogous, easy to analyze case).[2] In a case inputs, A and B, are close to each other the computation error is higher. The error evaluation tools (Sect. 3.4) help in detecting and analyzing problematic cases (e.g., where A and B are close), thus enabling a user to redesign the CRN. In our examples, A and B usually differ by at least 1. Runtime assertions in the simulation package that automatically notify a user about these kind of problems would help identify the source of the error. Note that many algorithms can be refactored to reduce the error (see Sect. 6).

3.1.6 *Mul* module

Multiplies inputs (first and second argument) and stores into destination (third argument). The multiplication CRN is shown in Sect. 2. This CRN does not affect inputs and has exponential convergence speed (Buisman et al. 2009).

We have presented modules for performing arithmetic operations (*ld*, *add*, *sub*, *mul*). These modules are implemented within a single *step*. Multiple modules can be executed in parallel within a single *step* as long as there is no cyclic dependence between species: for example *mul*[a, b, c] and *add*[c,d,a] forms a cycle, the output of the *mul* is input to the *add*, and vice versa. Also, the CRN implementation imposes the restriction that same species cannot be used as both input and output to the same module (which is really a cycle of length 1). We now introduce the *cmp* module providing for conditional execution, which is executed in two *steps*.

3.1.7 *Cmp* module

Compares the two values, and produces signals (flag species) informing which value is greater or if they are equal.

The *cmp* module is implemented using two sequentially executed sets of reactions, which trigger in consecutive clock phases. In the first phase, the inputs (X and Y) are mapped to flag species X_{gtY} and X_{ltY}. Values are mapped to the range [0–1], by setting the initial concentrations of X_{gtY} and X_{ltY} such that their sum is 1. If, for example, $[X] = 80$ and $[Y] = 20$, flag species X_{gtY} and X_{ltY} converge to 0.8 and 0.2, respectively.[3] The mapping is done in order to preserve the original values of the inputs (X and Y), considering that the next phase of comparison consumes the compared values (flags). The mapping CRN is shown in CRN 7, and exhibits exponential convergence speed according to our analysis.
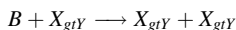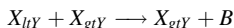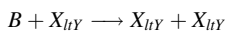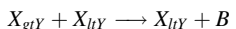

CRN 7 CRN for mapping compared values

$X_{gtY} + Y \longrightarrow X_{ltY} + Y$
$X_{ltY} + X \longrightarrow X_{gtY} + X$
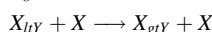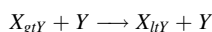
The goal of the second phase of comparison is to detect which value is greater. We use a chemical *Approximate Majority* (AM) algorithm (Cardelli and Csikász-Nagy 2012) to detect if X_{gtY} or X_{ltY} is in the majority. All molecules of the less populous species convert to the more populous species. The AM reactions are:

---

[2] A problematic feature of this module is that the concentration of H increases without bound if $[A] < [B]$. Although this does not affect the correctness of the computation, it is an interesting open question whether there is another truncated subtraction module that satisfies composability and exhibits exponential convergence but has bounded concentrations for any input regime.

[3] This convergence happens irrespective of the initial concentrations of the flag species X_{gtY} and X_{ltY} (as long as they sum to 1), so they do not have to be reset before the mapping.

CRN 8 approximate majority CRN

$X_{gtY} + X_{ltY} \longrightarrow X_{ltY} + B$

$B + X_{ltY} \longrightarrow X_{ltY} + X_{ltY}$

$X_{ltY} + X_{gtY} \longrightarrow X_{gtY} + B$

$B + X_{gtY} \longrightarrow X_{gtY} + X_{gtY}$

Recall that the normalization CRN is run in a previous step, so:

$$[X_{gtY}](0) + [X_{ltY}](0) = 1 \qquad (4)$$

Then in the AM network, the species (X_{gtY}, X_{ltY}, B) converge to values (1, 0, 0) if $X_{gtY}(0) > X_{ltY}(0)$ and (0, 1, 0) if $X_{gtY}(0) < X_{ltY}(0)$. In the subsequent step constructs, the species X_{gtY} are used as a catalysts in reactions that execute when $X > Y$, and the species X_{ltY} for the opposite case.

We now argue that CRN 8 exhibits exponential convergence. Due to normalization, (4) holds, and moreover $[X_{gtY}](t) + [X_{ltY}](t) + [B](t) = 1$ (for all time t). Taking the previous equality into account the ODEs of AM are:

$$\frac{d[X_{gtY}]}{dt} = [X_{gtY}](t)(1 - [X_{gtY}](t) - 2[X_{ltY}](t)) \qquad (5)$$

$$\frac{d[X_{ltY}]}{dt} = [X_{ltY}](t)(1 - [X_{ltY}](t) - 2[X_{gtY}](t)) \qquad (6)$$

Consider the function $f(t) = \frac{(X_{gtY} - X_{ltY})^3}{X_{gtY} X_{ltY}}$ [the choice of this function is guided by the closed form solution for t as a function of $X_{gtY}(t)$ and $X_{ltY}(t)$ (Perron et al. 2009)]. Taking the derivative of $f(t)$ with respect to t and substituting (5) and (6) results in $f(t)$ again. This means that $f(t)$ has a solution $f(t) = Ce^t$, which is unique by the Picard-Lindelöf theorem. Therefore, $Ce^t = \frac{(X_{gtY} - X_{ltY})^3}{X_{gtY} X_{ltY}} < 1/(X_{gtY} X_{ltY})$ since the concentrations are at most 1. This constrains $X_{gtY} X_{ltY} < C^{-1}e^{-t}$, which implies that X_{gtY} or X_{ltY} converges to 0 exponentially quickly.

If input species are initially equal ($[X_{gtY}](0) = [X_{ltY}](0)$) then the system preserves their equality, and following ODE holds: $[X_{gtY}]'(t) = 2[X_{gtY}](t)(1 - 3[X_{gtY}](t))$. This is a logistic differential equation which converges exponentially quickly to values (1/3, 1/3, 1/3).) Even though convergence is exponential we do not know how the time changes as a function of the difference between the input species, when the difference is small. Thus, even if the compared species are close to each other, the system may exhibit undesirably slow convergence to (1, 0, 0) or (0, 1, 0); As we later explain, our error evaluation framework can help detect such cases.

Equality checking Due to the ever-present error in chemical computation, checking for equality is actually approximate-equality checking. Consider having a chemical program with real values, then if the values are close to each other it is impossible to tell if they are actually equal but affected with error, or they represent different real valued signals. Due to this issue, while comparing for equality is impossible, we compare for ϵ-range equality. For discrete algorithms we use equality checking with $\epsilon = 0.5$, allowing easy comparison of the integer values (e.g., values in range (2.5, 3.5) are considered to be equal to 3).

To support equality checking we compare $x + \epsilon$ with y (generating signals X_{gtY} and X_{ltY}), and at the same time compare $y + \epsilon$ with x (generating signals Y_{gtX} and Y_{ltX}). Combining the signals of the two comparisons gives the desired result: If $X = Y$, signal X_{gtY} is high (X_{ltY} low) and Y_{gtX} is high (Y_{ltX} low) due to the added offset. To execute a reaction upon equality both X_{gtY} and Y_{gtX} are used catalytically. If $X > Y$, signal X_{gtY} is high (X_{ltY} low) and Y_{ltX} is high (Y_{gtX} low), so both X_{gtY} and Y_{ltX} should be used catalytically. Symmetrically for $X < Y$, both X_{ltY} and Y_{gtX} are used catalytically. After calling *cmp* in a step, a user can use instructions *ifGT* (*greater than*), *ifGE* (*greater or equal*), *ifEQ* (*equal*), *ifLT* (*less than*), *ifLE* (*less or equal*) in subsequent steps to conditionally execute reactions. Note that the flags are active until the next call to the *cmp* module.

3.2 Sequential execution

*CRN++*allows programming in a sequential manner, despite the intrinsically parallel nature of CRNs. To model sequential execution it is necessary to isolate two reactions from co-occurring, and control the order in which they
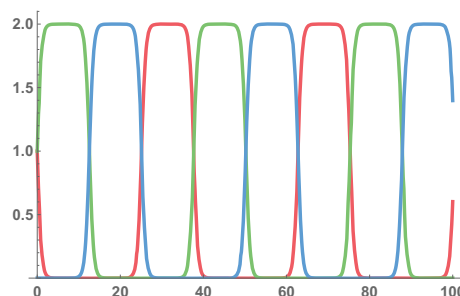

Fig. 4 Chemical oscillator containing 3 species: X_1 (red), X_2 (green), and X_3 (blue). (Color figure online)

happen. The key construct we rely on to achieve these goals is a chemical oscillator.

A chemical oscillator is a CRN in which the concentrations of species oscillate between low and high values. The oscillatory CRN (Lachmann and Sella 1995) we use is described with a following set of reactions:

CRN 9 oscillator CRN

$i = 1, \ldots, n-1 : X_i + X_{i+1} \longrightarrow 2X_{i+1}$

$\quad X_n + X_1 \longrightarrow 2X_1$
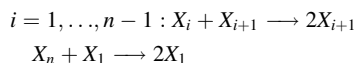
Concentrations of the clock species (X_i) oscillate (see Fig. 4). Different clock species have different oscillation phase and reach minimum and maximum at different times. To control the rate at which a reaction fires, clock species are added as both reactant and product (catalyst), in that way preventing reactions from co-occurring and ordering them (see CRN 10). While overlap between the clock species exists, it is small and thus enables sequential execution. To ensure the small overlap, in *CRN++* we use every third clock species, i.e. X_3, X_6, X_9, etc., to catalyze the reactions that execute at different time moments.

CRN 10 ordering reactions: original reactions (left), ordered (right)

$A \longrightarrow B$	$A + X_3 \longrightarrow B + X_3$
$B \longrightarrow C$	$B + X_6 \longrightarrow C + X_6$

The chemical oscillator is abstracted from a *CRN++* user, who can order reactions using the *step* construct. Reactions in different steps are isolated from each other through clock species acting catalytically.

Non-conflicting instructions can be part of the same step. Splitting instructions across multiple steps is needed in a case of (a) conditional execution—comparison needs to be done before conditional execution is possible; (b) reading and writing to the same species—this is not possible within the step (as discussed earlier), and requires temporal ordering. The number of clock species used is determined by the number of step instructions. Each step requires three clock species, with the exception of steps in which *cmp* module is used, for which six clock species are used. The *cmp* module requires two temporarily ordered operations: normalization and approximate majority, and thus six clock species are used. The oscillatory behavior of the clock species causes steps to get repeated eventually, causing the loop-like behavior.

3.3 Grammar

$\langle Crn \rangle ::= \text{`crn= \{'} \langle RootSList \rangle \text{`\}'}$

$\langle RootSList \rangle ::= \langle RootS \rangle$
$\quad | \quad \langle RootS \rangle \text{`,'} \langle RootSList \rangle$

$\langle RootS \rangle ::= \langle ConcS \rangle$
$\quad | \quad \langle StepS \rangle$

$\langle ConcS \rangle ::=$
$\quad \text{`conc['} \langle species \rangle \text{`,'} \langle number \rangle \text{`]'}$

$\langle StepS \rangle ::= \text{`step ['} CommandSList \text{`]'}$

$\langle CommandSList \rangle ::= \langle CommandS \rangle$
$\quad | \quad \langle CommandS \rangle \text{`,'} \langle CommandSList \rangle$

$\langle CommandS \rangle ::= \langle RxnS \rangle$
$\quad | \quad \langle ArithmeticS \rangle$
$\quad | \quad \langle CmpS \rangle$
$\quad | \quad \langle ConditionalS \rangle$

$\langle RxnS \rangle ::= \text{`rxn['} \langle Expr \rangle \text{`,'} \langle Expr \rangle \text{`,'} \langle number \rangle \text{`]'}$

$\langle ModuleS \rangle ::= \text{`ld ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$
$\quad | \quad \text{`add ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$
$\quad | \quad \text{`sub ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$
$\quad | \quad \text{`mul ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$
$\quad | \quad \text{`div ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$
$\quad | \quad \text{`sqrt ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$
$\quad | \quad \text{`cmp ['} \langle species \rangle \text{`,'} \langle species \rangle \text{`]'}$

$\langle ConditionalS \rangle ::=$
$\quad \text{`ifGT ['} \langle CommandSList \rangle \text{`]'}$
$\quad | \quad \text{`ifGE ['} \langle CommandSList \rangle \text{`]'}$
$\quad | \quad \text{`ifEQ ['} \langle CommandSList \rangle \text{`]'}$
$\quad | \quad \text{`ifLT ['} \langle CommandSList \rangle \text{`]'}$
$\quad | \quad \text{`ifLE ['} \langle CommandSList \rangle \text{`]'}$

$\langle Expr \rangle ::= \langle species \rangle \{ \text{`+'} \langle species \rangle \}$

Listing 1.1: *CRN++* Grammar

CRN++ grammar expresses syntactic rules of *CRN++* programs, and is shown in Listing 1.1. A program consists of a sequence of *RootS* statements, where *RootS* can be either *ConcS* which defines the initial concentration of species, or *StepS* which orders execution. Furthermore, a *StepS* statement is divided into a list of *CommandS* statements, where each *CommandS* is either: (1) *RxnS* which explicitly defines a reaction, (2) *ModuleS* which defines a module, or (3) *ConditionalS* which conditinally executes a block based on the result of a previous comparison. Based on the result of the comparison, only the appropriate conditional blocks are executed: *ifGT* (*greater than*), *ifGE* (*greater or equal*), *ifEQ* (*equal*), *ifLT* (*less than*), *ifLE* (*less or equal*). Note that the semantic rules of *CRN++* require that comparison is done in a step prior to the conditional execution.

The grammar can be easily extended; e.g., new modules can be added to the list of *ModuleS* nonterminals.

3.4 Error evaluation

Programming chemistry is inherently error-prone. We identify three specific sources of error in $CRN++$. First, CRNs converge asymptotically—only in the limit is the correct value reached—thus leaving certain amount of error in a finite time. Second, we cannot completely turn off modules which are not supposed to be currently executing, whether they belong to another sequential step, or to another branch of execution. Third, comparison has to take into account possible error in the compared values.

Our design decisions were based on minimizing the error; however since error cannot be avoided altogether, we provide a toolkit that helps in error analysis and guiding the CRN (program) design. Using the tool, users can, for any species of interest, track the difference between the correct value, and the (simulated) value in chemistry. For example, if operation add[a, b, c] is executed in a step, than $c = a + b$ is expected in the following step. $CRN++$allows measuring the difference between the expected $c = a + b$, and actual simulation value. To use the error evaluation, a user simply runs a Mathematica function we provide, called *EvaluateError* which accepts two inputs: (1) $CRN++$program and (2) simulation duration. This helps users analyze the error, and detect if the error builds up over time.

We analyze the value of operand a from GCD example Fig. 3, and plot the error in Fig. 5. In Fig. 5, the x-axis represents time, while the y-axis shows the difference between expected and actual value of a. Note that the error is sufficiently small that the algorithm executes correctly throughout the analyzed time. The error is not constant, which opens interesting questions of correlating the error with instructions in the program. To correlate error with program instructions we examine the GCD simulation (Fig. 3b). By looking at the time axis, it is easy to connect the first two spikes of the error with the subtraction of a.

We provide the error evaluation framework with the vision of it being a guiding element for programming in $CRN++$. We found this technique particularly useful for validation of programs, analyzing the error, understanding the sources of error, and redesigning the CRN for correctness.

4 Evaluation

In this section, we evaluate $CRN++$on a set of discrete and real-valued examples. Later in this section, we characterize the error of basic modules.

4.1 Examples

We first show a set of discrete algorithms implemented in $CRN++$—counter, factorial, integer division, integer square root; followed by real-valued algorithms—approximating Euler's and π constants.

4.1.1 Discrete counter

We implement a discrete counter that counts from a predefined value to zero, and repeats the process. Figure 6 shows both the $CRN++$program and the simulation results. Variable c stores value of the counter, *cInitial* stores the initial value of the counter for later refills, while *one* and *zero* store constants 0 and 1, respectively. Initial concentrations of the species are set in lines 2–3; where *c0* is a parameter of the program representing the initial value of the counter. In step 1 (lines 4–7), *one* is subtracted from the counter and stored into *cnext*, and at the same time the counter is compared with *zero*. In step 2 (lines 8–11), if the counter is zero, then its value is reset to the initial value; otherwise, *cnext* is stored into the counter. Steps exhibit looping behavior, thus described commands repeat.

4.1.2 Factorial

We implement a program in $CRN++$that computes the factorial function. Figure 7 shows both the program and simulation results. To compute the factorial of a positive integer n, we store n in the iterator variable i, and repeatedly multiply f with i, decreasing i until it becomes zero. Initial concentrations of the species are defined in line 2. In step 1 (lines 3–7), the iterator i is compared with *one* to check the termination condition, f is multiplied with i storing the value in the temporary variable *fnext*, and finally i is decremented storing the value in the temporary *inext*. In step 2 (lines 8–13), if $i > 1$, the temporary variables are stored back to f, and i.

4.1.3 Integer division

We implement integer division of two numbers, computing the quotient and the remainder of the operation. Figure 8
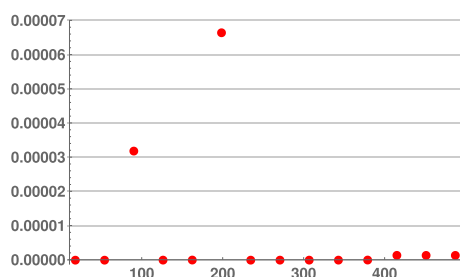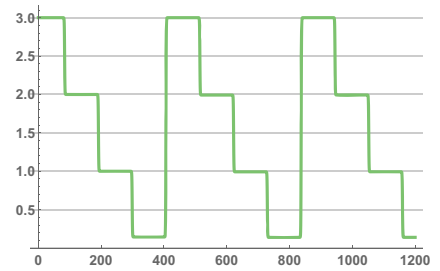

Fig. 5 Error evaluation of species a from GCD program

Fig. 6 Discrete counter. (Color figure online)

```
1 crn = {
2   conc[c,c0],  conc[ cInitial ,c0],
3   conc[one ,1],  conc[zero ,0],
4   step[{
5     sub[c,one,cnext],
6     cmp[c,zero]
7   }],
8   step[{
9     ifGT[{  ld [cnext,c]  }],
10    ifLE[{  ld [ cInitial ,c]  }]
11  }]
12 }
```
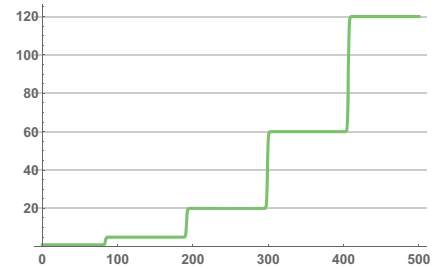
(a) *CRN++* code.

(b) Simulation results for $c0 = 3$; value of c is shown (green line).

Fig. 7 Factorial. (Color figure online)

```
1 crn={
2   conc[f ,1],  conc[one ,1],  conc[i ,f0 ],
3   step[{
4     cmp[i,one],
5     mul[f, i ,fnext ],
6     sub[i ,one ,inext ]
7   }],
8   step[{
9     ifGT[{
10      ld [ inext , i ],
11      ld [ fnext , f]
12    }]
13  }]
14 }
```

(a) *CRN++* code.

(b) Simulation results for $f0 = 5$; value of f is shown (green line).

Fig. 8 Division. (Color figure online)

```
1 crn={
2   conc[a,a0],  conc[b,b0],  conc[one ,1],
3   step[{
4     cmp[a,b]
5   }],
6   step[{
7     IfGE[{
8       sub[a,b,anext],
9       add[q,one,qnext]
10    }]
11  }],
12  step[{
13    IfGE[{
14      ld [anext,a],
15      ld [qnext,q]
16    }],
17    ifLT[{ ld [a, r ]}]
18  }]
19 };
```

(a) *CRN++* code.
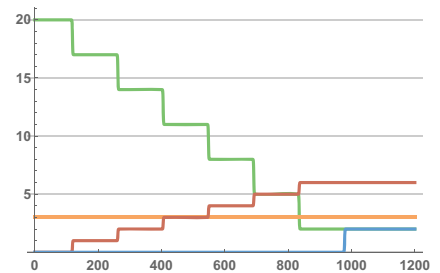
(b) Simulation results for $a0 = 20$, $b0 = 3$; values of a (green), b (orange), q (red), and of r (blue) are shown.

shows both the program and simulation results. Variable a stores the dividend, b the divisor, q the quotient, and r the remainder. The divisor is subtracted from the dividend until the dividend becomes smaller than the divisor. In step 1 (lines 3–5), the dividend and divisor are compared to detect if the termination condition is satisfied. In step 2 (lines 6–11), if $a > b$, the divisor is subtracted from the dividend, and the quotient is incremented. In step 3, if $a > b$, the new values for the dividend and quotient are restored from the temporary variables into the original ones. Also, in step 3, if $a < b$, the dividend is stored into the remainder (line 17).

4.1.4 Integer square root

Algorithm 1 Integer square root.

1: **procedure** INT SQRT(n)
2:      $z \leftarrow 0$
3:      **while** $(z + 1)^2 \leq n$ **do**
4:          $z \leftarrow z + 1$
5:      **end while**
6:      **return** z
7: **end procedure**

We implement a program that finds the integer square root of a number. Algorithm 1 shows the pseudo-code algorithm: the square root of a number n is found by iterating through positive integer numbers: 0, 1, 2, etc, until square of the number overshoots n. We map this algorithm to a *CRN++* program, and show the code and simulation results in Fig. 9. In step 1 (lines 3–7), z is incremented ($znext := z + 1$), and the square of $z + 1$ is computed ($zpow := znext * znext$) and compared with n. In step 2 (lines 8–11), if $zpow < n$, $znext$ is stored into z; otherwise, the result is computed and stored in *out*.

4.1.5 Euler's number approximation

So far we presented discrete algorithms, however chemistry naturally allows for real-valued (analog) computations. For programming with real values we make use of *CRN++* module for performing division. The *div* module follows the same design principles and characteristics as other arithmetic modules we presented (see Table 1).

We implement a program that approximates Euler's number. Euler's number can be computed using the following infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots$$

We map this program into *CRN++* code, as shown in Fig. 10. Variable e stores the current approximation of the constant, while *element* stores the current element of the series. In step 1 (lines 5–9), *element* is divided by the *divisor*, *divisor* is incremented for the next iteration, and e is incremented by the current element of the series. In step 2 (lines 10–14), the temporary variables *elementNext*, *eNext*, and *divisorNext*, are restored into the original variables. The precision achieved at the end of simulation is up to 5 decimal digits—2.71828.

4.1.6 Approximating π

We approximate the constant π via a *CRN++* program. We rely on the following infinite series to do so:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Figure 11 shows both the code and simulation. In step 1 (lines 6–13) following instructions are executed: (a) 4 is divided by the current divisor *divisor1* and stored into *factor1*, (b) 4 is divided by the *divisor2* and stored into *factor2*, (c) *factor1* subtracted by *factor2* is added to the *pi*, (d) *divisor1* and *divisor2* increased by 2 are stored into *divisor1Next* and *divisor2Next*, respectively. In step 2 (lines 14–18), the temporary variables *divisor1Next*, *divisor2Next*, and *piNext* are restored to the original variables. At the end of simulation we measure the output value 3.20185. Error evaluation shows that the error is greater in computing π compared to Euler's number due to using subtraction (of close values), which is the most error-prone out of all arithmetic operations we present [see (4.2)].

4.1.7 Size of CRNs

Table 2 shows the sizes of CRNs, in terms of the number of reactions and species, that result from compilation of corresponding *CRN++* programs.

4.2 Error characterization

In this section we evaluate error of the *CRN++* modules.

4.2.1 Error of arithmetic modules

Using our error evaluation mechanisms (Sect. 3.4) we analyze the error of the modules. We evaluate each module separately, on different inputs, to characterize its behavior. Figure 12 shows the error evaluation results, where a and b axes reflect the values of the first and second operand, respectively, and the height shows the magnitude of the error. The plots provide useful information such as: (a) The error of the *mul* and *add* modules increases as the value being computed increases; (b) Since these CRNs are symmetric with respect to the inputs, the error does not depend on the order of the arguments; (c) The *sub* module exhibits the largest error when the inputs are close to each
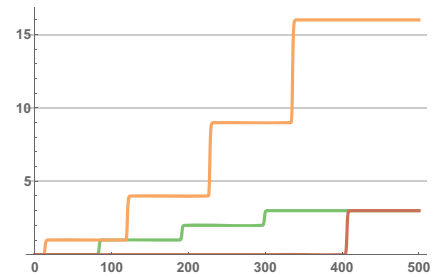
Fig. 9 Integer square root. (Color figure online)

```
1 crn = {
2   conc[one,1], conc[n,n0],
3   step[{
4     add[z,one,znext],
5     mul[znext,znext,zpow],
6     cmp[zpow,n]
7   }],
8   step[{
9     ifLT[{ ld [znext,z]}],
10    ifGE[{ ld [z,out]}]
11  }]
12 };
```

(a) *CRN++ code.*

**(b)** Simulation results for $n0 = 10$. Values of z (green), $zpow$ (orange), and out (red) are shown.

Fig. 10 Approximating Euler's number through infinite series. (Color figure online)

```
1 crn = {
2   conc[e,  1], conc[element,  1],
3   conc[ divisor , 1], conc[one, 1],
4   conc[ divisorMultiplier , 1],
5   step[{
6     div [element,  divisor , elementNext],
7     add[ divisor , one, divisorNext ],
8     add[e, elementNext, eNext]
9   }],
10  step[{
11    ld [elementNext, element],
12    ld [ divisorNext ,  divisor ],
13    ld [eNext, e]
14  }]
15 };
```

(a) *CRN++ code.*
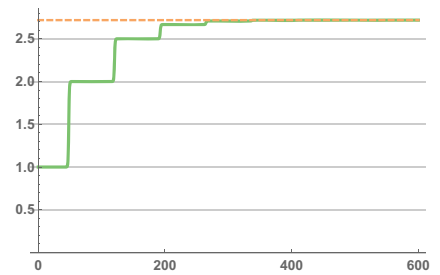
(b) Simulation results. Approximation of Euler's number is shown in green line, while dashed orange line shows the correct value as a reference.

Fig. 11 Approximating Pi constant through infinite series. (Color figure online)

```
1 crn={
2   conc[four ,  4],
3   conc[ divisor1 ,  1],
4   conc[ divisor2 ,  3],
5   conc[pi,  0],
6   step[{
7     div [four ,  divisor1 ,  factor1 ],
8     add[ divisor1 ,  four,  divisor1Next ],
9     div [four ,  divisor2 ,  factor2 ],
10    add[ divisor2 ,  four,  divisor2Next ],
11    sub[ factor1 ,  factor2 ,  factor ],
12    add[ pi,  factor ,  piNext]
13  }],
14  step[{
15    ld [ divisor1Next ,  divisor1 ],
16    ld [ divisor2Next ,  divisor2 ],
17    ld [piNext,  pi]
18  }]
19 };
```

(a) *CRN++ code.*
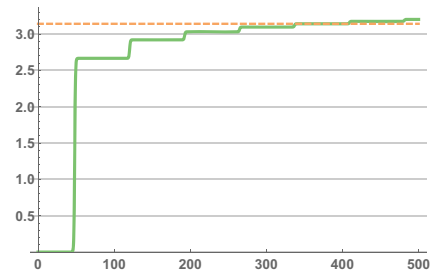
(b) Simulation results. Approximation of π constant is shown in green line, while dashed orange line shows the correct value as a reference.

other, and in general, has a higher error than the other arithmetic modules. This information is useful when designing *CRN++* programs: error-prone subtraction of inputs close to each other is the reason why the error in the program approximating π (4.1.6) is higher than in the one approximating Euler's number (4.1.5). Having this in mind, a user can optimize a program; for example, the subtraction

of close operands can often be done in alternative, less error-prone ways (see below, Fig. 13b). We plan to add runtime assertions to *CRN++* programs that alert for possible issues in the program; for example, when values being subtracted are closer than ϵ to each other.

Table 2 Size of CRNs

Program	#Species	#Reactions
Discrete counter	25	31
Factorial	26	33
Integer division	32	39
Integer square root	26	32
Euler	24	20
π	29	29

4.2.2 Reducing the error through program refactoring

The *sub* module has a high error when the operands are close to each other. In this section we show an example where the error can be reduced by replacing the *sub* module with an alternative subtraction algorithm. Figure 13b shows the alternative code for performing subtraction. The value of b is subtracted from a, by repeatedly subtracting 1 from both a and b, until b reaches 0. This method ensures smaller error which is also constant in time, however it is less time efficient.

5 Related work

Computational power of chemical reaction networks Previous research demonstrated techniques of achieving complex behaviors in mass-action chemistry, such as computing algebraic functions and polynomials (Buisman et al. 2009; Salehi et al. 2017, 2018), logarithms (Chou
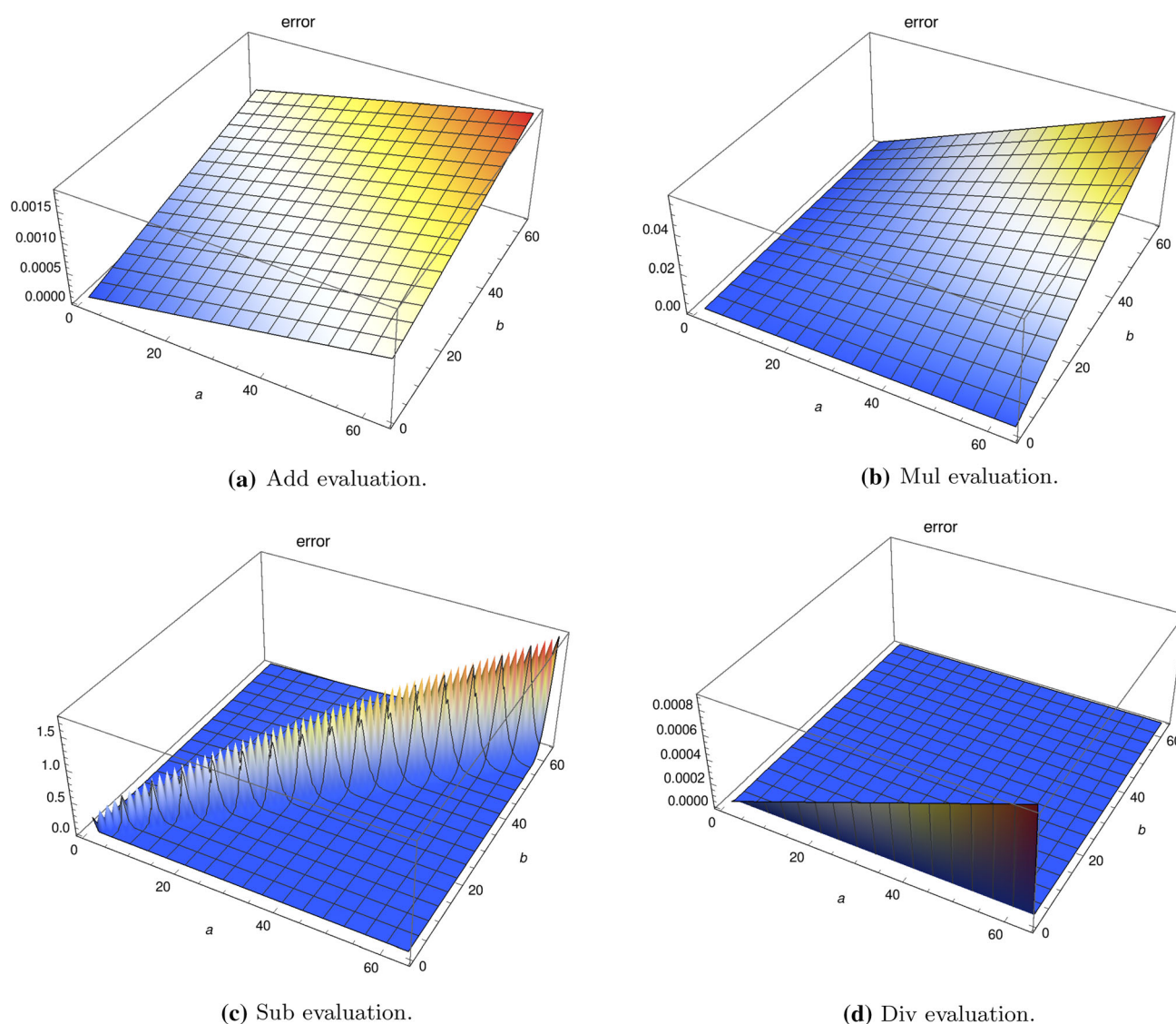

(a) Add evaluation.

(b) Mul evaluation.

(c) Sub evaluation.

(d) Div evaluation.

Fig. 12 Error evaluation of arithmetic modules. Axes a and b show the values of the first and second operand, respectively; the height shows the absolute value of the error (difference between the correct and actual value of the operation)

Fig. 13 Comparing error of *sub* with the alternative way of subtracting (**b**). Error evaluation is shown (**a**) for the cases when the operands are equal (minuend and subtrahend same), since *sub* exhibits the highest error in that case
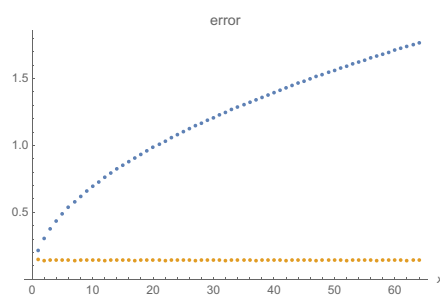
(a) Comparing error of Sub module (blue lines) and alternative way to subtract (orange lines). X-axis show the value of both minuend and subtrahend.

```
1  crn = {
2    conc[a, a0], conc[b, b0],
3    conc[one, 1], conc[zero, 0],
4    step[{
5      cmp[b, zero]
6    }],
7    step[{
8      ifGE[{
9        sub[a, one, anext],
10       sub[b, one, bnext]
11     }]
12   }],
13   step[{
14     ifGE[{
15       ld[anext, a],
16       ld[bnext, b]
17     }]
18   }]
19 }
```

(b) Alternative way to subtract.

2017), implementing logic gates and finite state machines (Hjelmfelt et al. 1992; Magnasco 1997; Ge et al. 2016), and neural networks (Hjelmfelt et al. 1991; Salehi et al. 2018). Moreover, the Turing completeness of chemistry has been proven, using the strategy of implementing polynomial ODEs (which have been previously shown to be Turing universal) in mass-action chemical kinetics (Fages et al. 2017). Even though Turing complete, this translation to chemistry can result in infeasibly complex chemical reaction networks, which motivates other, more direct methods.

Modular reactions Adding even a single reaction to a CRN can completely change its dynamics, which makes the design process challenging. The idea of "composable" reactions seeks a set of reactions that can be composed in a well-defined manner to implement more complex behaviors. Buisman et al. (2009) computed algebraic expressions by designing the core modules that implemented basic arithmetic operations, which can be further composed to achieve more complex tasks. Our goal is to make modular designs, and we follow some of the proposed design principles for achieving the goal, such as input-preserving CRNs.

Synchronous computation Previous work utilized synchronous logic to achieve complex tasks. Soloveichik et al. implemented state machines in chemistry by relying on a "rock-paper-scissors" type of chemical clock (oscillator) (Soloveichik et al. 2010). We use the same clock module, with clock species acting catalytically to order reactions. Jiang et al. (2011), also relying on clock species, designed a model of memory in chemistry to support sequential computation, demonstrating their technique on examples of a binary counter and a fast Fourier transform (FFT). Previous work shows the promise of programming synchronous logic in reactions, which we advance by providing an explicit programming language and framework for designing and testing wide-range of programs.

Asynchronous computation Huang et al. (2012) used "absence indicators" to implement complex algorithms such as integer division and GCD. An absence indicator is a species that is present in high concentration when a target species is present in low concentration (Senum and Riedel 2011). Absence indicators can be used to drive a reaction when a particular reaction has finished, providing a method for executing modules in desired order. Generally speaking, the absence indicator for species A is produced at a constant rate and gets degraded by A. The absence indicator has to be produced slowly, or else it will be present in non-negligible concentration even if A is present. The absence indicators in the literature relied on a difference between rate constants of several orders of magnitude; e.g., Huang et al. (2012) uses two reaction rates, 'fast' and 'slow', where the fast rate needs to be orders (2–3) of magnitude larger to ensure the proper function of the system. Since, in practice, biochemical systems allow for a restricted range of reaction rates, requiring a large difference in rates slows down the computation when the computation speed is dictated by the slow rates. In contrast, we allow all reactions to take the same (or comparable) rate constants.

While the goal of our work is not to compare asynchronous and synchronous computation, we mention a few insights which we gained through empirical studies. First, absence indicators are not robust, and typically require fine tuning to get the system right. Second, error detection is easier with synchronous logic. Since all operations follow the clock signal, there is a direct mapping from a time moment to a command that is executing, which provides a way to check correctness of a system at any point of time. We provide a framework for implementing molecular programs which is easily extensible, and can be used to

compare synchronous and asynchronous logic. We include support for absence indicators through the *ifAbsent* construct, thus allowing easier comparison of the two paradigms.

6 Discussion and conclusions

There are multiple ways in which we can further improve *CRN++*. Note that currently every high-level module is mapped to exactly one CRN implementing the operation. Letting the tool decide which implementation to use in different contexts could boost the performance. For example, the described modules have a useful property of preserving inputs, but that property might not be needed in every case. If the input preserving property is redundant, *CRN++* could choose to use the more optimized version (for example the more compact subtraction CRN discussed above). Also, we could provide a more flexible programming experience by (a) letting the compiler automatically schedule instructions to different steps (instead of the explicit `step` construct); (b) allowing the same species as both input and output of a module and automatically generate the additional instructions.

We plan to further explore the support for nested loops in *CRN++*. Currently nested loops can be mimicked through conditional execution: the loop condition is computed through comparison and the main loop conditionally executes the instructions of the desired loop. Besides explicit support for nested loops, future work will support nested conditionals by adding multiple flag species for multiple comparisons.

An important direction for future research concerns reducing the error in our construction, and understanding how it builds up over time. We noticed that different algorithms, even computing the same function, accumulate varying levels of error. For example, as seen in 4.2.1, the error of the Sub module increases with the magnitude of the operands, and also increases the closer they are. However, we also found an alternative way to subtract, that keeps the error constant and independent of the operands (see Fig. 13b) at the cost of a slower run-time.

Our error analysis shows that for most examples we tried, but not all, error builds up over the course of the computation. For *CRN++* programs where the error builds up in this way, there is some maximum input complexity beyond which the error overwhelms the output. Can all *CRN++* programs be refactored (preferably automatically) to bound the cumulative error of every module such that it does not build up over time? Note that if this were possible, we would obtain another, more efficient, way to achieve Turing universality.

To the best of our knowledge we are the first to provide an imperative programming language which compiles to chemical reaction networks. Moreover, we build tools that can help users get a better understanding of CRNs and improve their design. Although absolutely correct computation is not achieved, we provide tools that help understand why error occurs, and thus help improve the design of CRNs. We release our toolkit as open-source, to encourage new research and improvement of the *CRN++*, with the hope of advancing the engineering of information processing molecular systems.

In this work we advance the state of imperative programming with CRNs. It remains an important open question, however, whether fundamentally new programming paradigms are needed to fully and effectively utilize the computational power of chemistry.

References

Baccouche A, Montagne K, Padirac A, Fujii T, Rondelez Y (2014) Dynamic DNA-toolbox reaction circuits: a walkthrough. Methods 67(2):234–249

Bournez O, Graça DS, Pouly A (2017) Polynomial time corresponds to solutions of polynomial ordinary differential equations of polynomial length. J ACM 64(6):38

Buisman HJ, ten Eikelder HMM, Hilbers PAJ, Liekens AML (2009) Computing algebraic functions with biochemical reaction networks. Artif Life 15:5–19

Cardelli L, Csikász-Nagy A (2012) The cell cycle switch computes approximate majority. Sci Rep 2:656

Chen YJ, Dalchau N, Srinivas N, Phillips A, Cardelli L, Soloveichik D, Seelig G (2013) Programmable chemical controllers made from DNA. Nat Nanotechnol 8(10):755

Chou CT (2017) Chemical reaction networks for computing logarithm. Synth Biol 2(1):ysx002

CRN++ Github page. https://github.com/marko-vasic/crnPlusPlus

CRNSimulator Mathematica package. http://users.ece.utexas.edu/~soloveichik/crnsimulator.html

Fages F, Le Guludec G, Bournez O, Pouly A (2017) Strong turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In: International conference on computational methods in systems biology, pp 108–127

Ge L, Zhong Z, Wen D, You X, Zhang C (2016) A formal combinational logic synthesis with chemical reaction networks. IEEE Trans Mol Biol Multi-Scale Commun 3(1):33–47

Hjelmfelt A, Weinberger ED, Ross J (1991) Chemical implementation of neural networks and Turing machines. Proc Nat Acad Sci 88(24):10983–10987

Hjelmfelt A, Weinberger ED, Ross J (1992) Chemical implementation of finite-state machines. Proc Nat Acad Sci 89(1):383–387

Huang DA, Jiang JHR, Huang RY, Cheng CY (2012) Compiling program control flows into biochemical reactions. In:

Proceedings of the international conference on computer-aided design, pp 361–368

Jiang H, Riedel M, Parhi K (2011) Synchronous sequential computation with molecular reactions. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC), pp 836–841

Lachmann M, Sella G (1995) The computationally complete ant colony: Global coordination in a system with no hierarchy. In: European conference on artificial life. Springer, pp 784–800

Magnasco MO (1997) Chemical kinetics is Turing universal. Phys Rev Lett 78(6):1190

Perron E, Vasudevan D, Vojnovic M (2009) Using three states for binary consensus on complete graphs. In: IEEE INFOCOM 2009. IEEE, pp 2527–2535

Salehi SA, Liu X, Riedel MD, Parhi KK (2018) Computing mathematical functions using DNA via fractional coding. Sci Rep 8(1):8312

Salehi SA, Parhi KK, Riedel MD (2017) Chemical reaction networks for computing polynomials. ACS Synth Biol 6(1):76–83

Senum P, Riedel M (2011) Rate-independent constructs for chemical computation. PLoS ONE 6:e21414

Soloveichik D, Seelig G, Winfree E (2010) DNA as a universal substrate for chemical kinetics. Proc Nat Acad Sci 107(12):5393–5398