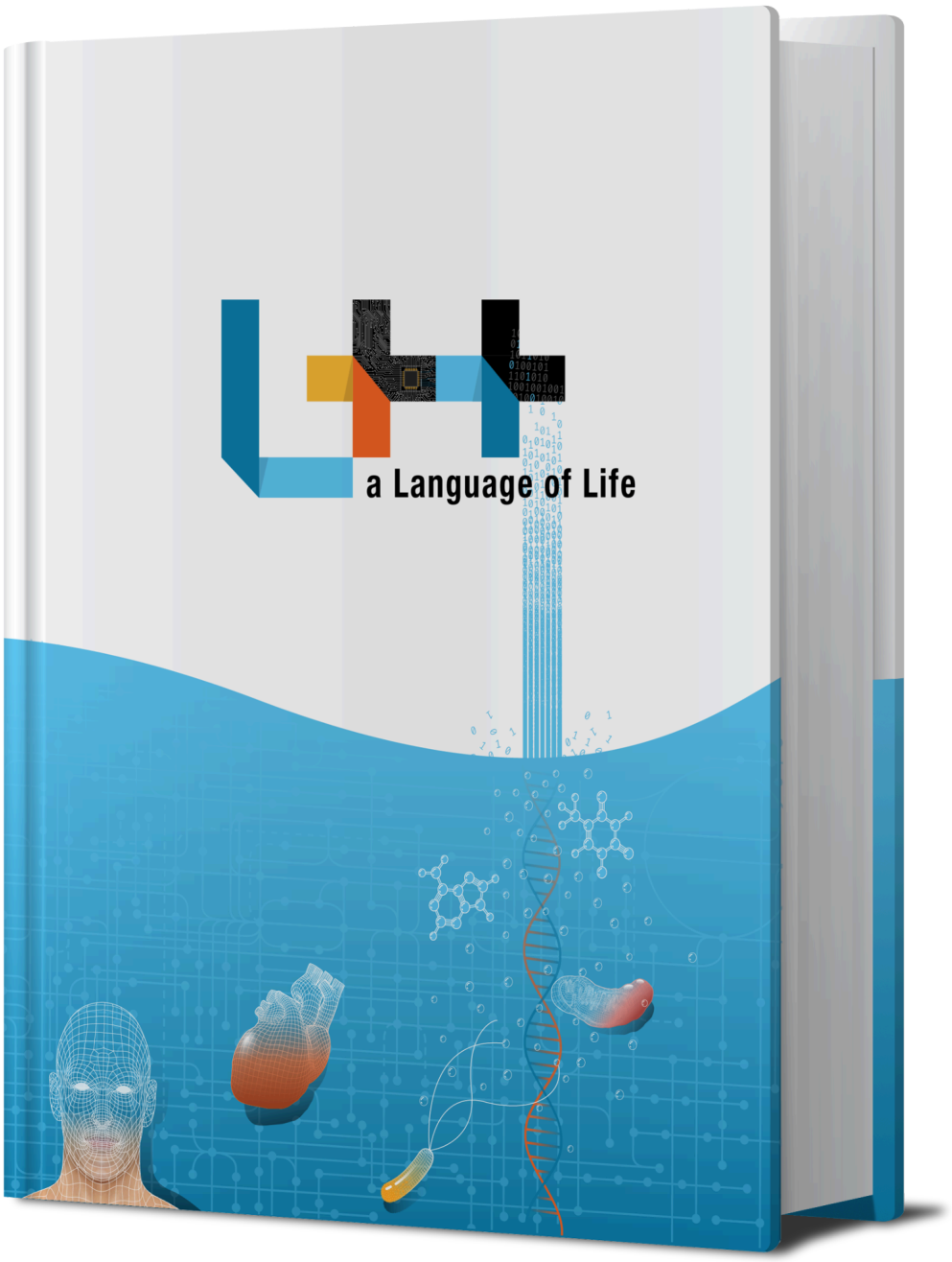
L++, a language of life



https://lplusplus.org/

Table of Contents:

Introduction

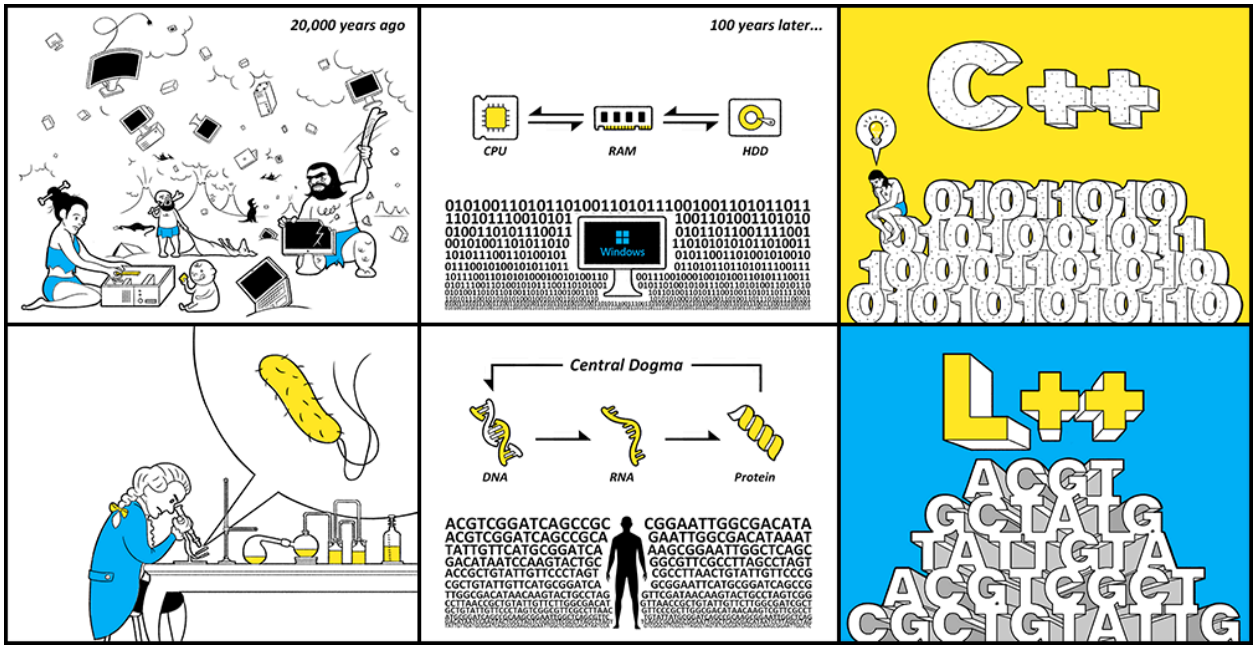
L++ is a powerful biology-oriented, high-level programming language. It is designed to abstract away microscopic interactions at a programmer's level via familiar syntaxes while providing remarkable insight into intricate simulations when necessary.

The biological interpreter is written in C and C++, which parses and transcompiles L++ experimental code into Python to run efficient matrix-based, stoichiometric data calculations to run complex simulations.

This syntax tutorial aims to comprehensively introduce a variety of fundamental data types and structures virtually found in any simulation context, whether it's high-level at an experimental level or low-level at the molecular level.

I. Biology Becoming Programming

Imagine the hypothetical scenario in which humans found thousands of computers falling from the sky around 20,000 years ago. For the sake of discussion, imagine that the computers are self powered. In many ways, this is comparable to examining a new lifeform, as they find these silicon creatures that appear to think and operate on their own. These early humans may begin trying to eat them, hit them, anything to provoke a response and better understand these strange "organisms" they have discovered. This level of experimentation and observation would continue for centuries as the humans take slow steps towards understanding these mysterious machines.



Eventually, these proto-scientists develop a more functional understanding of computers and data storage. They name parts of the computers with labels such as CPU, RAM, and HDD and notice that there is some information flowing through bits (0s and 1s) among these components to carry out computations. Also, thunderstorms might provide insight into the connections between the binary code and the computers' software program features and functions shown on monitors. For instance, one bit change (e.g., 0 to 1) could lead to the software incorrectly displaying some text or rendering the mouse unusable. In our imagined scenario, the ancient humans realize that computers are capable of understanding binary code, and people attempt to write simple programs using this machine language. This binary code (perhaps, billions of 0s and 1s) turns out to be too much and too complicated for people to efficiently understand and modify. The aspiring programmers then begin to consider a method for simplifying the machine coding interface.

This hypothetical scenario mirrors when advanced programming languages such as C++ began to be developed in real life, and the breakthroughs were world-shaping. It is incredible to consider how much computer science has changed the world over the past century, giving rise to innovations such as the internet, Windows/iOS operating systems, and other key advancements that have ushered humanity into an unprecedented era of connectivity and technological growth.

Similarly in real life, around the 1600s people began to observe microorganisms. Considering the significant changes the world has undergone as a direct result of discoveries in computer science, it is very exciting to consider how biology might follow the same trajectory of changing the world. This discovery of microorganisms was in many ways just as jarring as our hypothetical ancient people discovering computers. These antiquarian biologists began running experiments and making observations about lifeforms. As their understanding deepened, they noticed a consistent code for life, which we now called DNA. We analyzed this novel molecule and uncovered its structure and functions. We now have a more advanced understanding of DNA, how it replicates, how it is transcribed into RNA, and how that RNA is eventually translated into the proteins that make up an organism. Much like machine code, it can be difficult to manipulate DNA in a way that can lead to much innovation. We have developed some simple methods of DNA manipulation such as molecular cloning, or CRISPR technology, but we still lack a simplified method of describing life. In the same way that C++ and other high-level programming languages were developed to describe machine code more efficiently, Univeristy of Texas Southwestern Medical Center* and Omphalos Lifesciences are developing a new programming language, L++, that can act as the first high-level programming language to efficiently describe lifeforms as an alternative to DNA.

These stark similarities between the advancement of computer science and biology signal that we are on the precipice of another era of rapid world change. Everyone can see what the creation of languages such as C++ has done for the world—computers have become an integral part of each of our lives. As we harness the capacity for efficient DNA programming via L++, we anticipate revolutionary change in the world at an even larger scale than that resulting from efficient machine language programming via high-level languages.

No doubt, we are living in exciting times for the field of biotechnology. With languages and tools such as L++ on the horizon, the possible biological innovations are breathtaking to consider. In the same way that computers and programming have revolutionized our lives, lifeform programming will propel the world into a new era of innovation with advancements that were previously thought to be confined to science fiction. Indeed, it appears there is a bright future for humanity with the impending discoveries in biotechnology.

II. Two Abstractions Enabling the Precise and Unambiguous Descriptions of Organisms

At first glance any system can look like a complicated web of components; however, it's important to not get bogged down in every tiny detail. If you were given a recipe, you would only be required to know basic details like which ingredients to use, and how to prepare and heat them. You would not require instructions detailing which knife to use for the job or how to generate the heat needed to prepare your meal. It is critical that systems like these have languages that describe them in order to simplify the complex relationships within each system, and now we are using the L++ programming language to help us simplify the systems of life.

Programming life with L++ requires only that you understand the intended behavior of your molecule of interest to simulate it, the details of your molecule will come on their own. Think about turning on a television. The interface between you and the television is typically a handheld remote with well-defined buttons. You know what each button does and the intended behavior of the signal sent from that button once it is implemented by the television. Even if you do not know how the signal from the remote's "mute" button is transduced into an electrical signal that is sent wirelessly to the television to turn the sound off, you can still use that button to perform that function. Behaviors of virtual molecules and life forms will be similar to those of the remote by using high-level abstractions coded by L++ to interface with a compiler that implements low-level details about the molecule, cell, or organism of interest. A user could code a simple reaction that is meant to describe the reaction that transforms carbon dioxide into carbonic acid as CO_2 -> H_2CO_3. Although this process can occur on its own slowly, organisms use carbonic anhydrase to drive the reaction to generate carbonic acid faster. Additional parameters like these can be supported by L++ to give more specificity to a program.

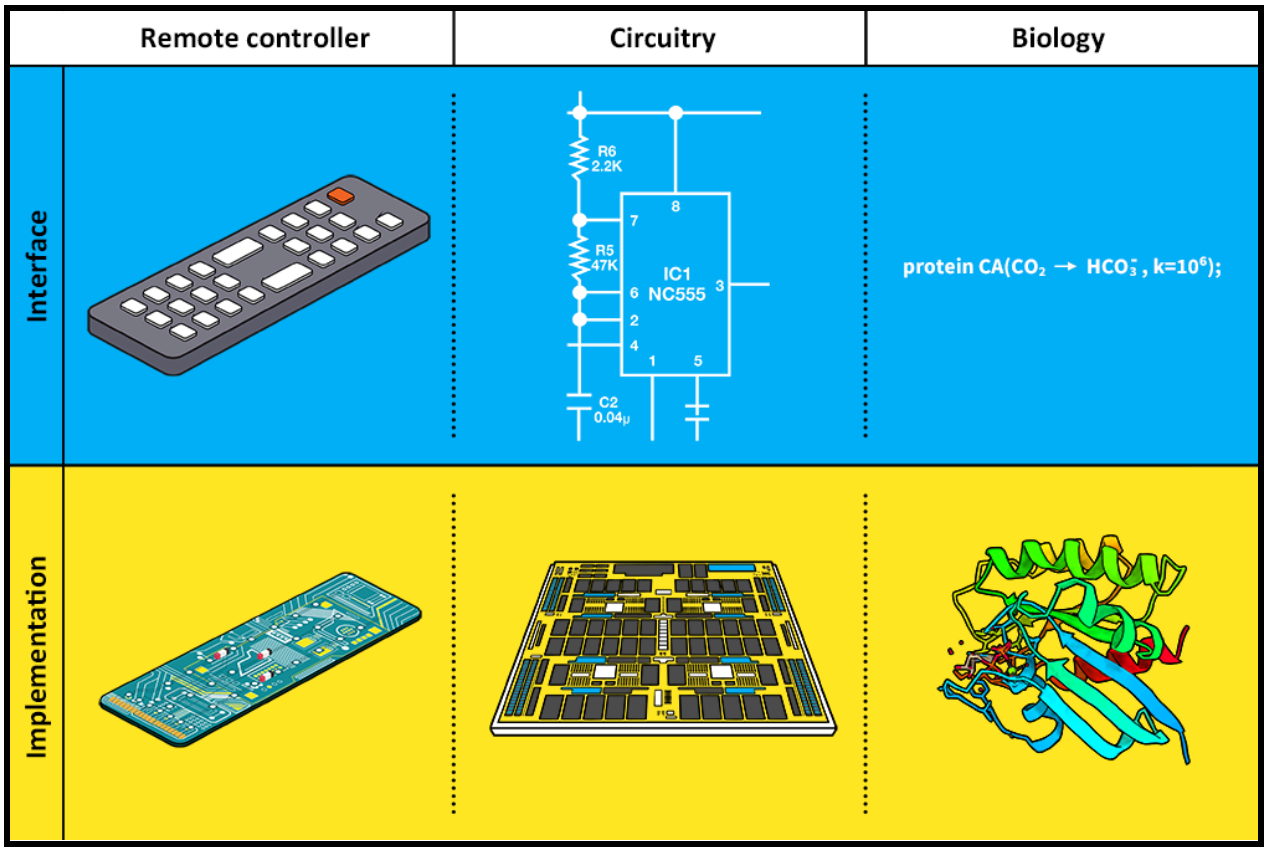


Figure 1: Interface and implementation of various devices and systems. TV viewers turn on TV or switch channels and control volumes via remote controller, without having to instruct how the details of it will be implemented to control the behavior of TV. Similarly, the circuit designer can draw schematics of the circuit, which can be manufactured by the engineers to generate such a circuit board. Using L++ as an interface between biology and programming allows us to design systems that act much like a remote control and a circuit board with short commands we can carry out detailed simulations of biosystems.

While a simple reaction may suffice for some purposes, you may want to expand on the reaction, adding parameters to describe the kinetics of the reaction, enzymes that interact with your molecules of interest, its location within the cell, or environmental conditions that may affect the reaction. All of these parameters can be described in L++ to generate thorough models of the physical properties of molecules and the reactions that they participate in. Let's take a look at a specific protein: carbonic anhydrase. Using a single line of code we can specify several important details in the interface: The enzyme, its function of converting carbon dioxide to carbonic acid, and the kinetics of the reaction. In using this basal information, the implementation of the code will provide low-level details of the enzymatic activity of carbonic anhydrase such as the zinc binding domain necessary for activation, and the carbon dioxide binding domain that is essential for the production of carbonic acid. The physical properties of molecules that can be computed using the L++ compiler (LCC) also leaves room to create models of larger pathways.

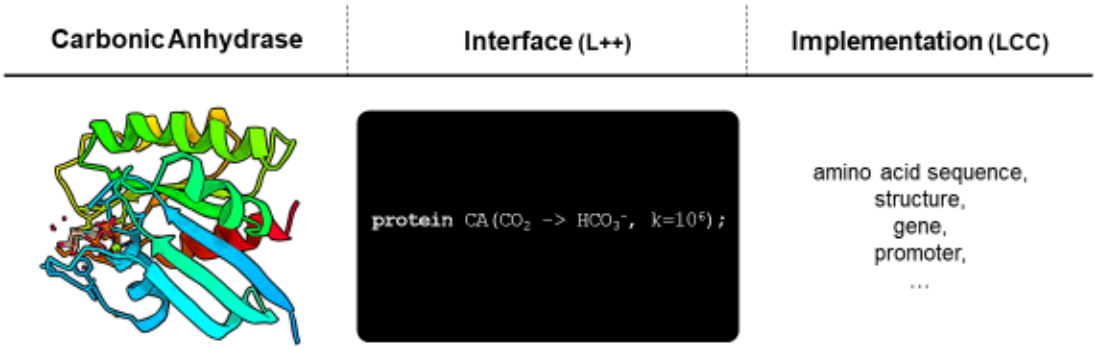


Figure 2: L++ Interface and implementation of carbonic anhydrase. Molecules in the L++ interface can be rigorously described using high level abstractions in code. The L++ compiler (LCC) with databases and structure function prediction algorithms will output detailed information about the target molecule.

The structure of a molecule should be indicative of its function. The molecules within a system should keep a spatial relationship that allows for the proper interactions between each component to carry out the purpose of a system as a whole. Take for example the actin protein that is utilized to form structures that are critical to the structural integrity of a cell. While actin exists as a monomer, it has the ability to bind to itself to form strong filamentous polymers. If we wanted to explore the growth of an actin filament, with just a simple code we could describe the growth of an actin filament from basic monomers to the more complicated helix of a filamentous polymer. The implementation of L++ code into a simulated polymer will cover the low-level details of the polymer such as the spatial relationships between each actin monomer, how the monomers join together, and how the growth of the polymer changes the shape of the filament. With the implementation of L++ code we will be able to describe essentially all biosystems.

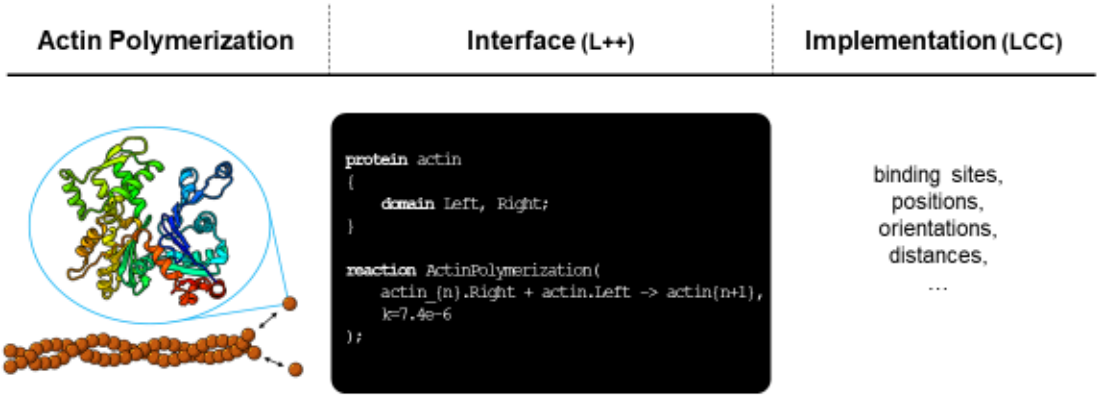


Figure 3: L++ Interface and implementation of actin polymerization. High level spatial relationships of molecules in biosystems can be described in L++, and the low level details may be implemented by the L++ compiler (LCC) with databases and capabilities of molecular dynamics simulations.

These two abstractions, descriptions of low-level details of individual molecules and biosystems from high-level language, will allow us to program biosystems at different resolutions. From as

small as interactions between individual molecules to spatial organizations of molecules interacting in a biological pathway, to whole-cell interactions, tissue formation, and at a whole organism scale. Refer to our previous blog post Transcending Time, Space, and Resources via Life Virtualization. The different resolutions of life that L++ describes will allow for expedited research and drug development through modeling drug trials on virtual humans which requires accounting for effects on organisms at the molecular, cellular, and organismal level, our first virtual organism, *E. coli,* is already being built. With requiring only a few lines of code to describe cellular processes, L++ will make molecular modeling more accessible to the scientific community.

PART I: BASICS

Chapter 1: Getting Started

In this chapter we will take a look at some L++ code as we prepare you to begin writing your own biological programs. L++ code can be written in any text editor, so it is not necessary to install a new text editor if you have one that is familiar to you; however, L++ code is processed using the L++ compiler OmVisim.

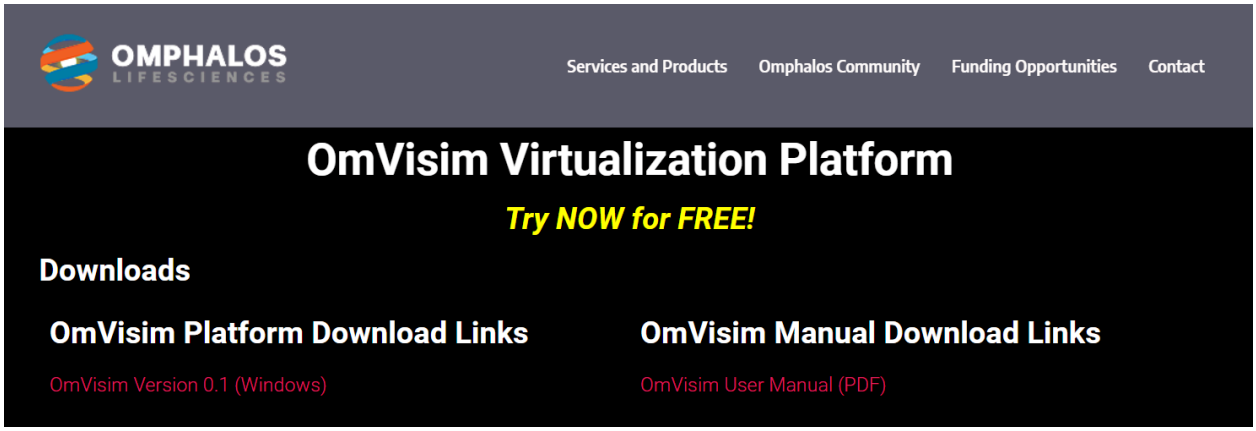
1.1 Installing and Running OmVisim

The OmVisim IDE is designed specifically to be used with the L++ language to implement L++ code into virtualized life. OmVisim installation is intuitive and simple and can be downloaded through the Omphalos Lifesciences Inc homepage.

The OmVisim package comes with a set of projects, which users can simply select and run without having to write L++ code. These included projects are publicly available through GitHub of the Daehwan Kim Lab in the Lyda Hill Department of Bioinformatics at the University of Texas Southwestern Medical Center.

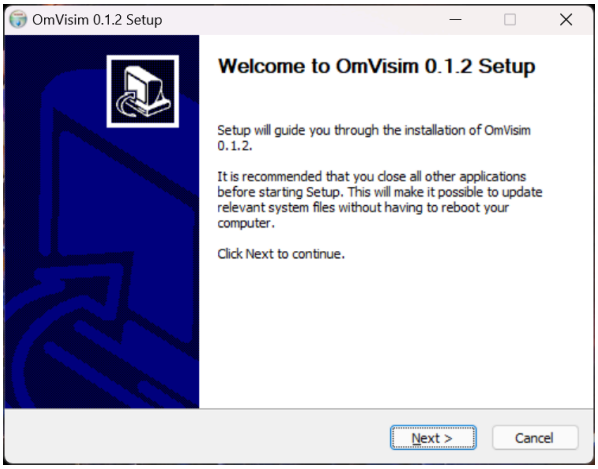
Installation

Download the OmVisim package from the Omphalos Website.

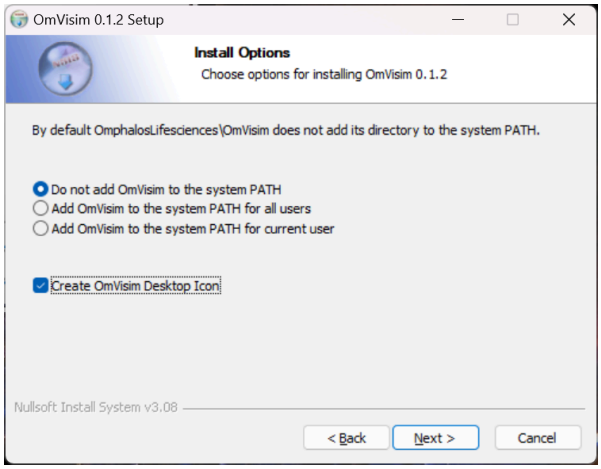


Install the packages. By default, OmVisim will be installed at `C:\Users\<UserName>\OmphalosLifesciences`.

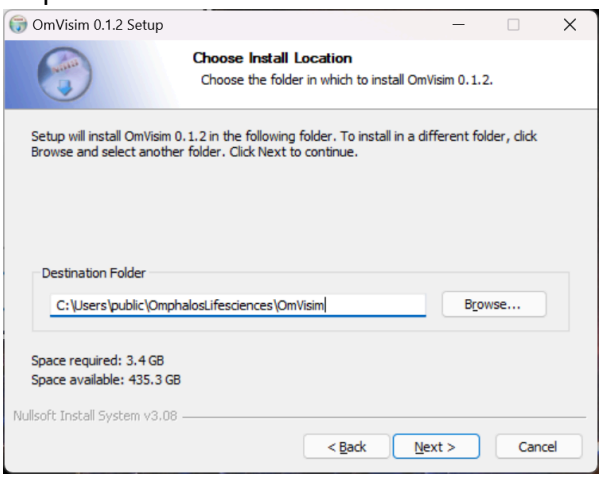
Step 1



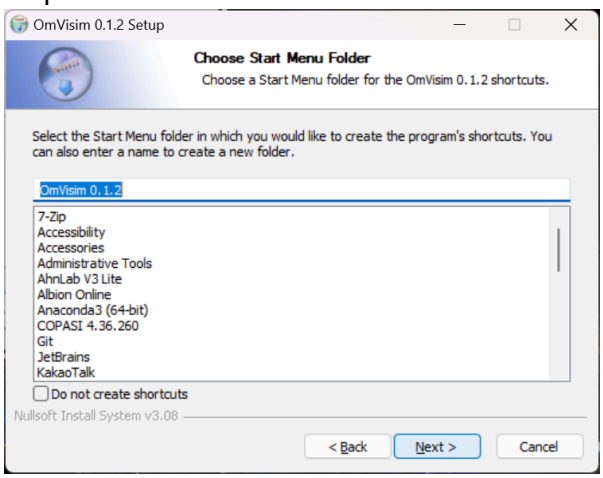
Step 2



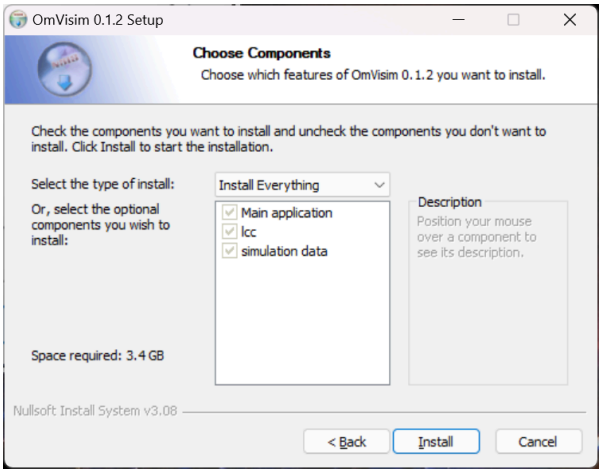
Step 3



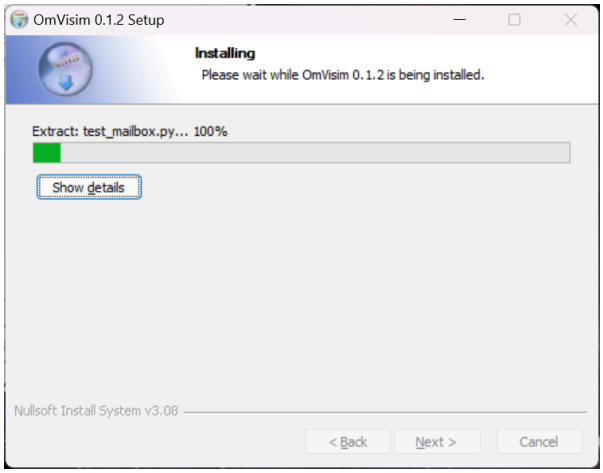
Step 4



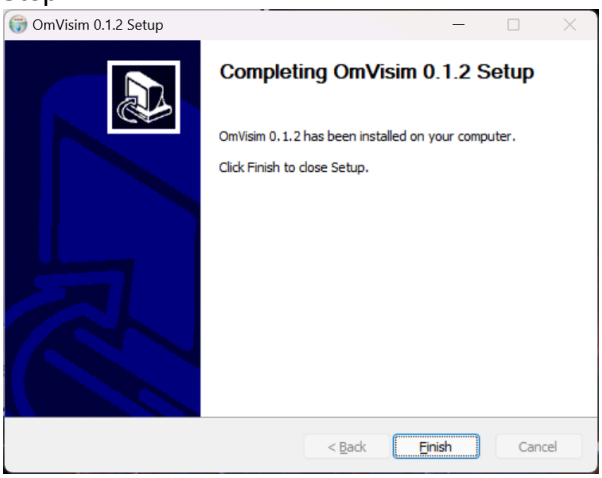
Step 5



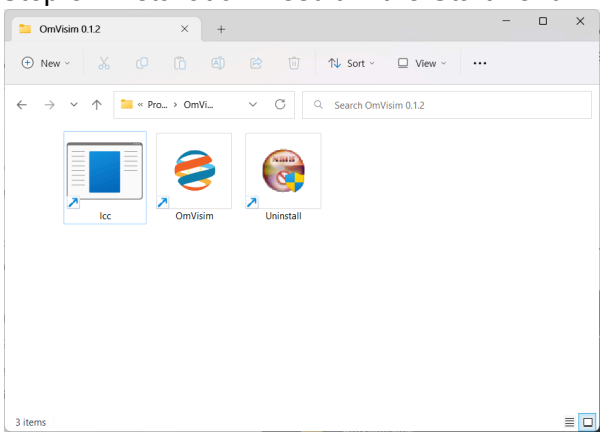
Step 6



Step 7

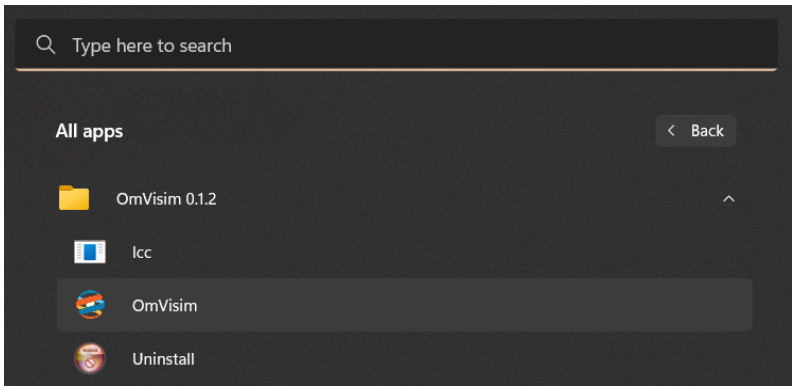


Step 8 - Installation Result in the StartMenu



Running OmVisim

Now that you have installed OmVisim you are almost ready to run your first program. First **Launch** the OmVisim app in the Start Menu.

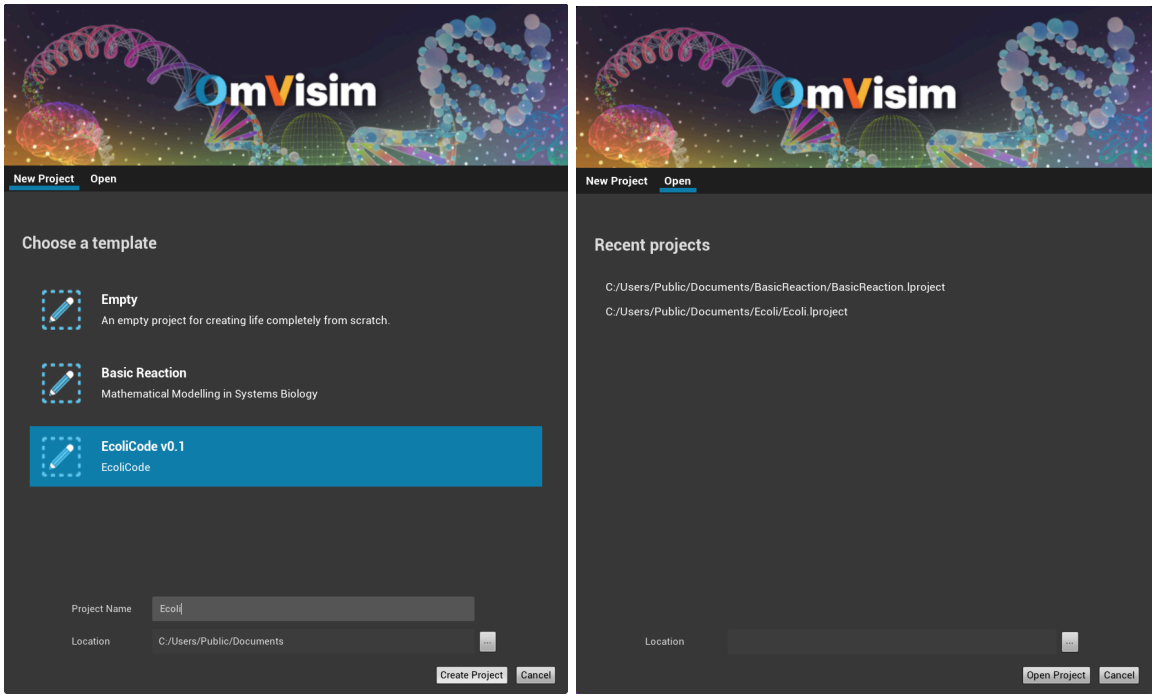


The configuration window will appear first. From here we can prepare our project.

Creating a Project

As we prepare to start our project we will find two tabs in the configuration window: **New Project** and **Open**. If we navigate to the New Project tab we can start creating one of our own, either as an empty project that you can build on your own, or if you want to start from one of the template projects available in OmVisim you can select one under the **Basic Reaction** projects or the ***E. coli*** projects. If you want to navigate back to a project, switch to the Open tab.

Once you have a project in mind, give it a name and a location in which it can be saved so you can access it later. Finally you are able to click **Create Project** and start writing.



1.2 Examining our first L++ code

Throughout this book we will connect principles of biology and programming to prepare you for designing virtual life, and whether you have a background primarily in biology or experience in programming we should not lose sight of the bigger picture as we work out examples of designing L++ code. So let's start with something simple. Consider the following reaction:

reaction.lpp

```
reaction R1(A -> B);
```

In this short line of code we are creating a reaction that we have named R1, and when this reaction is performed molecule A is converted into molecule B. Pretty intuitive, right? But life is complicated, which leaves room for us to expand on the reaction. What if we wrote:

reaction2.lpp

```
reaction R1(A -> B, k=9);
A=10;
B=0;
```

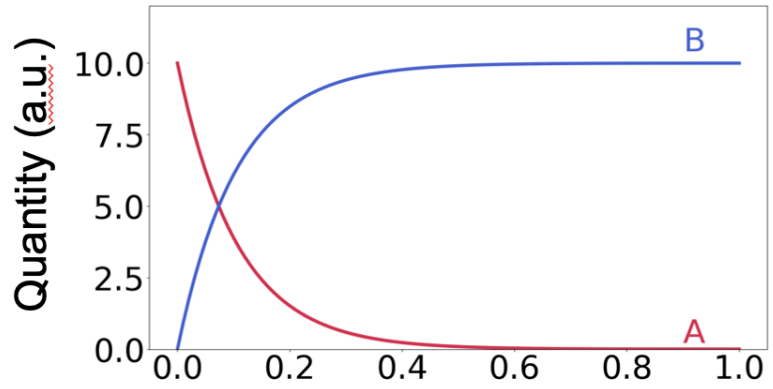


Figure 1.2.1

Here we have added two new components to our reaction. First, we have a reaction rate coefficient (k), which specifies the rate and direction of the reaction. Second, we have starting values for our molecules of interest which allows us to create models for change in the chemical concentration over time. Now we can also add another layer to our R1 reaction that lets us describe a reaction that can not only convert molecule A to molecule B, but also a reversal of that reaction that can revert molecule B to molecule A. To accomplish this we can add reversal kinetics to the reaction.

Equilibrium .lpp

```
reaction R1(A -> B, k=9, krev=12);
A=10;
B=0;
```

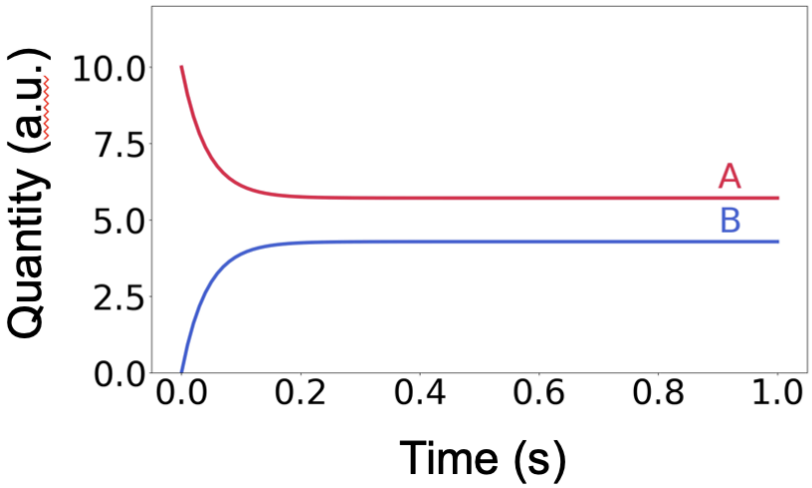


Figure 1.2.2

Now we can see how altering a single parameter in our code can affect the dynamics of our reaction. By adding a value for `krev` we made the reaction reversible and allowed for it to reach

a state of equilibrium instead of converting every molecule A into molecule B. We'll discuss how to use additional parameters in the next chapter.

Summary

In this chapter you have learned a bit about the L++ programming language, its utility, and how it is used in the OmVisim compiler. You've also prepared OmVisim and you have taken a look at your first programmed chemical reaction and the basic components of it.

In the next chapter you will learn about basic types of data that can be used in L++, how to use them, and how use operators in biological contexts to build complex biochemical reactions.

Chapter 2: Variables, Reactions, and Units

Just as analog machinery is programmed by mechanical forces generated by different components such as cogs, gears, and sprockets, programs require different forms of data to run effectively. These data can be in the form of variables that act based on the descriptive operators and units that contextualize them within a biosystem.

Variables can take many forms, but to put it simply they are just going to be containers that we can use to store data for later use. Variables in traditional programming languages include data types like integers, float values, booleans, and names such as the following:

```
int x = 1;
float y = 1.6;
z = true;
m = MoleculeA;
```

In the above example we have an integer specified as x=1, a float variable in the form of y=1.6, a true bool value to describe z, and the last variable m described by the string of a fictional molecule MoleculeA. We will cover additional details about the syntax involved in specifying these variable types soon, but first we should address some additional data types that L++ supports that are more descriptive of biosystems.

While previously we used a fictional molecule, MoleculeA, to describe our variable m, L++ also supports real molecules such as carbon dioxide, adenosine triphosphate (ATP), or any other that comes to mind. These molecules can be described using their chemical formula such as CO2 in the L++ format, or by their name for those with more complex formulas. Determinants of how molecules interact in their environment such as the chemicals they can bind to, or the concentration of each in their environment can be specified in L++ code.

In this chapter we will discuss the syntax of these variables and which operators we can use to create strings to describe interacting molecules such as reactions, pathways, molecular complexes, and other molecular components of these biosystems.

Now that we have discussed the types of data that you can use to write a program using L++, we should note that L++ is a typeless language, meaning it doesn't require data typing for more code efficiency. Data Types can be inferred in the program but also specified to fit the programmer's plan as well. In this case, both statements shown below would compile and run. Take a look at the following:

```
int x = 1;
x = 10;
```

In the example above, since `x` is initially declared as an integer (`int`), it can only be reassigned as another value of the same type. Reassigning it as any other data type will result in a compilation error. However, if `x` was never declared as an integer type, and rather just as

```
x = MoleculeA;
```

Its type depends on the context, and can therefore be reassigned to another experimental structure data type like so:

```
x = MoleculeB;
```

2.1 Syntax

This chapter comprehensively covers all the types of syntaxes needed to get your first L++ program up and running to simulate experimental data. Many examples are provided along the way to help understand how and when each type of syntax is used and how you can use them to design complex biological systems.

Operators are used to describe how data types will interact with one another in your program. Throughout this book they will be addressed in boxes such as this one. Typically code involving any operators with multiple operands should contain a space between the operator and the operands.

Comments and Simple Syntax

Similar to other languages, L++ comments can be performed by `//`, for single line comments, or `/* <multi-line comment> */` , for block-style comments. A comment is a line in your code that will not affect the program that you are writing. You can add your comments around the code in your program to leave annotations for yourself or other users that you may be collaborating with to design a program. For example a single line comment may be formatted like so:

```
// Repression of ThyA
```

A multi-line comment can also be added to your code by using `/*` and `*/` in between your comment.

```
/*
Repression of ThyA
*/
keywordname();
```

When we read text it becomes increasingly incomprehensible when a sentence starts to run on. Similarly, when we write code that we expect a computer to understand we sometimes need to add an indication that a line of code has ended and has to be interpreted differently than the code that comes after. To accomplish this we use the semicolon. Consider our example of data types from before:

```
int x = 1;
float y = 1.6;
z = true;
m = MoleculeA;
```

Each of these lines is an independent statement and thus we separate them using a semicolon so the computer understands that there is a pause between each statement.

2.2 Units

When we design experiments we have to be specific about how much of each reagent is added. If we looked at a protocol it would not read as “add 10 of molecule A, 1 of molecule B, and 0.5 of enzyme C.” A protocol like this would give us no information about how to perform the experiment, we give this information meaning by including units.

Units can follow numerical data to specify the experimental context. For example, the concentration of a solution would be given via molarity or molality. Here are a couple of examples; further references regarding units can be found in 2.7 Prefixes and Units.

```
R = 5;
R = 5nM;
B = 0.1nM;
Am = 500nM;
```

Units

Units are essential to describe any chemical, physical, or biological process. When we specify units they contain two parts: the prefix, which is optional but can be used to shorten the value of the unit. The prefix will describe the magnitude that we are measuring with the unit, while the unit describes what we are measuring.

Prefixes

Prefixes are the optional, first half of a unit. Our prefix system matches that of traditional SI Unit prefixes. The following prefixes are supported. Note that the capitalization of the prefix matters.

Capital letters will annotate larger scales while lowercase letters will be annotating smaller scales.

Kilo (k) - 10^{3}
Deci (d) - 10^{-1}
Centi (c) - 10^{-2}
Milli (m) - 10^{-3}
Micro (u) - 10^{-6}
Nano (n) - 10^{-9}
Angstrom (Å) - 10^{-10}
Pico (p) - 10^{-12}

In a laboratory environment scientists use a range of magnitudes to describe the units that apply to their reagents and organisms. With research ranging from whole organisms, which may use magnitudes between Kilos and Micro, to single cells which may be described using magnitudes between micro and pico, there is a range of units that are supported in L++ that can apply to biochemical and medical research.

L++ will support magnitudes as small as the interatomic level, magnitudes such as Angstrom and pico will also be supported. For a larger list of supported magnitudes jump to Appendix E: Supplemental Tables.

Units

With the magnitudes supported in L++ established we can look at the units. The units are the required, second half of a unit specification. The following units are supported. Note that the capitalization of the units matters. Identifiers and parameters named after the units below will result in a syntax compiler error.

Volume	**Liter** - `L`
Time	**Second** - `sec` **Minute** - `min` **Hours** - `hr`
Mass	**Gram** - `g`
Temperature	**Celsius** - `C` **Fahrenheit** - `F` **Kelvin** - `K`
Electricity	**Volt** - `V`

	Ampere - `A`
Concentration	**Mole** - `mol` **Molarity** - `M` **Molality** - `m`
Brightness	**Candela** - `cd`
Speed	**RPM** - `rpm` **G-Force** - `G`

Energy	**Joule** - `J` **Calorie** - `cal`

Some units are going to be critical to designing your own virtual experiments or organisms. Concentrations will be used in many of your programs, as they describe the quantity of a given molecule per unit of volume. Typically these start by determining the number of moles (`mol`) of a given element are contained within the unit that you are measuring. Moles will be used to measure large quantities of small items such as atoms and molecules, but when we give them additional context they will be measured using either their molarity(`M`) or molality(`m`). The molality of a substance will describe the number of moles of a solute present per kilogram of solvent, but more commonly you will see the usage of molarity, the number of moles of a solute per kiloliter of solvent. The concentration of a substance will be important when you consider how to design your reactions!

A foundational principle found in biology is the Law of Mass action which states that the rate of a reaction will be directly proportional to the concentrations of the reactants. Intuitively this makes sense, with greater concentrations there will be a greater likelihood of molecules interacting with one another in their environment. Molecules also have natural concentration ranges in certain environments, and these natural concentrations should be considered when you design your virtual life forms.

Indexing Operator: [ ]

Square brackets, commonly used in array indexing in other languages, have a different connotation in L++. Though arrays will be developed in the near future, currently, indexing is primarily used in declaring the concentration of a substance at a given timestep. Square brackets can be used directly after an identifier of a substance, with the index being the absolute timestep (seconds by default), and the assignment being the concentration to set the substance at. Here's the format:

identifier-name`[`*time*`] = conc unit`$_{opt}$

time **- Absolute Time (in seconds).** Required. Integer or floating point value.
conc **- Concentration.** Required. Integer or floating point value.
unit **- Unit.** Optional. Any SI prefix + unit is supported.

How is this useful? Initial concentrations are an example of a situation where concentrations must be declared in any given experimental context. Another example may be keeping certain substances at a fixed concentration throughout an experiment. The syntax for timesteps are similar to slicing in Python: an optional `:` may be used to specify timestep ranges.

```
Molecule[0] = 5nM;  // initial concentration at time 0
Molecule[:] = 5nM;  // fixed concentration
Molecule[1:] = 5nM; // concentration from timestep 1 & onwards
Molecule[:3] = 5nM; // concentration up to 3rd last timestep
```

2.3 Common Chemical Names

Molecules and chemical substances, both organic and inorganic, are the key components in simulating experiments. They're one of the most common data types that will be found and used across L++ code, especially within higher level abstractions and more advanced L++ data structures that will be explained further in 2.4 Advanced Molecular Structures.

L++ supports chemical equation parsing, such as $10H_2 + 5O_2 \rightarrow 10H_2O$; as such, the language supports coefficient-optional chemical formulas in order to declare specific molecular substances. The number of atoms of each element within a formula does not require a subscript; they can be directly typed out after the element. It's implied that the leading number of a molecular formula will be its coefficient. Spaces between the coefficient, elements, and subscripts are optional, and will be omitted during the compilation of the code. Here are a couple examples of simple and complex chemical formulas:

Chemical Formula	L++
H_2O	H2O
$2C_6H_{12}O_6$	2C6H12O6
$[Cu(NH_3)_4]^{2+}$	(Cu(NH3)4)2+

Traditional names of chemicals sometimes contain characters that would result in a syntax error when run as a code. One such as [6R]-5, 10-methylene-tetrahydrofolate would not be supported due to the brackets in the name; however, chemical names can also be assigned as keywords. Let's say we changed it such that

```
[6R]-5,10-methylene-tetrahydrofolate = chemical
```

Now when we need to call on this chemical we can just use `chemical` instead, but when you program your own chemicals you can use any name that is meaningful to you.

To simplify naming molecules and chemicals when you program organisms L++ comes with an imported database of millions of common chemical formulas along with their chemical synonyms. As long as the synonym is supported, typing out the synonym directly in replacement of its chemical formula is also supported. If the program fails to compile with an error at a chemical synonym, it is most likely not yet supported by the language yet. No additional libraries are required to be imported in order to use this functionality.

When we use molecule and chemicals in L++ code, there are several properties to consider. First, chemicals and molecules can exist in different states of matter which initially adds a complication when we consider how we should implement it into our programs. Thankfully we don't have to worry about that when we program life. L++ determines the states of matter of any of the reactants and products based on the context of the simulation that most matches the real-world experiment. Therefore, states of matter don't need to be explicitly declared in the reaction. Second, chemical compounds are associated with their chemical properties in the code. If we ran two lines of code written as:

```
reaction R (A + B -> C + D);
pathway Glycolysis(glucose + 2 ADP + 2 NAD -> 2 pyruvate + 2 ATP + 2 NADH);
```

Our first reaction above contains a generalized reaction of two reactants coming together to form two different products. With no chemical compounds associated with the reaction there are no properties associated with the reactants and products that could determine the kinetics of the reaction; however, in our second reaction we have reactants and products with defined chemical

properties. With the identities of the molecules in the reaction defined, L++ will assume their chemical properties from the identities of the molecules.

To create more scalable simulations that are capable of handling molecular interactions at a protein, or even organismal level in the future, declaring millions of chemical substances manually for each biochemical interaction is beyond cumbersome. Molecules and chemical substances are able to be encapsulated within more advanced data structures such as reactions, proteins, and pathways which we will get to in shortly. The object-oriented approach enables significant scalability of the language in a variety of contexts.

2.4 Reactions

Reaction Requirements

A chemical reaction is a necessary component of all biochemical pathways. With such a great importance to life you can imagine that chemical reactions will be used often in L++ programming. Now that we've seen our first reactions take a moment to look at the structure that we used above. A reaction is a statement on its own, and describes the stoichiometric relationship between different biochemical entities. Here's the basic syntax structure:

```
reaction identifier-name(equation, parameters_opt);
or
reaction identifier-name(equation_opt, parameters_opt) {
}
```

equation **- Chemical Equation.** Required. reactants -> products

parameters **- Other Parameters.** Optional. Includes constants, reaction rate, and more.

Arrow Operator: ->

Many experimental structures involve chemical reactions that describe the relationship between reactants and products. The `->` operator is used to not only separate the reactants from the products, but indicate the formation of products from reactants (in a forward reaction), and vice versa (in a reversible reaction). Every type of reaction uses the `->` operator; only the constants passed into a reaction are necessary to differentiate chemical reaction types.

```
reaction identifier-name(reactants -> products, parameters_opt);
```

A **chemical equation** is composed of reactants, products, and the relationship between them described using an arrow operator. The argument `eq =` can be used to explicitly specify a chemical equation, otherwise the equation can be directly written and implicitly passed in by the

program. The equation will typically be the first parameter in a chemical reaction declaration statement, followed by other parameters that describe the energetics of the reaction.

```
eq = <reactants> -> <products>
```

Reaction parameters describe other interactions in a chemical reaction that are not inferred from the reactants and products. One parameter to consider when designing a chemical reaction is the rate of reaction (`r`). This parameter will determine how quickly the reaction is carried out. Unless a specific unit is passed in, the default unit is 'per second', so consider the magnitudes of units that we previously talked about when you set your (`r`) parameter.

A second parameter to consider is the reaction rate constant which can be any of `k, krev, kcat, KM, Ki, Ka, Kd, nH`. A reaction's type can be specified by using an appropriate set of constants. For instance, some parameters may be used alone (`k` or `Kd`), or conjugated (`k` and `krev`), or always meant to be coupled (`Ki` or `Ka` constants with Hill's coefficient `nH`; `kcat` and `KM`). For additional information on rate constant parameters, refer to Appendix E: Supplemental Tables.

Designing a Reaction

Although we have our main components of a chemical reaction ready, there are additional ways to create more descriptive chemical reactions to better suit your life designs.

Coefficients can be used for both reactants and products to specify quantities of each in the reaction. A coefficient can be expressed through both integers and floating point numbers for more precise reaction stoichiometry. These values should precede the molecular substance that it is describing, whether it is the chemical formula or one of the chemical synonyms supported in L++. Recall that we have already seen coefficients used before when we took a look at chemical names in Chapter 2.3: Common Chemical Names and Chapter 2.4: Reactions. We saw the chemical formula for glucose describing two glucose molecules, `2C6H12O6,` and several quantities of both the reactants and products in the glycolytic pathway:

```
pathway Glycolysis(glucose + 2 ADP + 2 NAD -> 2 pyruvate + 2 ATP + 2 NADH);
```

Directionality can be conveyed using some of our reaction rate constants described earlier. Although all reaction types contain the arrow operator pointing right, you can use reaction rate constants to describe the direction of a reaction. Our reaction constant **`k`** will describe the forward direction of a reaction as reactants are converted into the products, but **`krev`** will describe the reverse reaction.

```
reaction RxnName(reactants --> products, k and krev);

// Forwards reaction (k)
// Reversible reaction (krev)
```

While it is important to convey the directionality of a reaction, sometimes biochemical reactions are controlled by regulatory elements. **Inhibition and activation** of a reaction can also be specified using parameters such as the `Ki` constant. Including this parameter will imply an inhibitory reaction of the regulator molecule in your reaction or pathway, and it is often used along with **`nH`**.

```
reaction RxnName(regulator -> reaction identifier, Ki and nH);

// Inhibitory reaction
```

Functions of molecules can change depending on where they are used. **Context dependency** can be included in your reactions that are enclosed within the protein or complex parentheses. The protein or complex (`self`) must be explicitly involved in the reaction or act as an enzyme that catalyzes the reaction.

As L++ supports a variety of experimental data structures, it's imperative to note that different declarations take on a unique set of parameters. For example, the parameter `temp` (temperature) wouldn't make sense for a pathway declaration. This section enumerates a complete list of possible parameters in L++ and their respective meanings.

L++ enables high-level abstractions of reactions without having to solely rely on biochemical algorithms, namely predefined reaction models which are often designed to describe low-level biochemical reactions involving law of mass action and kinetic parameters for enzymatic functions and molecular interactions. To generate these high-level abstractions, there are several customization options for each reaction. Each option can be expressed with a keyword and a matching code block placed underneath the reaction. As many customization blocks can be used for a given reaction. Here's the general syntax structure:

```
reaction <identifier-name>(...);
[k]
{
      <code-block>
}
```

`[k]` - Customization Keyword. Optional. Any keywords from the following are allowed.

`Specification` defines how the reaction statement should be executed without using a preset biochemical reaction model. Specifications are mathematical implementations of intended pathway designs, which are particularly useful when there's insufficient biochemical implementation available. They are numerical rules used to determine the concentration of a chemical substance at any point of time in any given reaction. Relationships between the quantities of substances are created through assignments to mathematical expressions referencing other molecular substance concentrations.

```
reaction Glycolysis(G6P + 2 ADP + 2 NAD --> 2 pyruvate + 2 NADH + 2 ATP)
specification
{
   float cg = 0.019M;
   ATP'  = (ADP / ATP) * cg;
   ADP'  = - ATP';
   NADH' = ATP';
   NAD' = - NADH';
   G6P'  = ATP' / 2;
   pyruvate' = G6P' * 2;
}
```

`ADP/ATP` represents the `ADP` to `ATP` ratio at a given time, which is used to calculate the amount of `ATP` generated at every time step. In future developments, the rate of reaction for only one of the participating molecules can be described, and the rest will be automatically adjusted according to the stoichiometry of the reaction.

Notice the apostrophes `'` following the naming of each reactant/product within the code block. This is used to describe the change of quantity in each simulation round, essentially a derivative where $ATP' = \frac{dATP}{dt}$. Furthermore, the order of each equation in a `specification{}` matters, as each following equation may depend on the concentration of a previously described relationship. Each equation can be separated via a semicolon, as they are each independent statements.

1. **`k_function` / `krev_function`.** Allows customization of forward and reverse reaction rate calculations. The return values are used to update the molecular quantities of each reaction substrate.

```
reaction FtsZ_Recruitment(FtsA~(FtsZ_{n}) + FtsZ{in} -> FtsA~(FtsZ_{n+1})); // (n
can be 0, as well)
k_function
{
   return FtsA / max(1, MinC{in});
}
krev_function
{
```

```
    return 0.5 + MinC;
}
```

In the example above, `FtsZ_Recruitment` uses two different functions to calculate the forward and reverse reaction rates.

Pathways

Synonymous with the biological term, a `pathway` describes a series of biochemical interactions that lead to some product or effect within a particular context, most commonly in a cell. Since these interactions are most commonly described through a sequence of reactions, we can use the same logic when we write pathways in L++. Here's an example:

```
pathway Transcription
{
    pathway Initiation
    {
        SigmaFactorBinding = RNAPHolo.SigmaFactorBinding;
        PromotorBinding = RNAPHolo.PromotorBinding;
        SigmaFactorRelease = RNAP.SigmaFactorRelease;
    }
    pathway Elongation
    {
        dsDNAUnwinding = RNAP.Elongation.dsDNAUnwinding;
        DNATranslocation = RNAP.Elongation.DNATranslocation;
        RNAElongation = RNAP.Elongation.RNAElongation;
    }
    pathway Termination
    {
        IntrinsicTermination = RNAP.IntrinsicTermination;
    }
    SigmaFactor = 1500, RNAPCore = 3000, NTP[:] = 9mM;
}
```

Like previously stated, the syntax and rules for reactions in general can be applied here. One can assign and reassign chemical reactions declared within pathways to previously declared chemical reactions elsewhere in our code.

As seen above, pathways aren't simply limited to containing reactions. They can contain a majority of other L++ experimental structures. Assignments for other structures, from `reactions` to `domains` to other `pathways`, can be made within `pathways` as well.

2.5 Macromolecules

Because L++ emphasizes the importance of encapsulation and abstraction, the language also aims to support higher-level molecular structures. This allows computational scientists to experiment with larger, more complex structures easily without needing to declare thousands of reactions at a microscopic level for every larger molecular entity such as a protein complex. Here are a couple of fundamental advanced structures that encapsulate some of the basic structures that have been covered.

Proteins

Proteins are a specific class of macromolecules made of amino acid residues. They include enzymes that can catalyze certain reactions and structural/functional components of a cell and generally serve as the molecular workforce of a cell. As proteins are often defined by the series of reactions they facilitate or are involved in, **`protein`** code blocks mainly contain relevant **`reaction`** declarations. If they simply encompass a single reaction, they can be declared similarly to how reactions are declared. Here's the basic syntax structure:

```
protein <identifier-name>[([reaction])][;]
```

Some proteins are meant to carry out a single, specific function; however, other proteins can perform several functions and may require other reactions to be included in its identity to carry out the functions that you require it to. A protein involved in multiple reactions should be placed in a code-block rather than a single statement like so:

```
protein <identifier-name>
[{
        [reaction];
        …
}]
```

Here are several examples of accepted protein declaration statements:

protein.lpp

```
protein CheB;
protein CheAm(CheB -> CheBP, k=0.05e9, krev=0.005);
protein CheR
{
    reaction CheA_Methylation(CheA -> CheAm, kcat=1, KM=1e-10nM);
    reaction CheAL_Methylation(CheAL -> CheAmL, kcat=1, KM=1e-10nM);
```

Let's use what we've learned so far to break this down. In our first statement, protein CheB, we have a declaration of protein CheB but no specified reaction. We've followed it with the declaration of protein CheAm, a protein that converts CheB to CheBP with given parameters for both a forward and reverse reaction. Finally we declared protein CheR and both of the reactions that it facilitates: the methylation of CheA and CheAL with appropriate reaction declarations and parameters. With the versatile nature of proteins, the code required to define a protein may look quite different for each individual protein but the basic structure will remain the same.

Membership Operator: .

The `.` operator is important in accessing specific data stored inside these abstractions. It functions the same way as the dot operator in languages like Java, C, and C++. Let's walk through an example of how it can be used:

```
protein p
{
    reaction r1(...);
    reaction r2(...);
}
```

A `protein` is instantiated in a code block, and the identifier for it is `p`. `p` is used to refer to this object in other places in the rest of the code. The `protein` takes in two `reactions` that describe `p`: `r1` and `r2`. These `reactions` may be accessed using the membership operator, such as `p.r1` and `p.r2`.

The membership operator is integral in high-level languages to describe simulations that become large quickly. Furthermore, it can be used to refer to specific structures located in imported files. Here's an example:

```
import Divisome; // .lpp file name, contains complex ZRing structure
ZRing = Divisome.ZRing; // accessing complex declared in Divisome.lpp
FtsZ_Recruitment = ZRing.FtsZ_MembraneRecruitment; // accessing reaction declared in
ZRing complex declaration
membrane Membrane;
Angle = Membrane.phi; // accessing intrinsic angle property of membrane variable
```

Nucleic Acids

Nucleic acids are used in biological systems in many ways, but they are most known for being the carriers of genetic information in a cell. Nucleotides are the building blocks of nucleic acids and encode the genetic information that will be used to manufacture proteins.
Simulating cells and entire organisms can largely be expressed through genes. Though DNA is one of the major structures, L++ supports a variety of other genetic-related structures that help describe the functional behavior of these abstractions.

DNA

Deoxyribonucleic acid, also called DNA, is a polymer - a naturally-occurring substance composed of macromolecules. In the case of DNA, the macromolecule refers to a nucleotide structure made of a nucleotide base (Adenine (`A`), Guanine (`G`), Cytosine (`C`), Thymine (`T`)), attached to a backbone composed of a five-carbon sugar and a phosphate group. Sequences of nucleotides are formed to create a singular strand, sometimes referred to as a template strand. DNA's double-helix structure consists of the template strand and a complementary strand containing opposing nucleotides for each nucleotide in the template strand. Here's a look at the structure of a sequence.

```
Template Strand:          CTGGCACNNN
Complementary Strand:     GACCGTGNNN
```

As seen in the example above, each strand can be expressed through the nucleobase that the nucleotide contains. Each base pair in a strand of DNA consists of a pairing between a purine and a pyrimidine base. Adenine, a purine, matches with the pyrimidine Thymine while Guanine, another purine, matches with the pyrimidine Cytosine through hydrogen bonds that form between the molecules. The more familiar you become with DNA, the easier it will be to recognize molecular properties of a sequence given what nucleotides make up the sequence such as recognizing binding sites, satellite DNA, or even just inferring the strength of the sequence from the nucleotide content.

DNA will be at the root of many biological processes, whether you are interested in the etiology of a biological process or disease at the genetic level, the regulatory changes that occur during development that require differential expression of genes spatially and temporally, DNA structure and organization, and other processes that we could discuss using an entirely new book. The function of DNA in your program will depend on the scope that you are capturing in the program.

It's important to note that many biological reactions can often change the state of a sequence. Sometimes, and quite commonly, certain nucleotides in sequence can't be determined until after a specific reaction occurs. To support this idea, L++ uses `N` to express the idea of an unknown nucleotide. Here are two examples:

Dnaori.lpp

```
DNA Ori;
reaction OriBinding(DnaA + Ori -> DnaA~Ori, Kd=1e-9);

reaction OligoSynthesis(DNAP~ssDNA.N{n-10:n} + c(dNTP) -> DNAP~dsDNA, r=1000);
```

First we have an example of code designating the origin of replication on a strand of DNA, binding DNA to the origin of replication with a parameter specifying the energy required for the components to dissociate. In the second example we have a line of code to simulate DNA synthesis, combining DNA Polymerase with a single stranded DNA. In this example, `N` in `ssDNA.N{n-10:n)` refers to the stretch of 10 nucleotides. Here we use `N` in place of a DNA sequence because when carrying out a process like DNA synthesis, L++ will fill in the complementary nucleic acids for whichever template strand you specify.

Length Operator: _{ }

The `_{}` operator is used to access the length information, that is the quantity of building blocks, of polymers, such as DNA, RNA and protein, and multimeric protein complexes. Compared to the positional operator, the length operator does not concern the identity of the building blocks whatsoever. Here's the format:

`<polymer-structure>_{<length>}`

`<length>` - Length. Required. Specifies the length of the polymer structure to be referenced.

Polymer elongation reaction is a type of reaction that can be efficiently described with the length operator. Here's an example of RNA polymerase extending the nascent RNA using ssDNA as a template:

```
complex RNAP // self = RNAPCore~RNA~ssDNA
{
	reaction Elongation(RNAP.RNA_{n} + c(NTP) + self.ssDNA.N{n} -> self.RNA_{n+1} +
PPi + self.ssDNA.N{n}, r=60);
}
```

RNA

Though genetic information is encoded within DNA, another nucleic acid is frequently found in biological contexts for other purposes. Ribonucleic acid is again a polymer and basically shares the same structure as DNA; however, Uracil (`U`) replaces Thymine (`T`) as one of its possible nucleobases, and like Thymine it is complemented by Adenine. Furthermore, RNA is single-stranded. It carries genetic information to perform its main purposes, which involve facilitating the coding of proteins via translation, for example.

Generally RNA has three main classes that perform different functions throughout a cell. Messenger RNA (mRNA) is used to translate genetic information into a protein at ribosomes. Transfer RNA (tRNA) typically serves as a means by which mRNA can interface with amino acids as proteins are constructed, they act as a physical link between the mRNA and amino acids as additional amino acids are brought to the growing amino acid sequence. Lastly we have ribosomal RNA, RNA that is the primary component of ribosomes at the site of protein synthesis. Here's an example of coding with RNA:

Rnasynth.lpp

```
RNA tRNA;
complex AA_tRNA;
protein AA_tRNASynthetase(tRNA + AA -> AA_tRNA, kcat=1, KM=1e-9);
```

Similar to how amino acid sequences aren't forced to be declared for proteins to describe their functions, L++ doesn't require DNA or RNA to be assigned nucleotide sequences as long as they are assigned functions. Assignments to specific sequences and relevant features will come with development in the near future.

Positional Operator: { }

When describing reactions involving macromolecules, precise referencing to specific locations of these structures may be needed. The positional operator enables positional specification of the building block along polymers, such as primary sequences of DNA, RNA and protein, and an individual protein in multimeric protein complexes. The `{}` operator is mainly used with DNA and RNA sequences in this format:

`<polymer-structure>[.<ss>]{<p1>[: <p2>]}`

`[.<ss>]` - Substructure. Optional. Accepts a single letter, which either represents a nucleotide base (`A, G, C, T, U`) or an amino acid residue (`G, A, V, L, I, M, P, F, W, S, T, Y, N, C, Q, D, E, K, R, H`).

For DNA/RNA, used to access nucleotide(s) at position/range. **`<ss>`** can take on any nucleotide base. Can access a stretch of nucleotides if a range is used instead of a number for the index. Example: `DNA.A{243}`

For proteins, used to access amino acid residue(s) at a position/range. **`<ss>`** can take on any amino acid residue. Example: `Protein.H{354}`

Usually, these represent too high of a resolution. Instead, use a general unspecified `{n}`th position sequence, which is `N` for nucleotides or `X` for amino acid residues. The unspecified sequence is only needed when referenced from other parts of the reaction.

<position1> - **Position 1**. Required. `{<p1>}` indicates a specific position p1, `{:<p1>}` indicates from the start to position p1, `{<p1>:}` indicates p1 to the end. The latter two formats cannot be used alongside `[:<p2>]`.

[: <p2>] - Position 2. Optional. Specify inclusive range from position 1 to position 2.

Here's an example of how `{}` can be used in the context of RNA translocation during transcription:

```
reaction Translocation(RNAPCore~ssDNA{n} -> RNAPCore~ssDNA{n+1}, r=10);
```

In the example above, the `RNAPCore` binding to the nucleotide at the `n`th position of `ssDNA` has been replaced to the nucleotide at the `n+1`th position along `ssDNA`, resulting in a translocation of `RNAPCore` on its bound `ssDNA`.

Complex

Proteins often form together to create stable protein complexes via non-covalent molecular interaction. This provides modular regulation of molecular processes, as some processes can only be performed when activated or repressed by specific protein complexes. The `complex` syntax is largely interchangeable with `protein`. However, this data structure allows for shortcut complexation reactions for forming itself by specifying the subunits of the complex along with the kinetic parameter `Kd`.

Complex.lpp

```
protein RpoB, RpoC, RpoA, RpoZ;
complex RNAPCore(RpoB + RpoC + RpoA + RpoZ, Kd=1e-10);
complex RNAPHolo
{
    reaction SigmaFactorBinding(RNAPCore + SigmaFactor, Kd=1e-7);

    // '--> self' is assumed in the reaction argument
    reaction PromotorBinding(self + dsDNA.promoter -> self~dsDNA.promoter);
}
```

Complexes inherit information from their different protein subunits to access their domains (for proteins) and motifs (for DNA and RNA). If no reaction is passed in, the reaction arrow (`-> self`) is made as default. Further development of L++ will enable additional, exclusive features that differentiate complexes further from proteins.

Binding Operator: ~

Molecular interactions often lead to non-transient binding between the molecules. L++ uses `~` to represent that two biochemical structures are bound together. These structures are commonly proteins, domains, and the other molecular structures used to represent biochemical abstractions.

```
reaction SigmaFactorBinding(RNAPCore + SigmaFactor --> RNAPHolo, Kd=1e-7);
```

As previously mentioned, other L++ keywords like `self` can be used in conjunction with the binding operator. A common example, thus, is such that a protein, reaction, or any other structure referred to by `self`, can be bound to another declared biological experimental structure to express a merged state between yourself and another entity. Here's an example:

```
reaction dsDNAUnwinding(self~dsDNA~RNA{n-10:n+10} --> self~ssDNA{n-10:n+10} +
ssDNA{n-10:n+10}, r=1);
```

Here's an example of the multi-binding property of the operator:

```
reaction 70S_Dissociation(30S~50S~mRNA~tRNA~RRF --> 30S + 50S + mRNA + tRNA + RRF,
Kd=1e-9);
```

2.6 Structural Components: Cytosol and Membrane

Living matters are physically and functionally compartmentalized. The simplest organization is to have an aqueous compartment (`cytosol`) encapsulated by lipid bilayer (`membrane`), physically separated from the environment. L++ currently supports these two key structural components with `cytosol` and `membrane`.

Generally speaking, `cytosol` harbors large scale reactions driven by high abundance of molecules, such as metabolites. An example of a metabolic pathway occurring in the `cytosol` is glycolysis and protein biosynthesis.

On the other hand, `membrane` is often used to elevate local concentration of low abundance molecules to facilitate their intermolecular interactions. These structural code blocks can contain any combination of sequence of `reactions`, `pathways`, or other molecular abstractions. Here's an example:

```
import GlucoseTransport;
import Glycolysis;
membrane CellMembrane
```

```
{
   GlucoseTransporter = GlucoseTransport.GlucoseTransport[Use a specific name];
   cytosol Cytosol
   {
       Glycolysis = Glycolysis.LowResolution;
       G6P = 8.8mM, ATP = 9.6mM, ADP = 0.56mM, NADH = 8.3mM, NAD = 2.6mM;
   }
}
```

Notice how a `membrane` also contains molecular structures that can be found within a cell (like `cytosol`). Just like how the cellular `membrane` wraps around `cytosol` to form the shape of a cell, the `membrane` in the code above acts as an outer scope, encapsulating the `cytosol` existing within the `membrane`.

By now, it is quite clear that there does not exist a clear top-down hierarchy of these molecular structures. With the complexity that comes in a biological cell, molecular structures can contain, and simultaneously be contained within, other molecular structures.

Syntax for structural components will substantially expand in the near future to accommodate high resolution spatial simulations and allow for further compartmentalization of biological processes. As we develop more eukaryotic code we will have to accommodate for additional membranes surrounding the organelles of the cell, structures that perform highly specific and necessary functions.

When we consider the spatial location of molecules, there are many times in which a molecule will have to be moved across a membrane. In essence this is shuttling the molecule from one container to another, but we can use keywords to designate where the molecule will be located such as {in} and {out}. These spatial arrangements can also be used to designate where a molecule is relative to the cell, regardless of if it will be found within the cell's membrane or not.

Summary
In this chapter you learned about some of the types of data supported by L++ and the units that contextualize them in a realistic environment. You have learned how to define reactions in L++, how to use parameters to further describe those reactions, and how to define major macromolecules in L++ to use in reactions, pathways, and complexes.

In the next chapter you will learn how to use the scope of a program, namespaces, and the import function to organize your programs and allow for simplification of designing complex systems.

Chapter 3: Scope, Namespaces, and Modules

So far we have mostly discussed reactions, molecules, and complexes, and while these biological processes and objects will be major components of your L++ programs, life is vastly more complicated than just single reaction inputs and outputs. As you design more complex organisms and biological systems that rely on layers with differing activity it will be vital to keep your program organized.

Here you will learn about several methods for keeping track of your program's components as you build it and other methods of organizing your virtual organism. We will discuss how you can use scopes in your code to convey temporal and spatial separation in biosystems, and how you can utilize namespaces and module importing to keep your code organized as you build higher-level biosystems and organisms.

Figure 1. Microorganism communities are present at many levels in life, both as collaborating cells that form tissues or communities composed of many different species.

```
import GlucoseMetabolism;
```

```
import GlucoseTransporter;

organism Prokaryote;
{
    Glycolysis = GlucoseMetabolism.Glycolysis;
    PTSComplex = GlucoseTransporter.PTSComplex;
}
Prokaryote Ecoli;
Prokaryote Paeruginosa;
Prokaryote Vcholerae;

Ecoli.G6P = 7.9mM;
Paeruginosa.G6P = 6.2mM;
Vcholerae.G6P = 5.8mM;

petridish P
{
    glucose = 10mM;

    Ecoli(0, 0, 0) = 100;
    Paeruginosa(0, 0, 0) = 10;
    Vcholerae(0, 0, 0) = 2;
}
P = 1;
```

The above code highlights a few different biological and programming concepts discussed in this chapter: **Scope**, **Namespace** and **Modules**. The code describes a type of organism that here we will call `Prokaryote`. These organisms contain mechanisms for glucose metabolism, `Glycolysis`, and transport system, `PTSComplex`, that are imported from external modules. The program further defines three different types of Prokaryotes as `Ecoli`, `Paeruginosa`, and `Vcholerae` and sets a `G6P` concentration in each. The three prokaryotic species are growing in a petri dish, `P`, containing glucose, and are instantiated at some spatial coordinates.

You can already see that we have several layers to this program: an environment, the organisms in the environment, and the molecules that will be affecting the behavior of the organisms. So let's take a look at how we can use programming concepts to define layers of living systems.

3.1 Scope

When we think about how living organisms are organized, they require both separation between themselves and their environment, and in eukaryotic cells there is additional separation between the organelles of a cell and between other bodily structures such as tissues and organs in order to grant structures specificity. Additionally, many biological processes are time-gated and require a distinction on when certain events must occur to properly carry out the pathway. We can give our virtual life forms both spatial and temporal separation by using **scopes**.

In a biological sense, scopes can be used to define separation between two components of your program; however, in a programming sense scope will be used to give more structure to your code, allowing you to use similar names or identifiers at different scopes for different contexts.

```
cytosol Cytosol
{
    pathway Glycolysis(glucose + 2 ADP + 2 NAD+ -> 2 pyruvate + 2 ATP + 2 NADH)
    specification
    {
        float capacity_factor = 0.03;
        ATP' = (ADP / ATP) * capacity_factor ;
    }
    pathway PyruvateOxidation(pyruvate + 2 NAD+ + CoA -> acetyl-CoA + NADH + CO2)
    specification
    {
        float capacity_factor = 0.01;
        acetyl_CoA' = capacity_factor;
    }
}
```

We designate the bounds of a scope using `{ }`. Take another look at the example above. Our code begins with a spatial context for our reactions in the `Cytosol`. We then initialize a new scope within the cytosol using `{ }` that contains our **`pathways`** `Glycolysis` and `PyruvateOxidation`.

Recall our Reaction set up from section 2.4, we use this same scope to designate the specification of the reactions. We then use a new scope within our **`specification`** keyword to set specific factors of each reaction.

Because the scope refers to the coding region of an object, when you want to call on that object in your program you can only refer to it within its scope. In our above example both reactions are encapsulated within the scope of the cytosol; however, the `capacity_factor` in each

specification is unique and do not influence each other for they are found in two parallel scopes of code. Hence, the `capacity_factor` in the latter pathway will not overwrite the first pathway. On the other hand, the two pathways (`Glycolysis and PyruvateOxidation`) are simultaneously processing the same molecules (`pyruvate`, `NAD+` and `NADH`) in the same physical scope `Cytosol`.

```
// global variable
cytosol Cytosol
{
    // local variable
    pathway Glycolysis
}
```

The scopes that you will use in your program can be largely classified into two categories: **`global`** and **`local`.** Global scopes will house your local scopes, as seen in the previous example and the above code, we can use a global scope that we are designating as the cytosol to house our reactions that will take place in the local cytosolic scope.

3.2 Namespaces

We use names to give meaning to ideas and objects in our lives. Similarly in programming languages we assign names to objects to give them meaning within the program. These names are stored within **namespaces.**

As you design your programs in L++, you will find that using **namespaces** will be a valuable tool in your arsenal, and they will allow you to keep your program organized as you import additional modules. A namespace refers to the name, some identifier for an object, and the space in which it is stored, typically referring to its scope. As such, namespaces are often arranged in a hierarchical manner, think of the namespace like the files on your computer.

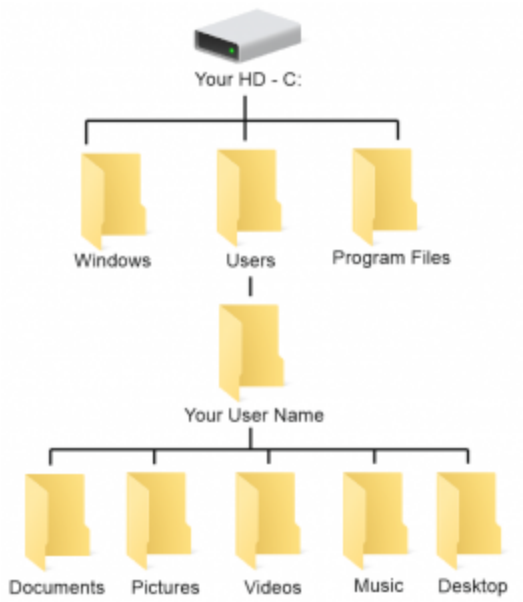


Figure 2. Namespaces and scope are used together to give programs a tiered structure like the organization of a computer's data.

Similar to a file organization, the namespace will include tiers of other objects beneath it that can all be referred to by calling on it. In the example above, `documents`, `pictures`, `videos`, `music`, and `desktop` would all fall under the namespace of `Your User Name`.

Let's suppose that we want to write a program that will give us a simulation of `Glycolysis` in three different species:

```
organism Prokaryote;
{
    Glycolysis = GlucoseMetabolism.Glycolysis;
    PTSComplex = GlucoseTransporter.PTSComplex;
}
Prokaryote Ecoli;
Prokaryote Paeruginosa;
Prokaryote Vcholerae;

Ecoli.G6P = 7.9mM;
Paeruginosa.G6P = 6.2mM;
Vcholerae.G6P = 5.8mM;
```

Here we have an organism that we are calling `Prokaryote` that will carry out a glycolytic pathway; however, we have three unique species of prokaryotes that will all contain this

pathway. Each one will require an initial concentration of Glucose-6-Phosphate, `G6P`, to carry out the reaction, but if we provided only a single concentration of G6P, the simulation would not be as meaningful as it would apply the same concentration to each species. In these situations, assigning a value beneath a namespace will help us out.

Here we can use each of our species as a namespace and assign a unique concentration of G6P to each using our membership operator: `Ecoli.G6P`, `Paeruginosa.G6P`, `Vcholerae.G6P`. Now each `Prokaryote` species in our simulation will have a unique concentration of G6P associated with it, which will not only save you time if the computation is confused by an ambiguous value, but it will also give more meaning to your simulation.

As you expand on programs like this and include other factors, you will find it simpler to import modules than to write every pathway in the same code.

3.3 Modules

The **modular programming** concept beautifully reflects how biology works. You will find that a piece of well written L++ code you wrote today may be a valuable module for you or other L++ developers to conveniently recycle into other projects. Consider applications in which biological programs can be recycled into other projects. In the natural world orthologous genes perform similar functions to one another, or how pathways involved in development are re-programmed for regeneration or sometimes hijacked in cancerous cells for rapid growth.

Yet there is a practical reason to promote modular programming practice in writing L++ code for writing large-scale biosystems. Living organisms possess complex biosystems that involve hundreds of metabolite species, thousands of unique proteins encoded in their genomes, and a multitude of reactions and molecular pathways that execute unique and interesting biological functions. Describing such large biosystems requires writing a book-worth of well-organized L++ source code. It would be a nightmare to write a human code and all of its detailing information in a single .lpp file.

To this end it is important to design your L++ code in a general way such that different parts of the code may be independent of each other. Writing modular code will allow for quick and easy utilization of a program in various contexts, rather than digging through code that you have already written when you need to use a similar biological code in another context. L++ provides an intuitive import and namespace system that enables modular integration and biologically meaningful organization of L++ code modules.

Modules in traditional programming languages
In programming, a collection of related functions/classes or data that together provide useful service to a program. A module needs to be imported into other code or programs to allow access to its functions, classes or data.

When a program imports a module, it automatically includes the module's source code and any associated resources (e.g. images, scripts, stylesheets). The imported module is "visible" to the program, but the program doesn't have to use all of the module's code.

Synonyms of a module include a package and a library, etc.

Now let's take a moment to look at how we can use modular programming to design programs in L++. Suppose that you are writing a code for cellular metabolism and want to use a glycolysis pathway previously written in *Glycolysis.lpp*.

```
pathway Glycolysis(glucose + 2 ADP + 2 NAD+ -> 2 pyruvate + 2 ATP + 2 NADH)
specification
{
    ATP' = (ADP / ATP) * 0.03;
}
...
```

To use an already written code you can import it! Importing the `Glycolysis` pathway from the `GlucoseMetabolism` module requires a simple **`import`** keyword followed by the module name and a semicolon.

```
import GlucoseMetabolism;
```

Intuitively, the objects and variables within the `GlucoseMetabolism` module can now be selectively accessed using the module name in the namespace followed by a membership operator `(.)` and the identifier of the objects or variables to access, similar to how we accessed the properties of objects earlier when we discussed importing modules.

Now, if we want to further expand on a program with a focus on cellular metabolism, we might want to provide our virtual cell with glucose that it can metabolize. Now if our cell is not photosynthetic we have a problem, it won't have any intracellular glucose to metabolize. There's a quick fix! Import another module to transport glucose into the cell as such:

```
import GlucoseTransporters;
import GlucoseMetabolism;
membrane CellMembrane
{
   GlucoseTransporter = GlucoseTransporters.PTSComplex;
   cytosol Cytosol
   {
       Glycolysis = GlucoseMetabolism.Glycolysis;
       G6P = 8.8mM, ATP = 9.6mM, ADP = 0.56mM, NADH = 8.3mM, NAD = 2.6mM;
   }
}
```

Here we imported two modules! First note that we import a module for `GlucoseTransporters` that will provide a context for glucose to be transported across the cell membrane prior to glycolysis. Next we imported `Glycolysis`, giving us the process that we are interested in simulating within the cytosol. You can import as many modules as you need to in order to build your program, and once you have a module imported you can piece it together with other modules.

Note that any L++ code may conveniently import any other *.lpp* files. Thus, the available *.lpp* files will grow as additional L++ users design their own biosystems. More modules and libraries and databases may be accepted in the future versions of L++.

Advantages of Modular Programming

The use of modules and namespaces becomes handy especially when building a code for large biosystems such as high level cellular processes or even organisms. For example, when developing a central carbon metabolism pathway we can modularly organize smaller pathways into distinct *.lpp* files at a functionally and practically manageable size, instead of writing out all the molecules and reactions involved in a single .lpp file.

Often the module to be imported is not found in the same folder as the *.lpp* file. If the module is organized in the directories under the current directory you can easily access them by using a string of directory names and the file name connected with the membership operator. Each level is connected through the membership operator. For example, you may be building *E coli* code and want to import *Transcription.lpp*, but it is located in the *GeneticInfoProcessing/Transcription* directory, it would look like this:

```
import GeneticInfoProcessing.Transcription.Transcription;
```

Here we are pulling a single file, *Transcription*, out of the directory *GeneticInfoProcessing*. When we think of modular programming, this ability to pull a single file from a directory without being required to import every file associated with it will let us keep our imports structured, organized, and clean.

Sometimes the place in which a file is saved or stored is beneath several tiers of other files, and at first this may make it look difficult to import a file that is stored within another file in a directory. Thankfully it is also possible to import a module that is not directly under the current directory. You can import it as long as its absolute path is defined from the **project directory**. The project directory will serve as the primary repository for your L++ projects, when you import a file the project directory will also be one of the first places that the program searches to retrieve the designated file.

For example, let's say we want to design a pathway for nucleotide synthesis. Nucleotides are not created by just assembling the required atomic components, but rather they require metabolic pathways that restructure and repurpose other molecules. Below we have *.lpp* files in a directory for nucleotide synthesis.

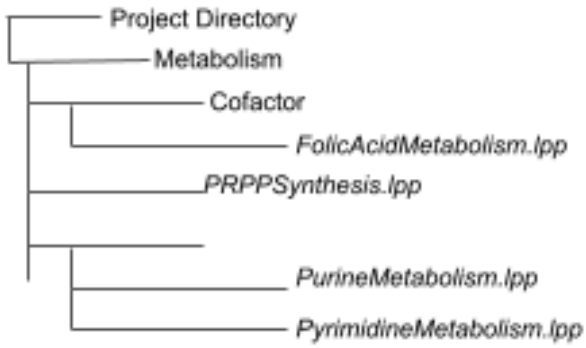


Figure 3: A tiered organization of program files within the L++ project directory (Placeholder)

```
// FolicAcidMetabolism.lpp in ProjectDirectory/Metabolism/Cofactor directory
import Metabolism.Cofactor.FolicAcidMetabolism;

// PRPPSynthesis.lpp in ProjectDirectory/Metabolism
import Metabolism.PRPPSynthesis;

// PurineMetabolism.lpp and PyrimidineMetabolism.lpp under the same directory
import PurineMetabolism;
import PyrimidineMetabolism;
```

First we have a *FolicAcidMetabolism.lpp* file. Folic acid is metabolized from folate, a naturally occurring molecule that we do not synthesize, but rather take in from our diet. Folate derivatives are used in the synthesis of purine and pyrimidine nucleotides. In addition, PRPP

(Phosphoribosyl pyrophosphate) synthesis is required in order to obtain the ribose 5-phosphate molecules necessary for purine and pyrimidine synthesis. Already we have a biosystem that is going to require several reactions and many components to build, but we can import each pathway individually to construct the complete synthesis pathway.

Biology: Nucleotide Synthesis Nucleotide synthesis is essential for the health of a cell, but it is no simple task to accomplish regardless of if the nucleotides are recycled or produced *de novo*. Nucleotides are synthesized in part by breaking down molecules like folate and glucose-6-phosphate.

Another type of useful importing method is to use the `from` keyword and `*` symbol. Here is an example of importing a database of metabolite concentrations from another L++ code *MetaboliteConcentrations.lpp*:

```
from MetaboliteConcentrations import *;
from ProteinConcentrations import FolA;
```

With the import statement above, every component in the module `MetaboliteConcentrations` (from *MetaboliteConcentrations.lpp*) is directly inserted from the module. In this case, all of the metabolite concentrations written in *MetaboliteConcentrations.lpp* will be directly accessible. Note that the `*` symbol used here will operate different than the `*` operator which we will discuss later in Chapter 5: Traditional Programming Language. The `*` symbol is context dependent, and its usage in a `from` keyword's statement will tell the program to use it differently than it would have as an arithmetic operator.

Unlike other programming languages, in L++ this "import all" action means more than a simple access to its variables. In L++ declaring a reaction and instantiating physical objects such as molecules or devices means that they will be directly incorporated into the simulation. You need to be careful when to and when not to use this particular way of importing. For example, this can be particularly useful for importing L++ databases, but not so much when the imported code contains more than what is to be used in the intended biosystem.

Note that where the import statements appear matters! Although most import statements may be introduced anywhere in code, modules are typically imported at the beginning of the source code all at once for code organization and clarity and style. It is particularly critical when importing all (from XX import *) is made because it will affect the very scope it is being stated.

Modular programming will have many advantages as you use it to design L++ code. Large programs designing whole organisms would require entire books of code, which would not only be cumbersome to write but also hard to organize and debug.

By breaking up your code into smaller modules, not only will your code be easier to debug, as you will have to manage only individual modules in each debugging process, but you will be able to use chemical pathways interchangeably. Pathways and reactions can occur not only in multiple spaces within a single cell, but they will also be active in other cells and organisms. Importing a module designated for a particular pathway, molecule, complex, or reaction will save you from re-writing the same code in each new program that requires it.

As other L++ programmers develop more *.lpp* files, there will be a larger arsenal of L++ files that can be imported for your convenience, or if you so choose, you can allow the *.lpp* files that you design to be used by other users in the L++ community!

The reusability of code modules, simplified debugging, and succinct code for complex organisms will be a valuable tool of yours as you construct life in L++.

Summary

In this chapter we discussed several key features of L++ programming that will allow you to give more structure to your code, allowing for easier modification, interpretation, and legibility of your code. We talked about how the **scope** of a program will allow you to give spatial and temporal context to virtual biosystems, how a **namespace** can be used to prevent ambiguity in your code, and how importing **modules** will simplify your life programs.

In the next chapter, Resolutions, we will discuss resolutions of life and how they are supported in L++, allowing you to design living systems ranging from the atomic level to whole organisms.

Chapter 4: Resolutions

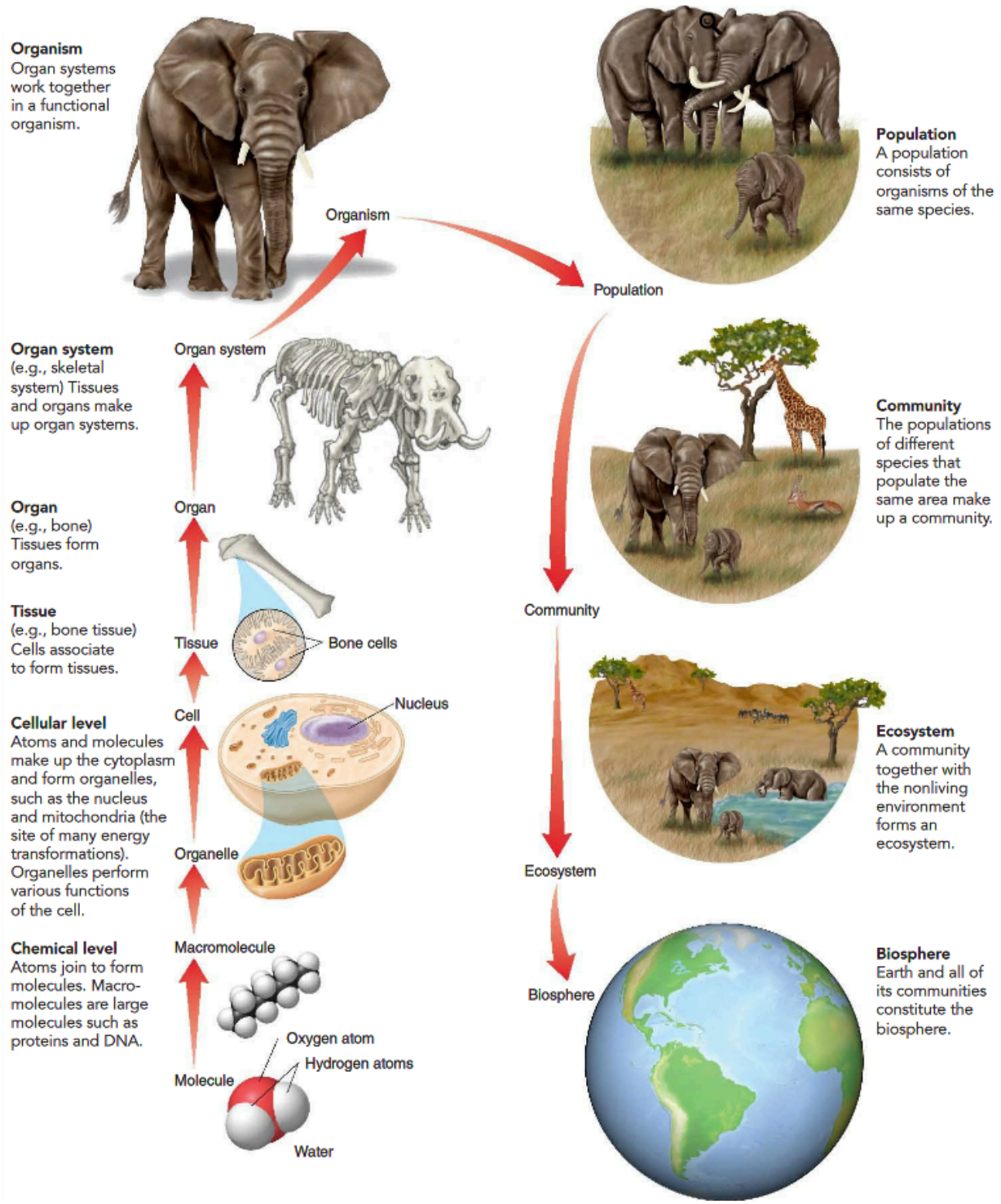


Figure 1. Resolutions of life span from atoms to biospheres and are build in a hierarchical and organized manner. [Unit 01 - Overview of Life - Mr. Scott's Online Classroom (weebly.com)]

What is the first thing you think of when you think of the word ‘Biology’? The image that comes to mind will be different for everyone. You might be an ecologist and think of a great ecosystem

and all of the interactions that constitute it between the biotic and abiotic components, or you might be a molecular biologist and your first thought is about the atomic interactions between a protein and its substrate. The study of biology spans many resolutions, so intuitively a platform built for designing life has to support all of those same magnitudes of life.

In this chapter we will talk about how we can program some of the many resolutions that are found across life forms. We will examine how we can designate cellular activity in L++, how we can construct tissues that collaborate to form organs, and how we can describe populations consisting of many organisms using L++.

4.1 Cells

So far we have talked about proteins, reactions, complexes, and other subcellular components of life. Although these are all essential components of life, they do not make up life themselves. Cells are the smallest unit of life that we can measure. They consist of a membrane that encapsulates the cytosol where reactions are carried out to help the cell function and grow. Although they constitute the most basic of life forms, cells come in various shapes and sizes and perform different functions, so we have to make sure that we can accurately capture cellular behavior and functions in L++ code.

By now we have already seen the use of cells in our code. Last chapter we used three prokaryotic species in our code to describe the utilization of namespaces, but we can also write code to describe cells and cell populations as well as unique differences between cells including morphological, molecular, and behavioral characteristics.

Take a look at the code provided below:

```
cell Macrophage
{
   reaction R1(LPS{out} + IL6{in} -> LPS{out} + IL6{out});
   reaction R2(LPS{out} + IL12{in} -> LPS{out} + IL12{out});
   reaction R3(LPS{out} + IL23{in} -> LPS{out} + IL23{out});
}
```

Here we designate several behavioral characteristics of a macrophage **cell.** In each of these reactions a bacterial toxin, lipopolysaccharice (**LPS**), is detected by the macrophage and results in production of interleukin 6, 12, and 23 (**IL6, IL12, IL23**) that is transported from the interior of the cell to the cell membrane. We can further expand on this code by introducing additional cells that will respond to the production of interleukins by the virtual macrophage.

Biologically, macrophages expressing antigens like interleukins will recruit T cells to the site of infection or invasion. Conveniently, we can express this process in L++ with a simple code.

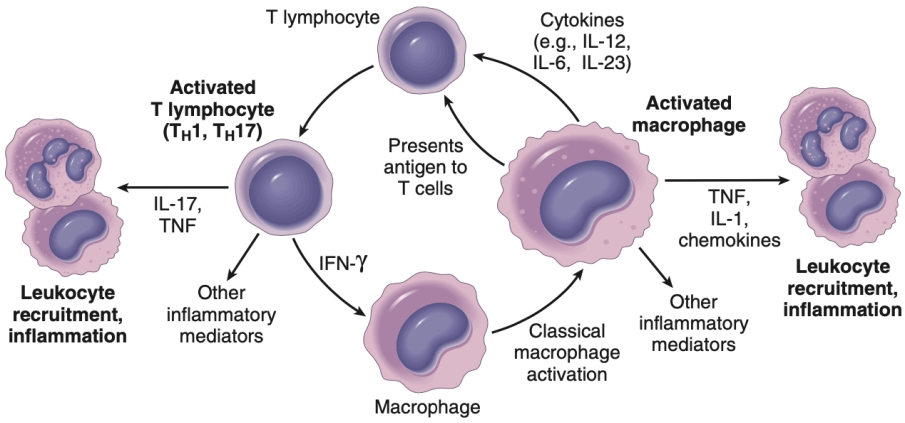


Figure 3-21 Macrophage-lymphocyte interactions in chronic inflammation. Activated T cells produce cytokines that recruit macrophages (TNF, IL-17, chemokines) and others that activate macrophages (IFN-γ). Activated macrophages in turn stimulate T cells by presenting antigens and via cytokines such as IL-12.

Figure 2. Behavior of macrophage cells in response to inflammatory signals and interleukin signaling.

```
cell TCell
{
}

TCell TH1
{
}

TCell TH17
{
}

reaction TCellActivation1(TCell + Macrophage.antigen -> TH1 + Macrophage.antigen);
reaction TCellActivation17(TCell + Macrophage.antigen -> TH17 +
Macrophage.antigen);
```

Here we designate a new cell type that is independent of `cell Macrophage` as `cell TCell` . More specifically we define two types of T cells, `TH1` and `TH17`. With our new cells added to this system we have a more complete understanding of how these cells can interact with one another. In our reactions that describe T cell activation. The interleukin expression on macrophages induces differentiation of a T cell into the helper T cells TH1 and TH17. Not only can we describe the behavior of any cell using L++, but we can also assign dimensions to the cell.

Cell morphology and function

We just took a look at how we can code the behavior of two cell types that work together to accomplish a task, but not all cells interact with one another. Despite this, cells may coexist within an organ or tissue and carry out their independent functions without requiring interaction with their neighboring cells.

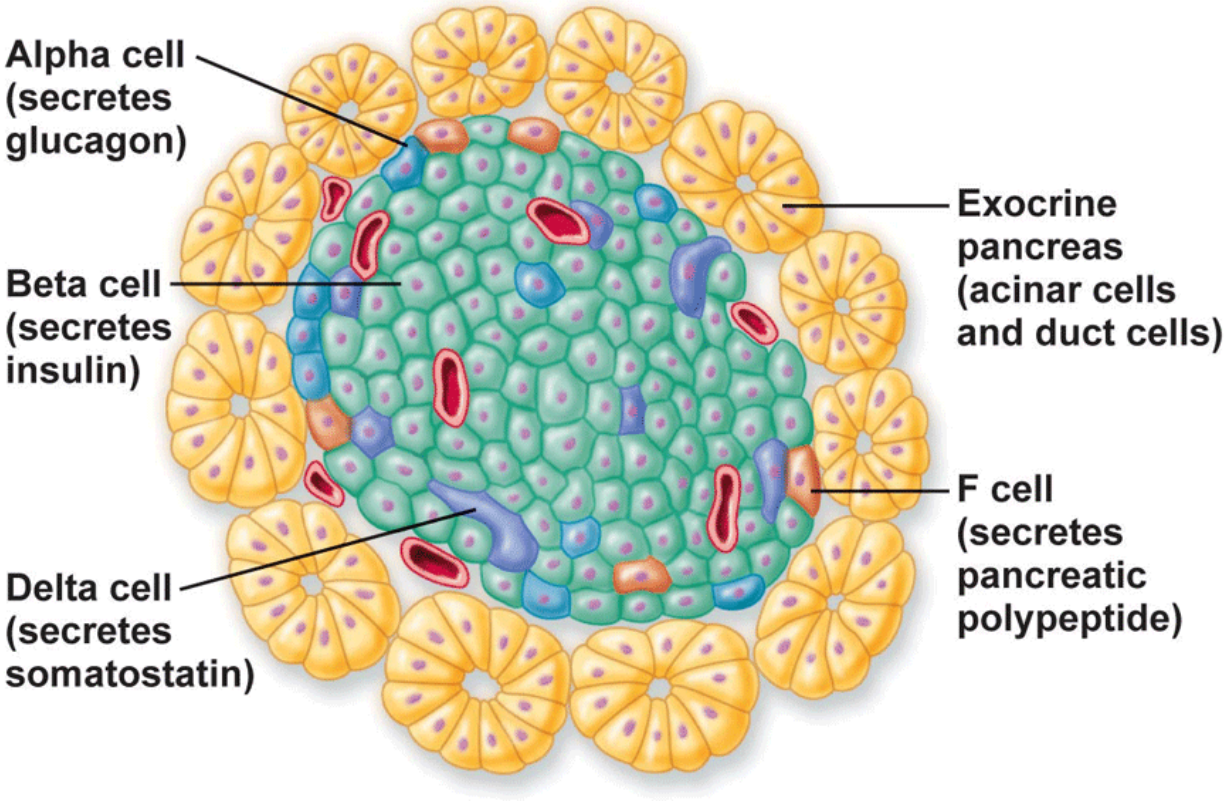


Figure 3. Cellular composition of a pancreatic islet. [How Glucagon Impacts Type 1 Diabetes and Vice Versa (diabetesmotion.com)]

Take for example some of the cell populations within our pancreas. The pancreas is responsible for producing a variety of hormones that will play important roles in regulating our body's internal environment; however, because life prefers to design organisms as efficiently as possible, not every cell in the pancreas produces the same quantities of every hormone. As such the pancreas contains specialized cells for producing specific hormones. We can designate both the behavior of the cell as well as the morphology by using the keyword `morphology`. Designating `morphology` requires a short description of the object's shape and dimensions. Just as before we can designate the behavior of a cell with a reaction.

Take a look at the following design for a pancreatic alpha cell below:

pancreatic islet.lpp

```
//
// Pancreatic islet
//

// produce glucagon
cell AlphaCell
{
  morphology.shape = “sphere”;
  morphology.dimension = (5um, 5um, 5um);

  peptide Signal;
  transporter T(signal + glucagon{in} -> Signal + glucagon{out});
}
```

Just as we discussed before we can designate our virtual alpha cell as a new cell using the `cell` keyword. Within this new virtual alpha cell we provide a description of the cell’s `morphology M` as a `“sphere”` with dimensions of 5um x 5um x 5um. Within the same scope we can designate the alpha cell’s behavior of glucagon secretion by moving glucagon from the cell’s interior out to the extracellular space by providing a `transporter T` to move glucagon in the presence of a signal.

We can use this same format to describe the morphology for pancreatic beta, delta and P cells, as well as their behavior of secreting insulin, somatostatin, and pancreatic polypeptides, respectively.

pancreatic islet.lpp

```
// produce insulin
cell BetaCell
{
  morphology.shape = “sphere”;
  morphology.dimension = (5um, 5um, 5um);

  peptide Signal;
  reaction R(signal + insulin{in} -> Signal + insulin{out});
}
```

```
// produce somatostatin
cell DeltaCell
{
  morphology.shape = “sphere”;
  morphology.dimension = (5um, 5um, 5um);

  peptide Signal;
  reaction R(signal + somatostatin{in} -> Signal + somatostatin{out});
}

cell PCell
{
  // pancreatic polypeptide
  reaction R(Signal + pancreaticpolypeptide{in} -> Signal +
pancreaticpolypeptide{out});
}

cell ToBeDetermined
{
  // ghrelin
}
```

With all of our specialized pancreatic cells described physically and behaviorally, we can further group each of them as endocrine cells with a simple command:

EndocrineCell.lpp

```
cell EndocrineCell = {AlphaCell, BetaCell, DeltaCell, PCell};
```

As you grow from programming single cells to more complex structures that are made up of many cell types like tissues and organs, being able to have specialized classes of cells will help you keep your program organized.

4.2 Tissues

A tissue is a population of cells with a common origin and the extracellular matrix that they associate with. A tissue serves as an intermediate structure between a cell and an organ. Here

we will discuss how you can program the structure and function of a tissue using the knowledge that we have learned about how to program the behavior and structure of cells. We will discuss how we can tell our program to incorporate cell types into a tissue, how to specify the purpose of a tissue, and how we can design several tissues to collaborate with one another to form an organ.

As we build a code up from the cellular level to a tissue we have new factors to account for. We can still designate specific reactions to happen within a tissue just as we have done in individual cell populations, and we can also give our tissue of interest a designated morphology with dimensions that allow it to take a three-dimensional shape in our program. Now, along with these previous features that we have used, we can specify the proportion of cell types found within the tissue.

Recall how we previously grouped several cell types within the pancreas into a single cell that we called `EndocrineCell`. Let's look at two methods that we can use to describe the cellular composition of a tissue.

Alpha, Beta, Delta, and P-cells are all found in the pancreatic islets, also called the islets of Langerhans. Pancreatic islets are responsible for the endocrine activity of the pancreas and secretion of various hormones in the body. Just as we did when we programmed cells in the previous section, we begin by using a keyword that indicates what we are programming, in this case we will be using the keyword **`tissue`** rather than **`cell`.**

We have several methods of describing the cellular composition of a tissue and each requires just an addition of a simple line in our designation of the tissue's morphology. If you want to describe the proportion of several cell types found within a tissue we can describe the percentage of a tissue as a floating integer ranging from 0.0 to 1.0.

First we can describe tissue composition by simply telling our program that 100% of the tissue is composed of cells in our `EndocrineCell` grouping from the previous section 4.1: Cells. Recall that `EndocrineCell` is described by our previously programmed pancreatic cells as `EndocrineCell = {AlphaCell, BetaCell, DeltaCell, PCell}`. We specify this by incorporating a new line after our description of the tissue's morphology:

```
tissue PancreaticIslet
{
  morphology.shape = "sphere";
```

```
  morphology.arrangement = 1.0 EndocrineCell;
```

Note the floating integer that we use to describe the proportion of `EndocrineCell` in the code here. This tells the program that the entirety of the pancreatic islet will be composed of the cells that we defined in the Endocrine Cell. The alternative method of describing the composition of our tissue here is to give the tissue more specificity to the percentage of the pancreatic islet that each cell in `EndocrineCell` makes up. If we take a look at the following alternative method:

```
  // Or alternatively,
  // morphology.shape = "sphere";
  // morphology.arrangement = 0.25 AlphaCell + 0.25 BetaCell + 0.25 DeltaCell +
0.25 PCell);
}
```

We can note how each of the cell types in our `EndocrineCell` group is given an individual percentage of the tissue that it will make up. While here they are all represented equally, in your own code you could alter these percentages to observe how the islets may function with different proportions of each endocrine cell represented.

Now that you are an expert in describing the morphology of a tissue in L++ programming we can discuss how we can further describe neighboring tissues that will work together with the pancreatic islets to form the pancreas.

If we want to build another pancreatic tissue, the pancreatic acinus, we have to perform a similar process that we did with the islets. First we begin by coding a line that tells our program about the function of an acinar cell. Acinar cells are responsible for the production and secretion of a range of digestive enzymes, so before we program the acinus tissue let's describe the cells. Take a look at the following code:

```
// Digestive enzymes
protein PancreaticJuice = {trypsinogen, chymotrypsinogen, elastase,
carboxypeptidase, lipase, nuclease, amylase};
```

```
cell AcinarCell
{
  morphology.shape = “sphere”;
  morphology.dimension = (5um, 5um, 5um);
  reaction R(Signal + PancreaticJuice{in} -> Signal + PancreaticJuice{out});
}
```

Just as we grouped several cell types in a namespace that we called endocrine cell, we can also create a new namespace that holds the information for several proteins. We first describe the `PancreaticJuice` that is produced by acinar cells before initializing our acinar cell. Pancreatic juice is a liquid that is secreted by the pancreas that contains several digestive enzymes, each of which is supported in L++ so we don’t need to specify the functions of each enzyme found within our pancreatic juice code.

Because we’re programming a cell again rather than a tissue we’ll follow our morphology keyword with dimensions for the cell rather than the composition that we used in the previous code for pancreatic islet tissue. Lastly we include a line describing the reaction that takes place when a signal tells an acinar cell to secrete pancreatic juice.

```
tissue Acinus
{
  morphology.shape = "sphere";
  morphology.arrangement = {1000, 2000} AcinarCell;
}
```

Now that we have the cells that we can use to build a pancreatic acinus, we can describe the structure of the tissue. You have probably already noticed that we don’t use a similar method to describe the composition of this tissue as we did before, with only one cell type making up the tissue it would be redundant to describe the proportion of acinar cells found in a pancreatic acinus, so this time we give the cells an arrangement.

We can write a short yet informative program for a pancreatic acinar tissue with just a couple lines that describe the function of the tissue and the proteins associated with it and the tissue's morphology.

```
// Digestive enzymes
protein PancreaticJuice = {trypsinogen, chymotrypsinogen, elastase,
carboxypeptidase, lipase, nuclease, amylase};
cell AcinarCell
{
  morphology.shape = “sphere”;
  morphology.dimension = (5um, 5um, 5um);
  reaction R(Signal + PancreaticJuice{in} -> Signal + PancreaticJuice{out});
}
 //
tissue Acinus
{
  morphology.shape = "sphere";
  morphology.arrangement = {1000, 2000} AcinarCell;
}
```

How you describe the cellular composition of a tissue may vary depending on what you are trying to describe, but as we've seen there are many ways to provide your program with concise descriptions of a tissue.

By now we are familiar with ways in which we can build various tissues, so now we will take a look at several other pancreatic tissues that we can describe before designing the entire organ, use the techniques that we have discussed so far to examine how the tissue is built, which cells we use to build it, and the function of the cellular components of the tissue.

```
cell DuctCell
{
  morphology.shape = “sphere”;
  morphology.dimension = (5um, 5um, 5um);
  reaction R(Signal + PancreaticJuice{in} -> Signal + PancreaticJuice{out});
```

```
}

// define how cells are put together to define a tissue
tissue AciniDuct
{
  morphology.shape = "tubular";
  morphology.arrangement = 1.0 DuctCell;
}
```

Once you have designed a tissue, it can also be incorporated back into code for other tissues that perform similar functions because each tissue still encapsulates similar cellular components when we continue to build other tissues. If you take a look at this code for the Islet of Langerhans, a structure found in the pancreas, you will notice that we incorporate components from three other tissues that we have designed so far.

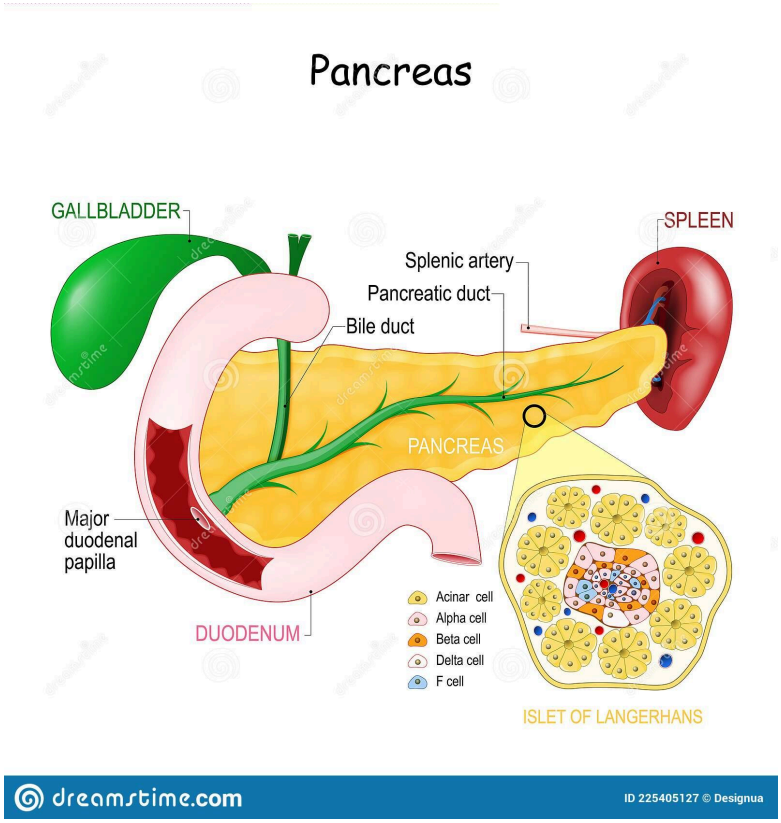


Figure 4. The morphology of a human pancreas.

```
tissue IsletOfLangerhans
{
  morphology.shape = “sphere”;
  morphology.arrangement = 0.9 (Acinus + AciniDuct) + 0.1 PancreaticIslet;
}

//
// Duct
//
```

To construct the Islet of Langerhans we recalled three tissues that we previously discussed: The acinus, acini duct, and the pancreatic islet. In using these previously written tissues in the program we can think of it as constructing the Islet of Langerhans using the cells that we used in the acinus, acini duct, and the pancreatic islet. Recall that in the acinus we used acinar cells, the acini duct was composed of duct cells, and the pancreatic islet was composed of alpha, beta, delta, and p-cells. Consequently the Islet of Langerhans in this program will then be secreting not only the pancreatic juice but also the endocrine signals that are secreted in the pancreatic islet.

You can incorporate your previously written cells or tissues into other biological structures as you need to, provided they perform similar functions. The redundancy of life is useful in these situations, as some cells that you might design for a human could perform analogous functions in other organisms.

We have addressed the function of the pancreas as an organ responsible for the secretion of various enzymes and hormones, and those molecules have to be moved out of the pancreas before they can begin processing food passing through the body. The Duct of Wirsung is the main duct that transports pancreatic juice and other molecules from the pancreas to the duodenum, the first place of food processing in the small intestine, while the Duct of Santorini serves as a smaller duct that also leads to the duodenum. We can keep both of the programs for these structures relatively brief, as we are not assigning any specific cellular behavior to these tissues.

As we finish the code for some other tissues in the pancreas, the Duct of Wirsung and the Duct of Santorini, we can keep these programs succinct and clear.

```
// main duct
duct DuctOfWirsung
{
  morphology.shape = "tubular"
  morphology.arrangement = 1.0 cell;
}

// accessory duct
duct DuctOfSantorini
{
  morphology.shape = "tubular"
  morphology.arrangement = 1.0 cell;
}
```

Now that you have taken a look at how we can describe several pancreatic tissues, think about how you might want to design other tissues that might interest you. What is the function of those tissues and what cells are they built from? With these pancreatic tissues that you have seen programmed, let's now look at how we can use them to build an organ.

Note on duct direction

4.3 Organ

Organs in our bodies and other organisms are made up of groups of tissues that have been adapted to perform specific functions. Once you have programs describing the tissues that make up an organ, coding the organ itself is relatively simple.

We previously described several pancreatic tissues and the cells that are used to build them including pancreatic islets, the pancreatic acinus, the Islets of Langerhans, acini ducts, the Duct of Wirsung, and the Duct of Santorini. So let's look at how we can use these tissues to construct a virtual pancreas.

Unlike some cells which we can simply describe as `"sphere"` or the ducts that we described as `"tubular"` , the pancreas is more oddly shaped and does not match the description of just one shape. Instead, due to its unique shape, we describe its morphology as `"pancreas"`. Because most organs do not fit a simple description using a single shape their morphology is supported in L++ and you only need to provide the dimensions with their morphology.

```
organ Pancreas
{
  morphology.shape = "pancreas";
  morphology.dimension = (8cm, 4cm, 3cm);
  morphology.arrangement = 0.9 IsletOfLangerhans + 0.1 stroma + DuctOfWirsung +
DuctOfSantorini;
}
```

The arrangement follows a similar style that we saw while we were coding various pancreating tissues. Just as we described tissues by the percent of that tissue in which unique cell populations are found, we can describe the percentage of an organ that is built by specific tissues.

If we keep this code in the same program as our previous resolutions we do not need to repeat the cellular composition of the tissues as we incorporate them into the organ. `IsletOfLangerhans` encapsulates information from the `Pancreaticislet`, `Acinus`, and `Aciniduct`.

4.4 Microbial Communities

Microbial communities are groups of microorganisms that share a common space and interact with one another. These communities play important roles in ecological phenomena as well as individual well-being.

Similar to how we programmed tissues earlier in this chapter we can design a program that describes microbial communities by providing information on the population size and the spatial relationships between the microbes.

While the human gut is home to dozens of bacterial species, we included ten common species here that have been encapsulated in the small intestine (`SI`). The population of microbes that we will use can be described simply by providing some identifier, here we use their scientific name preceded by `M` to designate that they will be contained within the `HumanGutMicrobiome.lpp` code that we imported.

Microbial Community .lpp

```
import HumanGutMicrobiome as M;

hsapiens H
{
    small_intestine SI
    {
        M.Streptococcus.salivarius.thermophilus = 1e12;
        M.Lactobacillus.acidophilus = 1e12;
        M.Lactobacillus.brevis = 1e12;
        M.Lactobacillus.casei = 1e11;
        M.Lactobacillus.plantarum = 1e11;
```

```
        M.Lactobacillus.rhamnosus = 1e11;
        M.Bifidobacterium.bifidum = 1e9;
        M.Bifidobacterium.longum = 1e8;
        M.Enterococcus.faecalis = 1e6;
        M.Escherichia.coli = 1e6;
    }
} = 1;
```

Our next code looks like it has a lot going on but it can be simplified. Once you are familiar with each of the components the rest will look simple. We are describing several species of bacteria, so we can include several reactions for critical cellular processes that each species will require. Here we have the reactions: `CellWallFormation`,`ProteinSynthesis`,`DNASynthesis`,and `FolicAcidSynthesis` that have arbitrary inputs, but defined outputs for cellular components and rate constants for each reaction. **`reaction`** `CellDivision` gives us a short description of several components required for bacterial cell division to give the gut microbiota a means by which they can reproduce.

The reactions that follow specify various effects of antibiotic drugs on the bacterial populations in the virtual small intestine. The program uses information from the imported modules for the cellular processes affected by the drug, allowing biological effects of each antibiotic to be simulated in the virtual organisms. Each of these reactions can be specified simply with a basic formula of

```
reaction name(drug -> biosystem, Ki=, nH=);
```

`Ki` serves as a reaction rate constant that describes the inhibition of the biosystem by the drug and `nH` determines the degree to which `Ki` is going to inhibit the biosystem.

HumanGutMicrobiome.lpp

```
bacterium Bacterium
{
    // high level cellular processes
    reaction CellWallFormation(-> CellWall, k=1);
    reaction ProteinSynthesis(-> Proteins, k=1);
    reaction DNASynthesis(-> DNA, k=1);
    reaction FolicAcidSynthesis(-> FolicAcid, k=0.1);

    reaction CellDivision(1200 CellWall + 1200 Protein + 1200 DNA + 120 FolicAcid
-> CellDivision, k=1);

    // antibiotics action
    reaction PenicillinAction_1(penicillin -> CellWallFormation, Ki=0.5, nH=2.2);
    reaction Penicillins_2(amoxicillin -> CellWallFormation, Ki=0.55, nH=2.1);
    reaction Cephalosporins_1(cephalexin -> CellWallFormation, Ki=0.61, nH=2.3);
    reaction Cephalosporins_2(cefdinir -> CellWallFormation, Ki=0.44, nH=2.5);

    reaction Macrolides_1(azithromycin -> ProteinSynthesis, Ki=0.46, nH=2.5);
    reaction Macrolides_2(erythromycin -> ProteinSynthesis, Ki=0.52, nH=2.1);
    reaction Tetracyclines_1(tetracycline -> ProteinSynthesis, Ki=0.57, nH=1.4);
    reaction Tetracyclines_2(doxycycline -> ProteinSynthesis, Ki=0.59, nH=1.5);
```

```
    reaction Lincosamides(clindamycin -> ProteinSynthesis, Ki=0.42, nH=2.1);

    reaction Fluoroquinolones_1(ciprofloxacin -> DNASynthesis, Ki=0.62, nH=1.9);
    reaction Fluoroquinolones_2(levofloxacin -> DNASynthesis, Ki=0.61, nH=1.8);

    reaction Trimethoprim_sulfamethoxazole(trimethoprim_sulfamethoxazole ->
FolicAcidSynthesis, Ki=0.58, nH=2.2);

    // antibiotics transport
    Antibiotics = {penicillin, amoxicillin, cephalexin, cefdinir, azithromycin,
erythromycin, tetracycline, doxycycline, clindamycin, ciprofloxacin, levofloxacin,
trimethoprim_sulfamethoxazole};
    transporter PermissiveTransport(Antibiotics{out} -> Antibiotics{in}, k=1);
}
```

Lastly we can describe the various species that are going to be included in the program. The organization of the bacteria will be intuitive.

Each genus of prokaryote can be classified as a new value of Bacteria

```
// gut microbiome, organized in phylogenetic style
Bacterium Streptococcus
{
    species salivarius
    {
        species thermophilus
        {
            morphology.shape = "sphere"; // diplococci
            morphology.radius = 0.5 um;
        }
    }
}

Bacterium Lactobacillus
{
    species acidophilus
    {
        morphology.shape = "sphere";
        morphology.radius = 0.5 um;
    }
    species brevis
    {
        morphology.shape = "sphere";
        morphology.radius = 0.5 um;
    }
    species casei
    {
        morphology.shape = "rod";
        morphology.dimension = (4um, 1um, 1um);
    }
    species plantarum
    {
        morphology.shape = "rod";
```

```
        morphology.dimension = (6um, 1.2um, 1.2um);
    }
    species rhamnosus
    {
        morphology.shape = "rod";
        morphology.dimension = (3um, 1um, 1um);
    }
}

Bacterium Bifidobacterium
{
    species bifidum
    {
        morphology.shape = "rod";
        morphology.dimension = (1um, 0.7um, 0.7um);
    }
    species longum
    {
        morphology.shape = "rod";
        morphology.dimension = (3um, 1um, 1um);
    }
}

Bacterium Enterococcus
{
    species faecalis
    {
        morphology.shape = "sphere";
        morphology.radius = 2 um;
    }
}

Bacterium Escherichia
{
    species coli
    {
        morphology.shape = "rod";
        morphology.dimension = (2um, 1um, 1um);
    }
}
```

Summary

In this chapter you learned that L++ supports many resolutions of life. You learned how to design cell populations using succinct lines of code that describe the cell's morphology and behavior, how to design tissues using cell populations, and how we can describe cooperation of tissues to code virtual organs. In addition you learned how to describe another form of cellular coexistence between distinct species as virtual microbial communities.

In the next chapter you will learn more about traditional programming language including data types, control flow commands, and built-in functions.

Chapter 5: Traditional Programming Language Support

L++ uses many facets of traditional programming languages in its own syntax. We have previously discussed some data types of traditional programming language support in Chapter 2.1: Syntax such as variables; however, the data types that we discussed in that chapter were but a fraction of the components of traditional programming languages that L++ supports.

In this chapter we will talk more about the traditional programming language that L++ programming supports including data types, control flow statements, and the built-in functions available in L++.

As you develop programs using L++, these components of programming from traditional languages will help you write clear code that is easily interpreted by the computer.

5.1 Programming Data Types

Unlike L++ data types that we have discussed previously such as molecules and reactions, programming data types are found in every programming language. Programming data types are going to be the core components of any program and you will find yourself using them often. These data types can typically be classified as basic data types which include numbers, booleans, and strings, and complex data types which include lists, arrays, dictionaries, and other forms of data organized at higher levels.

Basic data type

Basic data types include numbers, booleans, and variables. Here we will discuss how numbers and booleans are used in L++ to convey information to your programs.

Numbers

Numbers in programming take two different forms: integers and floating point numbers (decimals). Integers and floating point numbers are both supported in L++. The language is largely based on the context in which the syntax is used, so the type of number used in a specific case matters. For example, in cases such as chemical coefficients, the molarity of a chemical substance, or specifying a timestep may allow for integers or floating points, but the number of pipettes being used in an experiment must be a whole number. Using the wrong number type depending on the context will result in a compilation error.

Take a look at the examples below. While a decimal does not need to be specified when using a `float` number, adding a decimal to any `int` value will not work.

```
int a = 5;
int b = 2.0;    // doesn't work
float c = 10.0;
float d = 6;    // works
```

Arithmetic Operators

L++ mathematical operators follow the same set of rules as any other programming language, in terms of associativity, commutative, distributivity, and order of operations. Order can be further enforced through the use of parentheses. The following operators are available:

Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Modulo	`%`
Exponentiation	`**` `e`

L++ includes new syntax for exponentiation. Similar to other languages, `**` is still supported for statements like `float x = 10**-5`, representing 10^{-5}. However, `e` can also be used to represent scientific notation with the format `<base>e<power>`. 10^{-5} can also be expressed as `float x = 10e-5`.

Booleans

Just like in other programming languages booleans can be expressed as `true` or `false`, or numerically as `1` or `0`. Nonzero numbers assigned to boolean types will be evaluated as `true`, while zero will be evaluated as `false`. They are commonly used within the context of control statements. Here are some examples:

```
bool is_true = true;
bool is_false = false;
x = true;
y = false;
```

While we can designate a value to be a boolean through these means, logical operators also result in boolean values in that the result of the operation results in relationships that are either

true or false. They are commonly used in program control flow, such as if-else statements to determine whether a piece of code should run based on a given condition.

A boolean expression can often be used to keep track of certain conditions of a value, and in life programming this can be helpful to call on a process or biosystem to check if it is currently active in your program. Boolean values can alter the way a program runs by changing the path that a program takes when a certain boolean value is met, but we will discuss that further in 5.2 Control Flow.

This kind of binary switch is common in life, whether it involves a gene being switched on and off, or the rotation of a flagellum changing direction there are many ways to apply boolean values in life.

Assignment Operator

Assignments are expressed through `=`, which is typically used to assign a value of any data type to a variable that can be manipulated later. The assignment operator has different meanings for each data type that performs the assignment. Here's a list of each data type and the meaning of the assignment, along with several examples. Sections corresponding to each data type and structure will contain further examples and explanations.

Basic Data Type

Assignment to a value of the corresponding data type. Integers - `int` and `float` - should be assigned numerical values like `5.0` and `3`, with an optional unit.

```
float x = 5.0;
```

Parameter

Assignment to a value accepted by the corresponding parameter.

```
kcat = 1.4;
```

Molecular Substance

Assignment to a numerical value to indicate a concentration.

```
H2O = 5nM;
glucose = 12uM;
```

Experimental Structure

Assignment to another experimental structure as a reference. The example below assumes p1 and p2 are the same type.

```
protein p1 = p2;
reaction r1 = p1; // does NOT work
```

Reference Variable

Assignment to alphanumeric strings depending on the parameter.

```
reference Ingalls_2013
{
   title = "Mathematical Modelling in Systems Biology";
   page = "151";
   figure = "6.2";
}
```

Reassignments of a variable that was initially, explicitly typed are only possible if the data type (before and after) matches. If a variable's type was not stated upon declaration, then its type is inferred through the biological context of the simulation.

String format

In traditional programming languages a string refers to text values such as names, places, objects, or sentences. A string is a sequence of characters enclosed within quotation marks, `“ “`. So in a traditional programming language we could create a string like the one shown here:

```
print(“Where is the lab located?”)
```

Which would return:
→Where is the lab located?

Calling on strings like this won’t always be necessary in L++ code, but we have other ways to use strings and you have already seen a couple of examples. When we designed a cell we described its morphology using a string like so:

```
morphology.shape = "sphere";
```

Alternatively strings can also be used to assign a name to a variable. Here we assign the variable `Name` as `E.coli` so we can call on it later when we use it in other functions.

```
Name = “E.coli”;
int i = 1;

Label = “{} {}“.format(Name, i); // “E.coli 1”
```

5.2 Control Flow

As in any modern programming language, control flow is used to express and introduce rules within a program, such that certain pieces of code can be run for a number of times or only run under specific conditions.

In a biological sense, control flow statements are used to regulate cellular processes such as transcription factors driving gene expression or changing the flow of water from a cell to accommodate for the influx and efflux of ions. Control flow will allow your program to take one path only under the specified conditions, or else sending the program on another path.

Conditional Statements

If and else statements are used to control whether a piece of program should be executed, and the execution of the program will only happen under a certain condition. In many experimental contexts, for example molecular interactions may only occur if a specific concentration is

reached. The condition for an if statement must be wrapped within parentheses following the `if` keyword, and code used within `if` or `else` statements must be wrapped within curly brackets following the condition. `if` statements can stand alone, while `else` statements must be paired following an `if` statement.

The format for an if-else statement is as follows:

```
if (condition)
{
    statement1
}
else
{
    statement2
}
```

We can look at a more specific example with a molecular basis. In this code if the location for the variable `x_location` is greater than `2.4e-05`, then `ATP = 20 nM`. Otherwise the program will assume that `ATP = 0.5 nM`.

```
if (x_location > 2.4e-05)
{
    ATP = 20 nM;
}
else
{
    ATP = 0.5 nM;
}
```

Inequality Operators

These are operators used to evaluate two numerical expressions, returning a single boolean value indicating whether the statement is true or not. The following inequalities are supported:

Greater Than: >
Returns true if the left hand side is greater than the right hand side. False otherwise

```
x = 3 > 2;  // True
    // x is true
```

Greater than or equal to: >=
Returns true if the left hand side is greater than or equal to the right hand side. False otherwise.

```
x = 2 >= 2; // True
```

```
y = 3 >= 2; // True
// x and y are true
```

Less than: <
Returns true if the left hand side is less than the right hand side. False otherwise.

```
x = 0 < 1;
// x is true
```

Less than or equal to: <=
Returns true if the left hand side is less than or equal to the right hand side. False otherwise.

```
x = 1 <= 2;
y = 2 <= 2;
// x and y are true
```

Equals: ==
Returns true if the left hand side is equal to the right hand side. False otherwise.
Does not assign a value to a variable.

```
x = 5 + 3 == 4 * 2;
// x is True
```

Not equals: !=
Returns true if the left hand side is not equal to the right hand side. False otherwise.

```
x = 1 != 2;
// x is True
```

These inequalities currently only support numerical comparisons. Operator overloading for customized comparison of experimental or user-created structures may be supported in future development.

Iterative Statements

In many experimental cases, repetitive tasks are necessary, such as preparing a series of beakers with a particular solution or replicates in an experiment. L++ supports familiar looping structures such as the `for` loop and the `while` loop for code efficiency, similar to how they work in other languages.

`for` Loop

A `for` loop should be used when a piece of code is designed to run a fixed number of times. An example may be wanting to run an Electrophoresis experiment 15 times, plotting all the results on a graph. The syntax for the `for` loop is similar to their counterparts in languages like C++ and Java, with the following structure:

```
for (initialization; condition; increase)
{
    code block
}
```

1) **initialization.** A variable assignment for a number.
2) **condition.** Uses the previously declared variable and is often used with an inequality to state that while the condition holds true, the code within the for loop will execute.
3) **increase.** The update of the declaration variable by a certain number, which can either take a positive or negative value.The declaration can “count down” by decrementing or “count up” by incrementing.

Here’s an example of how `for` loops can be used in L++ code. Note that the declaration, condition, and increment are all wrapped within parentheses, and each one is separated with a semicolon. Code to run in a loop must be wrapped within curly braces.

```
for (int i = 0; i < 30; i = i + 1)
{
    petridish P
    {
        ecoli E
        {
            ThyA = ProteinConcentrations.ThyA * i / 30;
        }
        E = 1;
    }

    P = 1;
}
```

This code will generate 30 petri dishes
It generates 30 petri dishes containing *E.coli* with different thymidylate synthase concentrations.

```
organism Ecoli;
float x_locations[] = { 0.1, 5e-03, -5 };

for (int i = 0; i < 3; i = i + 1)
```

```
{
   Ecoli(x_locations[i], rand(10), 0) = 1;
}
```

This generates *E. coli* at a random location three times. The random x, y location is drawn from a uniform distribution from 0 to 10, inclusive. The third parameter represents the z coordinate, which is 0 as the *E. coli* are generated on a 2D plane.

`while` Loop

The `while` loop should be used when a piece of code is designed to run until a certain condition is met, rather than a fixed number of times. This may mean that the piece of code can never run, run once, run infinitely, or anything in between. Here's the structure:

```
while (condition)
{
    code block
}
```

If the variable's value still satisfies the condition of the `while` loop, the code block will run. Every time the code block is run, it is the programmer's responsibility to increment or decrement the value of the variable at some point inside the code block. Although it's most common to perform this increment at the end of the code block, it's up to the programmer to decide how and where the variable's value is updated.

A common mistake for many programmers is forgetting to increment or decrement the variable, resulting in running the code block infinitely (infinite loop). It is further important to note that the declaration for the variable used in the condition must occur before the while loop itself. Otherwise, the variable used in the condition is unable to know what value it is referencing. Here's an example:

```
organism Ecoli;
int i = 0;
while (i < 3)
{
   Ecoli(rand(10), rand(10), 0) = 1;
   i++;        // ++i works as well
}
```

These looping structures will become more useful upon further development, such as looping over an array of experimental structures in simulations involving numerous entities/trials.

Increment Operators

Syntactic sugar to increment or decrement a variable, either by 1 or by `n`, is supported.

Increments or decrements performed to a variable by 1 can be performed via `++` or `--`. Addition and subtraction are supported, meaning that you can either add 1 to a variable or subtract 1 from a variable with both of these operators, respectively. The order of placing these operators before or after a variable matters.

Preceding a variable with `++` or `--` increments/decrements the variable's value first, then evaluates the variable. Following a variable with `++` or `--` evaluates the variable first, then increments/decrements the variable value. The difference in the order of these operations matters most when used in evaluating conditional statements and expressions. Here's an example.

```
// Case 1:
   x = 1;
   y = x++;        // x is 2, y is 1


// Case 2:
   x = 1;
   y = ++x;        // x is 2, y is 2
```

Increments or decrements performed to a variable by `n` wil can be performed by `+=, -=, *=, /= , or %=`. In any case, the operation is performed on the variable, and the result of the operation will be assigned to the variable. Here's the format:

`x [o]= n;`

`[o]` - Operator. Required. Can be one of `+`, `-`, `*`, `/`, or `%`.

5.3 Built-in Functions

There are many functions built into L++ that will be valuable tools while you build virtual organisms and biosystems. Here are some examples of functions that you will find useful.

min()	Returns the smallest value in a set of numbers
max()	Returns the largest value in a set of numbers
floor()	Rounds numbers down to the nearest integer
ceil()	Rounds numbers up to the highest integer
log()	Returns the natural log of a value
pow(x,y)	Returns the power of x to the value of y
random	Returns a random number
random.uniform	Returns a random number within a specified range
random.normal	Returns a random number from a sample within a normal distribution
int	Returns an integer

Summary

In this chapter you learned about some of the components of L++ derived from traditional programming languages, including data types, control flow, and built-in functions. While there are many more ways to give your programs structure these fundamental components of a code will be widely used in programming life.

PART II: Projects

Chapter 6. Analog Signal

Learning Objectives

1. To understand the chemical equations are functions that process analog signals in the form of molecular quantities.
2. To understand that L++ code can describe chemical equations with intuitive syntax.
3. To understand that by combining multiple chemical equations, more complex patterns of analog signals can be processed or generated. L++ can make this system building and analysis easy.

Introduction

Analog signals are used in systems where there is continuous change in the quantities being measured, giving them their distinct undulating shape. Analog signals are used to measure interactions in a variety of media including electrical signals, changes in pneumatic pressure, as well as mechanical and chemical signals. In audio recordings you are measuring signal voltage in response to changes in air pressure, and in biological contexts you will typically be measuring the response of one molecule to another chemical or molecule interacting with it.

Many aspects of life are characterized by different rates of change within a biological system, a pattern that is captured beautifully by an analog signal. The processes that we can picture in analog format span every discipline in biology. Consider the handful of examples shown here.

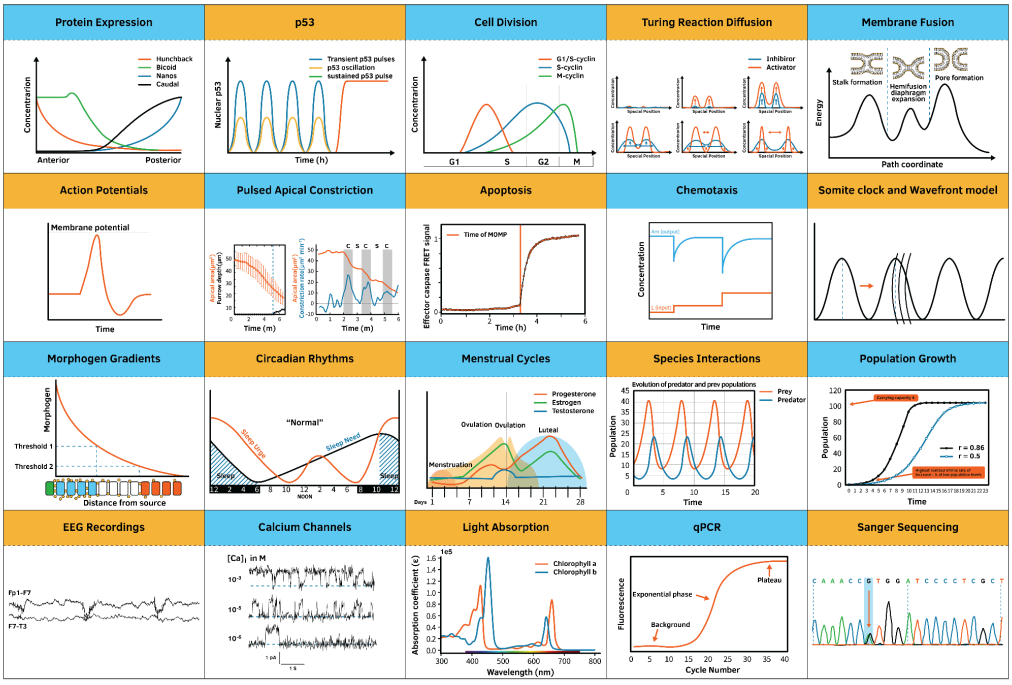


Figure 1. A Repertoire of Analog Signals.

Analog signals can capture both gradual and continuous change in a biosystem, and because life is rarely quiescent, analog signals are widely present in life. Simply speaking, analog signals define biosystems including organisms. Constructing various analog signals through molecules and reactions and combining these signals are the key to creating organisms.

Basic Chemical Reaction

Analog signals exist in many contexts within biological systems, and as we explore them in these different contexts we will see that they can take many unique forms determined by the interactions between molecules. As you design biological systems, it will be important that you understand these different analog signal forms so you can determine if a graphical representation of your program is working as intended at just a glance.

Here we will discuss how we can simulate various forms of a biological analog signal. First, consider a system in which we look at a change in two molecules, molecule A and molecule B.

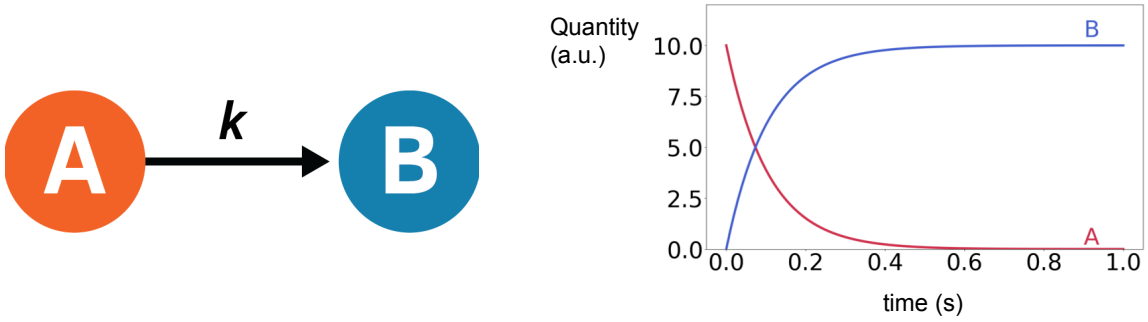


Figure 2. A simple chemical reaction in which molecule A is converted into molecule B.

By the law of mass action, the rate of reaction is directly proportional to the concentration of reactants (molecule A, in this case). The k is a reaction rate constant that describes the proportional relationship between the concentrations of reactants and the rate of the reaction. Note that this is a **function** processing the quantity of molecule A as **input** into the quantity of molecule B as an **output**. We can use L++ code to describe this chemical reaction like this:

```
reaction r1(A -> B, k=?);
A=10;
B=0;

// hint: this k value is an integer.
```

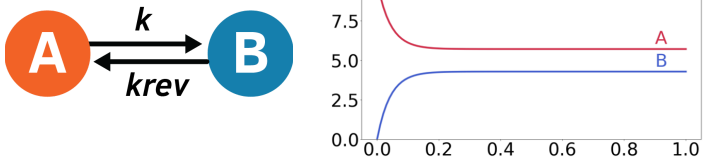
Notice that there is one important component missing from the code before we can generate a plot. We need a rate constant (**k**) to describe the energetics involved in generating the molecular interactions in this reaction. Try several values in your code and notice how different k values change the outcome of this reaction.

The above reaction descriptions can be mathematically expressed as a set of differential equations can describe their molecular dynamics over time (t):

$$A\,(t = 0) = 10\,;\; B\,(t = 0) = 0$$
$$\frac{dA}{dt} = -\,kA(t)$$
$$\frac{dB}{dt} = kA(t)$$

This simple conversion can be further expanded by providing a statement for a reverse reaction, allowing both molecule `A` and molecule `B` to reach a point of equilibrium. In order to achieve this we add, **`krev`**, an additional parameter to our differentials to describe the rate of reverse reaction that allows for the conversion of molecule `B` back to molecule `A`.

Now if we implement these changes in the code we get:




```
reaction r1(A -> B, k=?,
krev=?);
A=10;
B=0;
```

Figure 3. A chemical reaction in which molecule A and molecule B reach equilibrium.

The addition of a single variable or parameter can greatly affect the system behavior and change information processing, so it is important to note how these parameters work together and relate to each other. The question marks highlighted in yellow in Figure 3 indicate where you should input values for the reaction parameters, **`k`** and **`krev`**. Try assigning values to these parameters so that your reaction reaches a state of equilibrium at 6 for molecule `A` and 4 for molecule `B`.

Quantitatively, we can describe these new reactions by expanding our differential equations as:

$$\frac{dA}{dt} = -\,kA(t) + k_{rev}B(t)$$
$$\frac{dB}{dt} = kA(t) - k_{rev}B(t)$$

In terms of thermodynamics, at equilibrium the change of the quantity of molecules `A` and `B` are equal 0 ($kA(t) = k_{rev}B(t)$). Therefore, **`k`** / **`krev`** describes the steady state ratio of molecule `B` to `A`, which is called equilibrium constant, K_c.

In-class exercises

- Change the initial quantity of molecule `B` to 4.

- Do you predict that the steady state quantity of molecule `A` will stay at its original value of 6 or change?
- In this model, how would you update the **`k`** or **`krev`** parameters to keep the original steady state level of `A` at 6?

Let's look at how we can expand on these simple reactions to describe more complex biological interactions.

Signal Amplification

Many biosystems evolved to maximize efficiency, especially when a biosystem has to transmit a molecular signal. Signal amplification is essential to maximizing the efficiency of a molecular message across a cell or between cells.

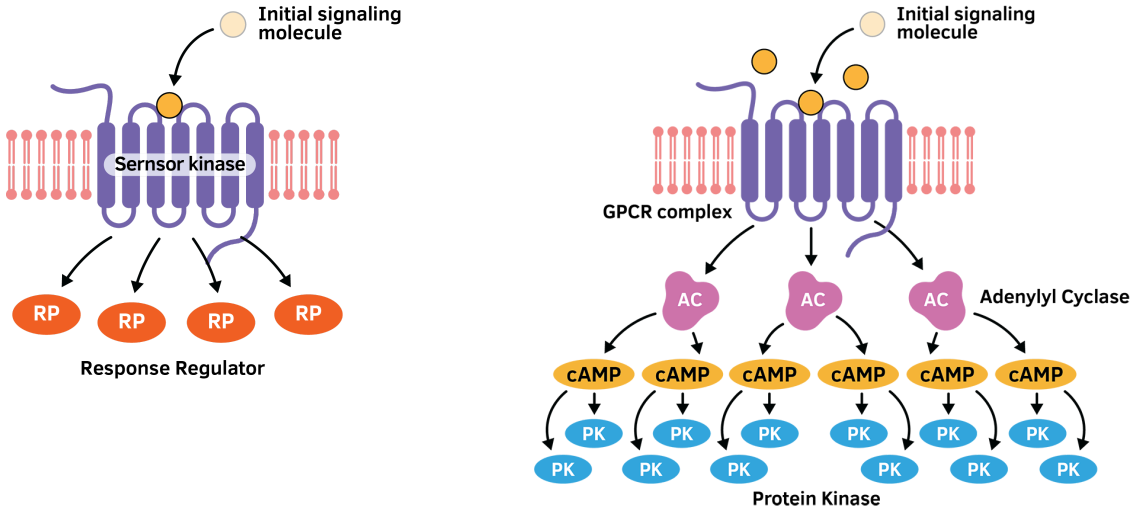
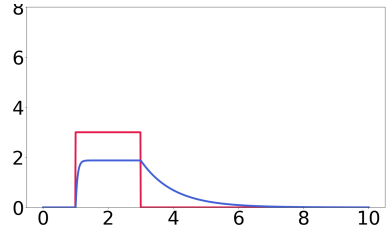
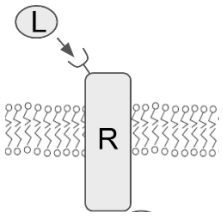


Figure 4. Two-component regulatory system and G-protein signaling cascade are common biological implementations of chemical signal amplification.

The two-component regulatory system (TCS) is a simple system of signal amplification that we can use to practice coding an amplification signal. These systems, as their name suggests, have two major components: a membrane-bound receptor and a response regulator protein. At first glance a system of two-component signaling has a small logistical problem, there is a small number of receptors that may not be able to fully capture the available signal around them and the receptors have to produce a greater response than their small population can produce alone. To accomplish this they activate response regulator proteins via phosphorylation that transmit the signal, and a single receptor being able to activate many response regulators is sufficient to amplify the signal.

We can simulate a form of signal amplification with a brief description for a two-component system in L++ code. We set up the system with a small amount of receptor `R` that can bind to `L` in a reversible way by introducing both **`k`** and **`krev`**:

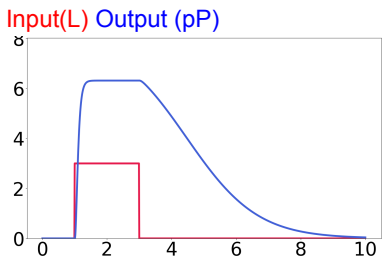


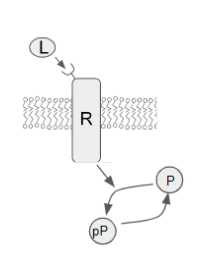



```
reaction r1(R + L -> RL, k=?, krev=1);

R=2;
L[:]=0, L[1:3]=3;
```

Figure 5. A model of a two-component signaling system before signal amplification.

Try different values for **k** to modulate the sensitivity of the receptor and saturate (or reach the maximum level) of ligand binding of the receptor. Note that it cannot produce a signal that is higher than the limiting quantity of the receptor. Let's see if introducing a more abundant messenger molecule `P` that can be turned on by the ligand bound receptor `RL`, which should be turned off in the absence of the ligand bound receptor.






```
reaction r1(R + L -> RL, k=?,
krev=1);
reaction r2(RL + P -> RL + pP, k=?);
reaction r3(pP -> P, k=3);

R=2, P=8;
L[:]=0, L[1:3]=3;
```

Figure 6. A model of a two-component signaling system with signal amplification.

Try different values for **k** in `r2` to amplify the signal from `RL`, which is the previous output of the system. The overall reversibility of the model is captured in the gradual decline in our output in the two-component system.

In-class exercises

- Let's suppose that there were a lot of receptors (R = 20) in the first place without the amplification step involving P (P=0). Considering RL being the output, how does the output level compare to that of the original model (pP)? Would this provide biologically meaningful signal amplification?

- How does the following set of reactions behave differently from the original code? Would this new code be a biologically meaningful code?

```
reaction r1(R + L -> RL, k=5, krev=1);
reaction r2(P + RL -> pP + RL, k=6, krev=3); // modified r2.
// no reaction r3

R=?, P=?;
L[:]=0, L[1:3]=3; // input
```

Now that we've taken a look at how we can amplify a signal, what about inhibiting one?

Product Inhibition

Regulation of molecules, genes, and proteins plays a crucial role in maintaining a system's balance. For instance, in an efficient system, energy production within a cell should only be activated when required, to prevent waste and potential toxicity from excessive energy molecules. Therefore, when a system has abundant energy molecules, this should signal the cell to halt further energy production, and this mechanism is called product inhibition.

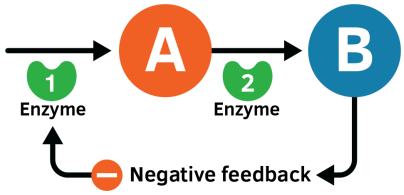
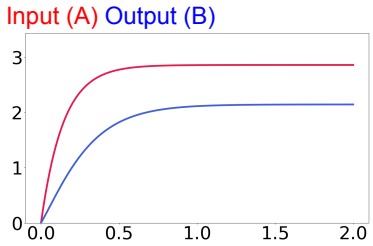
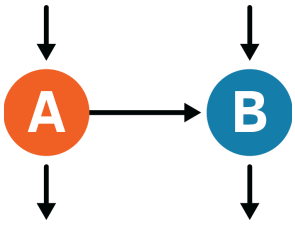


Figure 7. Negative feedback mechanisms are employed to block enzymatic activity in systems with product inhibition.

Let's situate a simple reaction that converts a molecule `A` to a molecule `B`, where the molecules are being introduced and removed in an open system similar to a metabolic network. Both of these molecules are assumed to be abundant in the system.

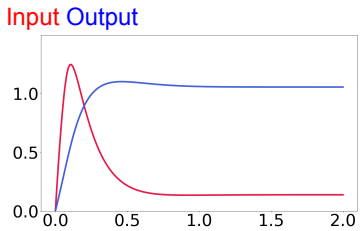


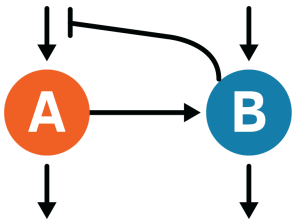



```
reaction r1(   -> A, k=20);
reaction r2(   -> B, k=5);
reaction r3(A ->   , k=5);
reaction r4(B ->   , k=5);
reaction r5(A -> B, k=2);
```

Figure 8. A simple reaction converting molecule `A` to `B` within an imaginary metabolic network without product inhibition.

What if molecule `A` is no longer useful but rather wasteful or even toxic once enough of molecule `B` has been produced? It would be beneficial for the system to prevent accumulation of molecule `A`. To shut down the influx of A, we can introduce an inhibitory reaction, where the "product" molecule `B` inhibits the molecule `A`-generating reaction `r1`. To describe the strength and degree of inhibition, an inhibition constant (`Ki`) and the Hill coefficient (`nH`) must be designated, which will assume an allosteric inhibition model. In this model of inhibition, an imaginary enzyme with multiple binding sites to cooperatively bind to the inhibitory molecule (`B` in this case):






```
reaction r1(  -> A, k=20);
reaction r2(  -> B, k=5);
reaction r3(A ->  , k=5);
reaction r4(B ->  , k=5);
reaction r5(A -> B, k=2);

// regulatory reaction
reaction r6(B -> r1, Ki=?, nH=?);
```

Figure 9. Introduction of a negative feedback loop that blocks production of molecule A creates a system of product inhibition.

Implementing the product inhibition mechanism in the model, we no longer accumulate molecule `A` after molecule `B` has reached a certain level of production. Quantitatively, how the set of inhibition constants are used in the whole model can be illustrated in the following ordinary differential equations:

$$\frac{d}{dt}A(t) = \frac{k_1}{1+(B(t)/K_i)^{nH}} - k_3A(t) - k_5A(t)$$

$$\frac{d}{dt}B(t) = k_2 + k_5A(t) - k_4B(t)$$

In the allosteric inhibition term, K_i indicates the concentration of an inhibitor required in order for ligands to occupy ½ of the receptor binding sites. The lower the K_i, the faster the inhibition gets. nH, the Hill Coefficient, describes the degree to which two molecules cooperate. The higher nH is, the more efficient inhibition will become.

In-class exercises

- Once we introduced the product inhibition mechanism, the final steady state level of molecule `B` turned out to be around 1, which was half of the amount of that of the model without product inhibition. How can we adjust the model to reach the steady state of molecule `B` in the presence of product inhibition?

Oscillations

Another important activity that we can measure is an oscillation in a system. We mentioned several systems that are driven by the cyclical characteristics of oscillations within a biosystem, such as protein synthesis, heartbeats, circadian rhythm, menstrual cycles, lotka-volterra interactions, cell division, neuron spiking and many more, you may have examples of oscillations in your own work that come to mind.

Oscillations drive biosystems that require continual, cyclical change. When we look at oscillations in an analog signal, the peaks and troughs of the output signal should be slightly offset from the input. Logically this makes sense, as the output cannot have a peak at the same time as an input. Often these oscillations begin with negative feedback loops that cause a time

delay in the output of a system while positive feedback allows for the oscillator to be tuned more acutely.

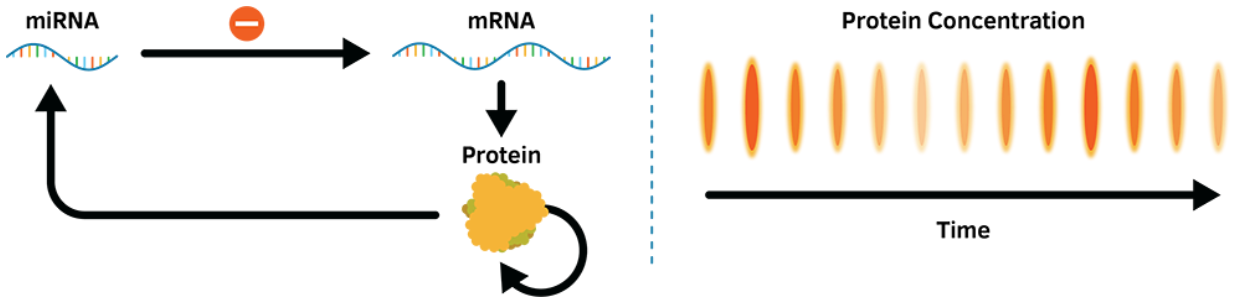
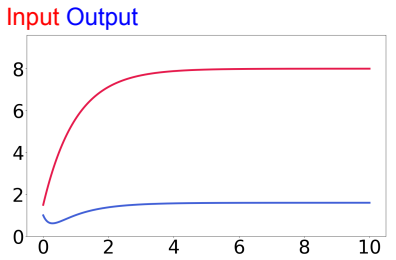


Figure 10. Fluctuations in the rate of mRNA translation result in oscillations in the concentrations of its protein over time.

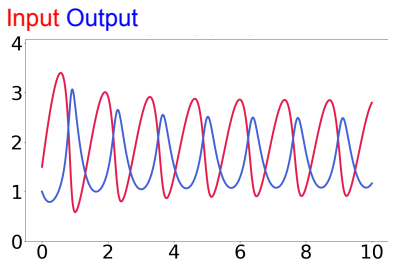
If we begin writing a code for an analog signal portraying some form of molecular oscillations, we should intuitively start similar to some of our past reactions: an arbitrary molecule, molecule `A`, will be used to produce molecule `B`. This conversion reaction has an intrinsic negative feedback mechanism as molecule `A` gets consumed by producing molecule `B`, which makes the model even simpler.

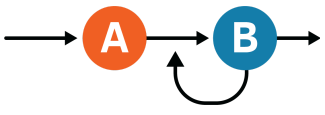



```
reaction r1(  -> A, k=8);
reaction r2(A -> B, k=1);
reaction r3(B ->  , k=5);

A=1.5, B=1;
```

Figure 11. The conversion of molecule A to molecule B prior to introducing a positive feedback loop.

Now let's add the positive feedback loop that increases the production of molecule `B` by adding an additional reaction, in which molecule `B` increases the conversion rate of molecule `A` to molecule `B`. This mechanism describes an allosteric activation of an imaginary enzyme catalyzing the conversion reaction, where we can specify the strength and degree of activation using half saturation constant, `Ks`, and Hill Coefficient, `nH`. (Note: this reaction would also deplete the available molecule `A` faster in this case).






```
reaction r1(  -> A, k=8);
reaction r2(A -> B, k=1);
reaction r3(B ->  , k=5);
reaction r4(B -> r2, Ks=?, nH=?);

A=1.5, B=1;
```

Figure 12. Addition of a positive feedback loop using a half saturation constant and hill coefficient transforms the system to have oscillations of concentration in molecules A and B.

With proper `Ks` and `nH` parameters, we can tune the system to enter an oscillatory pattern. Graphically our oscillations follow a pattern of off-set highs and lows. The input of molecule `A` (Red) allows for the production of molecule `B` as an output (Blue); however, this reduces the level of molecule `A` as molecule `B` increases its own production. Unlike our previous system of product inhibition, where molecule `B` can still be produced in the absence of molecule `A`, this system requires molecule `A` to produce molecule `B`. Thus the concentration of molecule `A` increases to accommodate for the lowered concentration and continues the oscillations of molecule `A` and molecule `B`.

Quantitatively this system of molecular oscillations can be represented by the following set of differential equations:

$$\frac{d}{dt}A(t) = k_0 - k_1[1 + (\frac{B(t)}{K_s})^{nH}]A(t)$$

$$\frac{d}{dt}B(t) = k_1[1 + (\frac{B(t)}{K_s})^{nH}]A(t) - k_2B$$

`Ks`, the half saturation constant, specifies the concentration of a substrate that is required to half the maximal reaction rate. `nH`, the Hill Coefficient, describes the degree of cooperativity between molecules `B`'s.

In-class exercises

- Try to find a set of reaction parameters that can also generate a persistent oscillation with `A=3 and B=1`.

Enzyme-catalyzed reactions

A biochemical convenience that has greatly helped life flourish is the enzyme. Enzymes help catalyze most biochemical reactions, providing energy required to facilitate reactions to either speed up a naturally occurring chemical reaction or to produce a reaction that would not otherwise happen under natural conditions regardless of the reactant concentrations.

An elementary reaction is one in which the reactions come together on their own to generate the products of the reaction. These reactions are the most basic form of a chemical reaction and the rate of the reaction is directly proportional to the concentrations of the reactants as they are driven solely by the Law of Mass Action. These reactions are not efficient when you consider that organisms cannot rely solely on the Law of Mass Action when they require a molecule or chemical, so organisms use enzymes to facilitate those reactions. We describe enzyme-catalyzed reactions by using Michaelis-Menten kinetics.

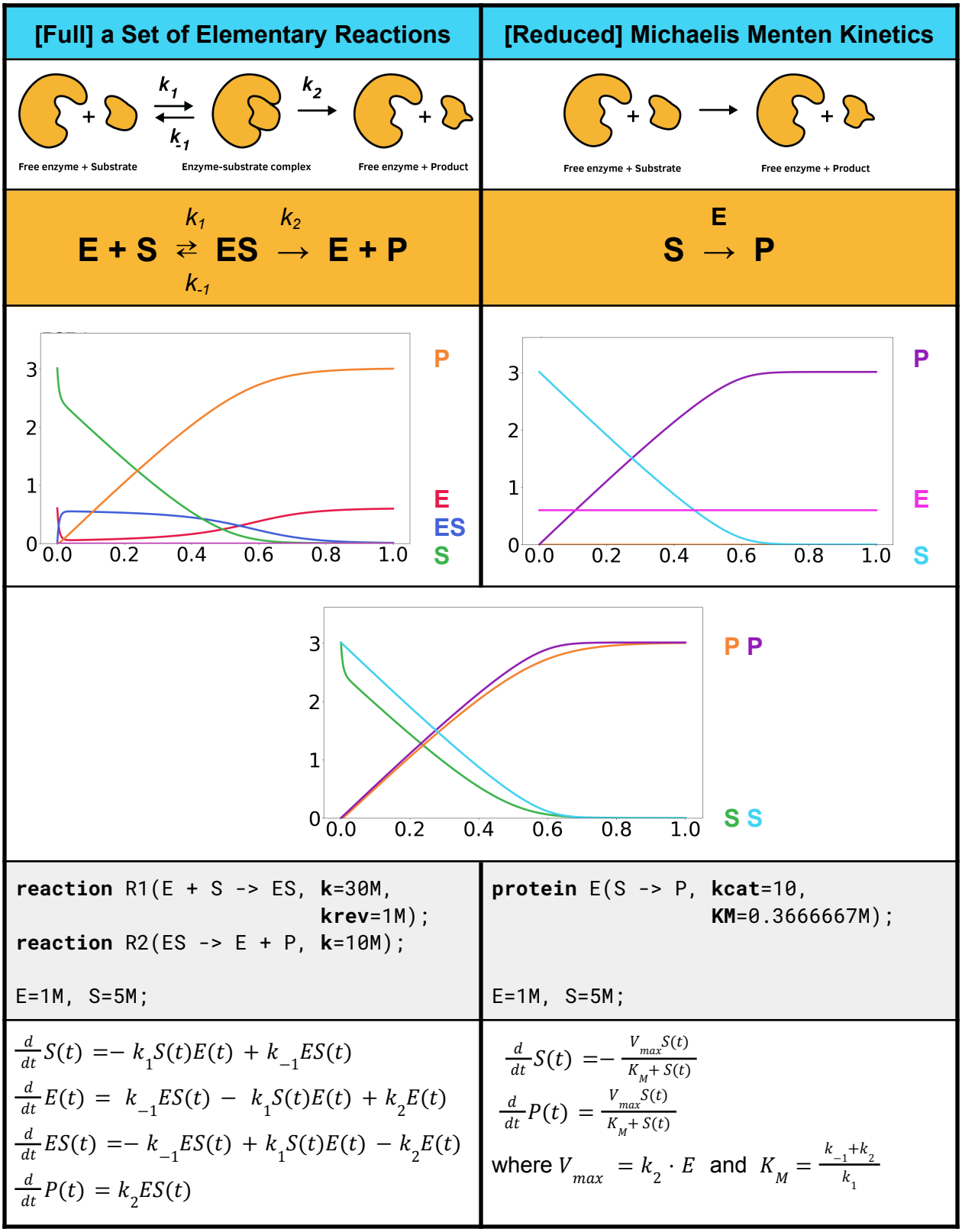


Figure 13. Simplification of enzyme-catalyzed reactions using the model of Michaelis Menten Kinetics.

The Michaelis-Menten Kinetic model allows us to simplify biochemical processes and pathways that are facilitated by enzymatic activity. Where some processes may require several elementary reactions, it is simpler for us to encapsulate each reaction into a single value (K_M) (from an experiment or for a model) than to account for the rate constants for each elementary reaction.

In addition, we have a value ($V_{max} = k_2 \cdot E$) that tells us about the limiting rate of the reaction. In other words, the catalytic reaction is dependent on the total amount of enzymes available. Linear growth would work with infinite substrate and infinite enzymes, but because the reaction is limited by the total number of enzymes there will come a point where the reaction plateaus with high levels of substrate.

In-class exercises

- Play with `kcat` and `KM` in the code. What happens to these models when we increase or decrease the parameters? Describe an efficient enzyme in terms of `kcat` and `KM`.

Supplementary Materials

Elementary vs enzyme-catalyzed reactions:

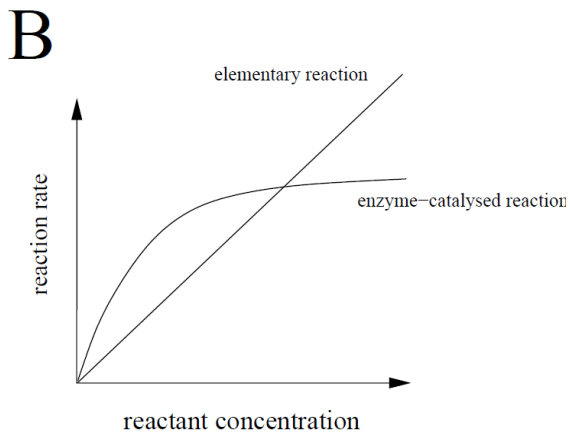
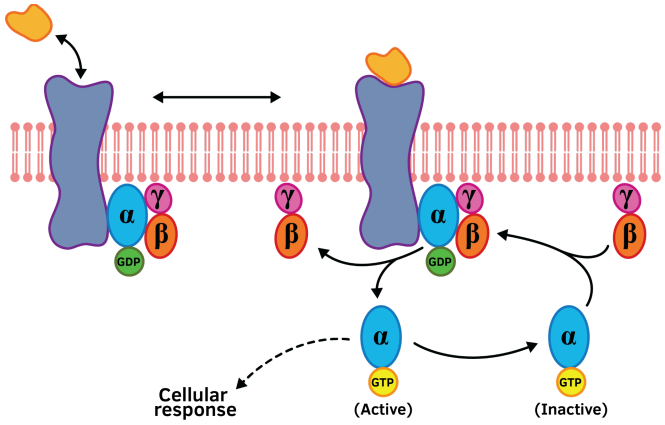


Figure S1. Enzyme-catalyzed reactions affect the reaction rate of biochemical reactions, plateauing with saturated reactants.

Problem Set

- GPCRs are a common form of signal amplification in biological systems, and their ability to greatly increase a signal from a single input have allowed for their use in many processes that require sending a signal a long distance including systems for vision and olfaction. A GPCR is activated by ligand binding which releases bound GDP for GTP on the GPCR and releasing the alpha subunits from the beta and gamma subunits that begins a signaling cascade. Implement this system using L++ code.



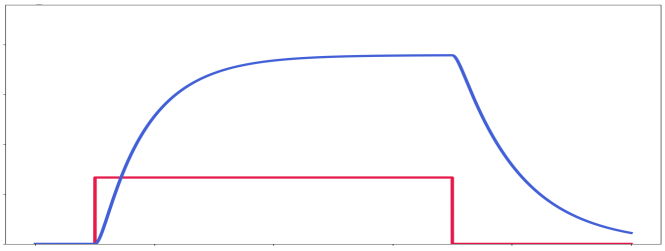
$$R + L \underset{k_{RLm}}{\overset{k_{RL}}{\rightleftarrows}} RL$$

$$G + RL \xrightarrow{k_{Ga}} G_a + G_{bg} + RL$$

$$G_a \xrightarrow{k_{G1}} G_d$$

$$G_d + G_{bg} \xrightarrow{k_{G1}} G_d$$

Input Output



- Parameter values to use: $k_{RL} = 2\times10^{-3}nM^{-1}\ s^{-1}$, $k_{RLm} = 10^{-2}\ s^{-1}$, $k_{Ga} = 10^{-5}$ (molecules per cell)$^{-1}\ s^{-1}$, $k_{Gd0} = 0.11\ s^{-1}$, $k_{G1} = 1$ (molecules per cell)$^{-1}\ s^{-1}$.
- The total G-protein population is 10000 molecules per cell, while the total receptor population is 4000 molecules per cell.
- **Input**: At time t = 100 seconds, 1 nM of ligand is introduced. This input signal is removed at time t = 700 seconds, causing the response to decay.
- *Watch the mixed units in molecules and concentrations!*

- If you recognize a form of an analog signal in your own research, what form of signal is it and how do you think you can implement that signal using L++ code?

References

1. Brian P. Ingalls, Mathematical Modeling in Systems Biology: An Introduction. The MIT Press. 1st edition (July 5, 2013)

Chapter 7. Bacterial Chemotaxis

Learning Objectives

1. To understand the known biological model of detecting relative changes in the environment through the example of adaptive behavior of bacterial chemotaxis.
2. To exercise L++ programming for multi-scale and high level descriptions of unknown molecular mechanisms, such as the flagellar rotation function.

Introduction

An important adaptation for an organism to survive in a dynamic environment is the ability to change its behavior in response to changes in the environment. Many mechanisms for sensing changes in an organism's environment exist, including aerotaxis, stimulation by wind, barotaxis, stimulation by pressure, and phototaxis, stimulation by light, among a variety of others. Mechanisms for sensing chemical gradients have been crucial to the survival of all organisms, and they are essential not only for humans, but even for simple single-celled organisms such as *E. coli.*

E. coli has two modes of movement, tumbling and running. Tumbling consists of unoriented movement in which the bacterium moves in a seemingly random manner, this is the natural state of an *E. coli* bacterium. Tumbling is a way by which the bacterium can survey its environment without exerting as much energy as running would cost. It achieves this motion by having its flagella arranged in a non-uniform manner around its cell body, and the clockwise rotation of the flagella keeps the flagella extended away from the bacterium, forcing it to move in a non-oriented manner. As soon as the bacterium senses a desirable or undesirable change in its environment, it must adapt and move in a directed manner either toward or away from the stimulus. It may choose to move toward sources of nutrition like glucose, or away from toxic chemicals like indoles. Counterclockwise rotation of the flagella will bundle the appendages together to generate the more efficient movement, running.

How can we model the behavior of an organism like chemotaxis in L++? Let's take a look.

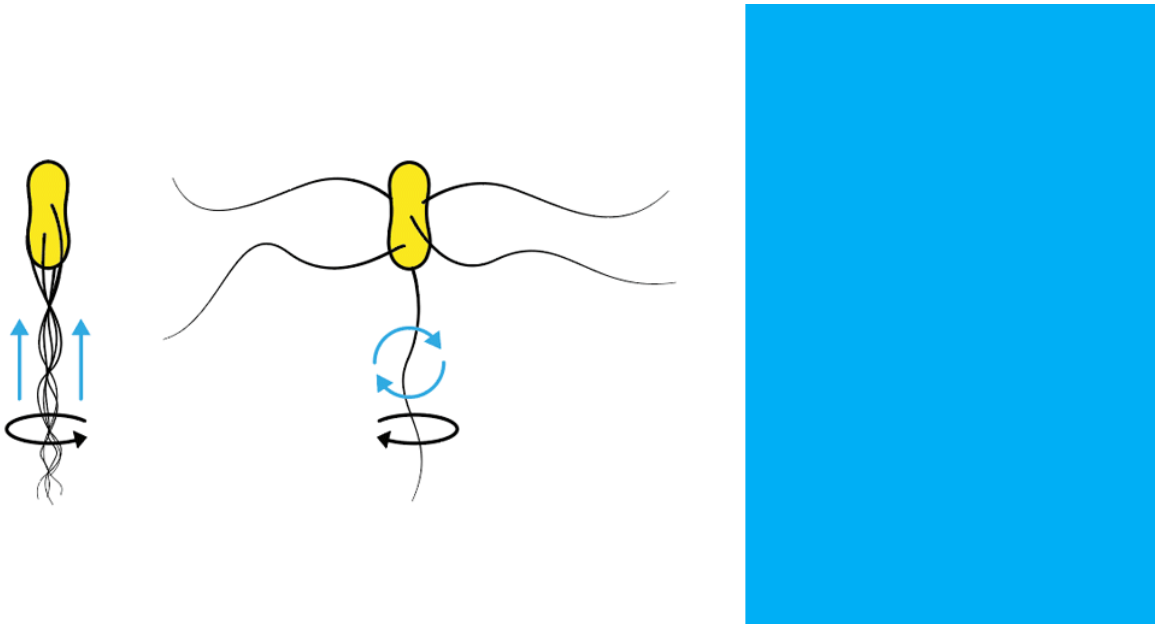


Figure 1. The direction of an *E. coli*'s flagellar rotation will determine its mode of movement.

Chemotaxis Part I: Sensing Changes in the Environment

When we think of movement toward or away from a stimulus we typically think of that movement as being directional. While a bacterium's running movement is mostly different from tumbling in that it is directional the individual cell has no actual sense of direction. The cell does not sample its environment from a specific part of its membrane. The function of *E. coli*'s chemotaxis is not to determine the absolute quantity of a ligand in its environment, but rather to detect changes in the ligand concentration between time points.

Model

In this model of bacterial chemotaxis, three 'proteins', CheA [pronounced as 'kee-ay'], CheBP, and CheR, work together to accomplish the modulation and replenishment of the tumbling signaling activity. This biological mechanism is implemented through regulating the methylation state of CheA, which in our slightly simplified model is a chemotaxis receptor (note that in a real *E. coli*, receptors and CheA are different proteins). CheR methylates CheA and CheB demethylates CheA. Methylated CheA is the active version of CheA that produces a tumbling signal. Methylated CheA also engages in kinase activity that phosphorylates CheB, activating it. This creates a negative feedback loop to keep CheA methylation level and CheB phosphorylation level in check and reset to the steady state of the model even after perturbation as long as the total concentrations of CheA, CheB and CheR in the system remain unchanged.

A note about the nomenclature in the model: in our code we have several versions of a protein that reflect the states of the protein throughout the molecular cycle that is activated during bouts of bacterial running. For the sake of brevity, we refer to CheA, CheB, and CheR as A, B, and R, respectively. A has three additional states: Am which indicates that R has methylated A, AmL which tells us that Am is bound to a ligand, and AL which tells us that AmL has been

demethylated. B has an additional state, BP, which tells us that it is currently activated by phosphorylation.

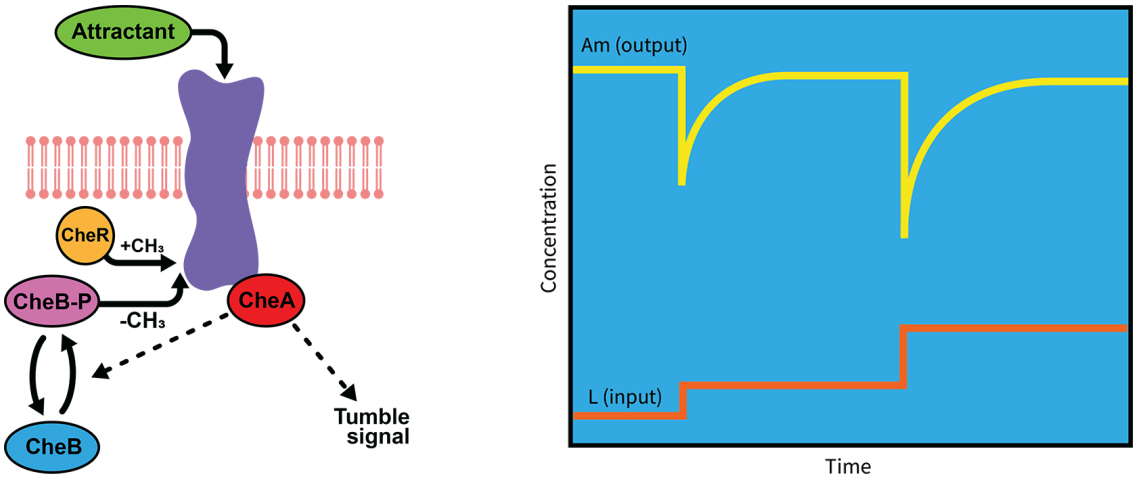
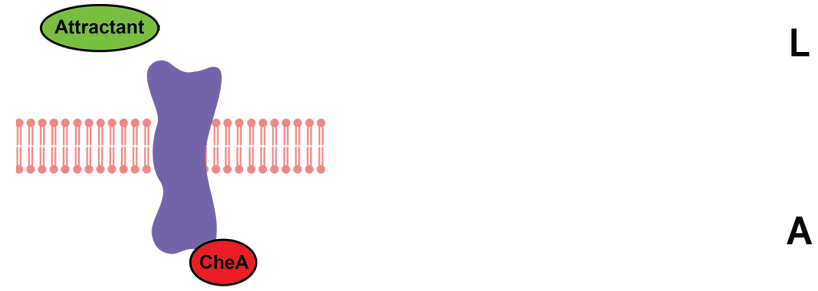
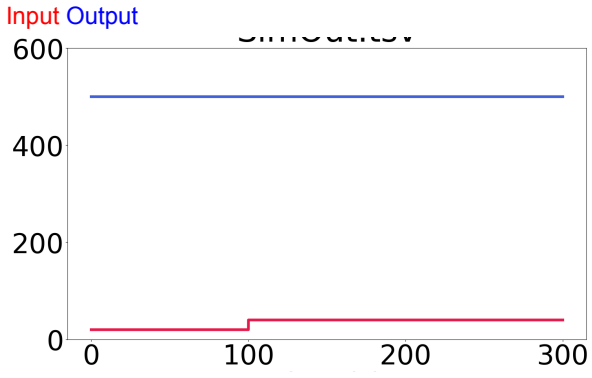


Figure 2. The system by which *E. coli* responds to changes in the concentration of a ligand in its environment relies on a network of several proteins that drive its tumbling and running movement.

We're going to take a moment to build the model step by step. We'll start with our receptor, A, and the signal in our system, L.

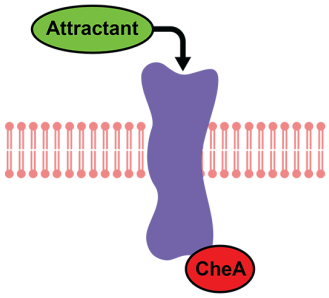


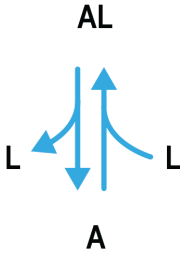


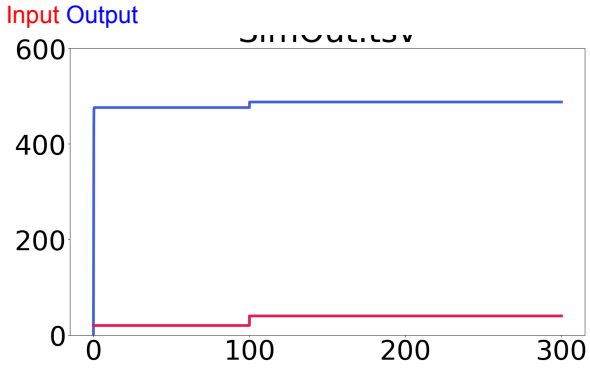

```
A = ?nM;
L[0:100] = 20nM, L[100:300] = 40nM;
```

Figure 3. The inputs and outputs of chemotaxis when only CheA and the Ligand are present with no method by which they can interact in the program.

Already this is far from sufficient to create a model of chemotaxis. We have a protein and a ligand, but we have no information that will tell our program how they will interact with one another. With that in mind, let's add a reaction that will allow A and L to interact.







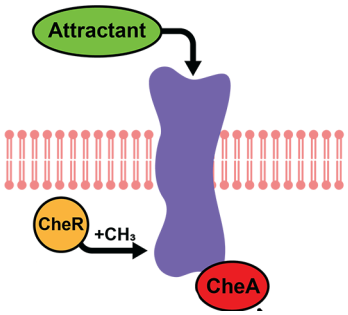

```
reaction ligand_binding2(A + L -> AL,
k=1e9, krev=1);

A = ?nM;
L[0:100] = 20nM, L[100:200] = 40nM;
```

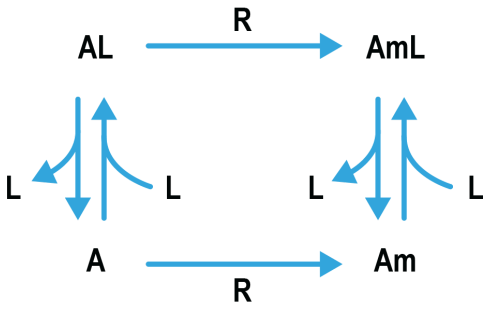
Figure 4. Introduction of a ligand binding reaction between CheA and the attractant changes the dynamics of the outputs.

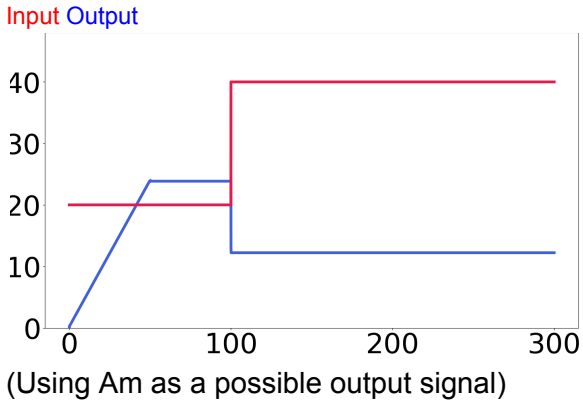
Now we're one step closer. Notice the parameters that we included in this new code. We include values of `k` and `krev` to give our reaction some kinetic values that describe both the rate of the reaction given the concentrations of A and L (`k`) and the rate of dissociation (`krev`).

Note how our syntax includes a new variable called AL. In a biological sense this variable is representing protein A after it binds to the signal, but the program interprets this as an entirely new entity that we have named AL.

This is starting to look a little better. If you recall back to the model however, you've probably already realized the next flaw in our code. We have a receptor and we have a ligand, the ligand can now bind to the receptor, but the receptor requires activation by a methyl group. Now we have a receptor that can bind to a ligand, but it won't cause any meaningful change in our system without a means by which to activate it. We fix this problem by adding a new protein to the code: R.








```
reaction ligand_binding1(Am + L -> AmL,
k=1e9, krev=1);
reaction ligand_binding2(A + L -> AL,
k=1e9, krev=1);

protein R
{
   reaction methyl_transfer1(A -> Am,
kcat=1, KM=1e-10nM);
   reaction methyl_transfer2(AL -> AmL,
kcat=1, KM=1e-10nM);
}

A = ?nM, R = ?nM;

L[0:100] = 20nM, L[100:200] = 40nM;
```

Figure 5. Introduction of CheR into the model's code allows for further regulation of the model's outputs.

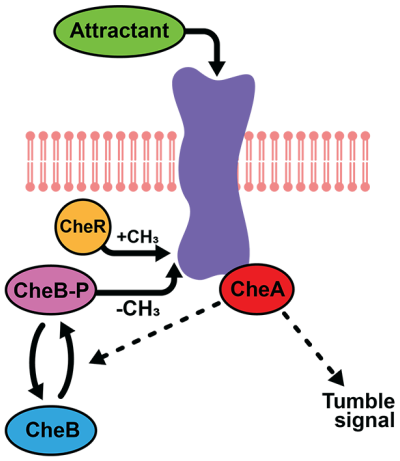
There are several things to take in now. Before you even look at the characteristics of our new protein you probably noticed that there's an additional ligand binding reaction in our program. With CheR activating CheA, our programmed A protein will now have two states, A and Am. The ligand can be bound to CheA independent of its activation state, thus we need to have another ligand binding reaction for Am.

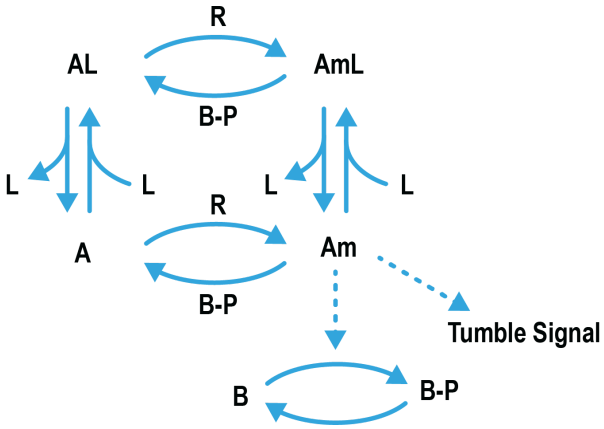
Before the ligand can bind to Am, we need to include a reaction that gives our program a way to introduce Am into it. In a biological context protein R accomplishes this by methylating A, giving us Am; however, within the program itself this can be interpreted as converting variable A into an entirely different entity, Am, just as we did with AL. This is starting to look better. We have a ligand that can bind to the receptor and we have a means to activate the receptor, but proteins that require activations sometimes require deactivation. We can code that into our system by introducing our third and final protein.

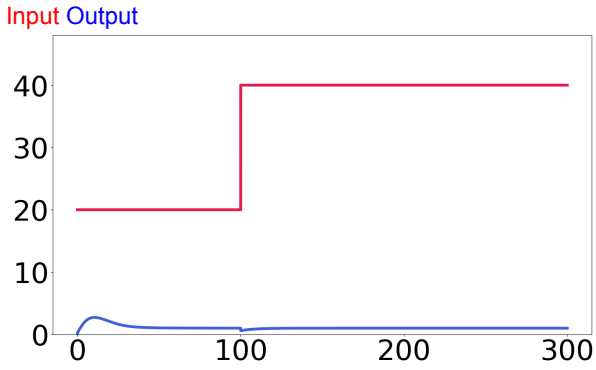
Notice how our new protein, CheB, operates in the context of our system. When active, annotated as CheBP, the protein will remove the methyl group that CheR introduced, thus deactivating the protein again.

Just like before you have probably already noticed that something is still wrong. We have a way to turn on CheA, and we have a way to turn it off. Similarly, we have a behavior coded for active CheB, but we don't have a way to activate it. Thankfully, this time around we don't require yet another protein in our system. Protein Am has an additional kinase activity that allows it to phosphorylate CheB, activating it.

Remember that we had two different states for Am: Am, and AmL. CheBP can deactivate either of these states, so we need to remember that we have to specify two reactions that CheBP can carry out. The first, named `methyl_transfer1`, removes the active state from Am, converting it back to A. In our program this will just be returning the object Am to a different object, A. The second, named `methyl_transfer2`, removes the active state from AmL, returning AL.







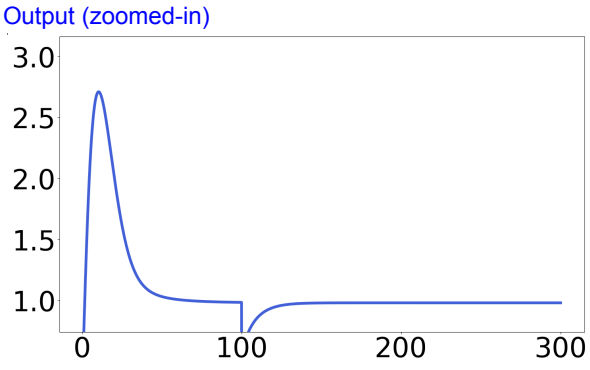


Figure 6. The complete model allows for an adaptive signal in the outputs of the model as the concentration of available attractant changes.

Chemotaxis Code 1

Bacterial Chemotaxis .lpp

```
pathway BacterialChemotaxis
{
    reaction ligand_binding1(Am + L -> AmL, k=1e9, krev=1);
    reaction ligand_binding2(A + L -> AL, k=1e9, krev=1);

    protein R
    {
       reaction methyl_transfer1(A -> Am, kcat=1, KM=1e-10nM);
       reaction methyl_transfer2(AL -> AmL, kcat=1, KM=1e-10nM);
    }

    protein BP
    {
       reaction methyl_transfer1(Am -> A, kcat=?, KM=1nM);
       reaction methyl_transfer2(AmL -> AL, kcat=1, KM=1nM);
    }

    protein Am
    {
       reaction phosphate_transfer(B-> BP, k=0.05e9, krev=0.005);
    }
}
R = ?nM, B = 0.1nM, A = ?nM;

L[0:100] = 20nM, L[100:200] = 40nM;
```

While we may not have any new proteins to include in our system now, we do have to provide one for our program. Remember that Am, the active state of CheA, is treated as a separate entity in the program, so we can include a reaction for protein Am that allows it to phosphorylate B into BP. Now we have a second switch for one of the regulatory proteins in our chemotaxis model. The regulatory effects of CheA by CheR and CheB allow for our virtual *E. coli* to generate an adaptive signal.

Once an active Am has been deactivated due to the binding of a chemoattractant (AmL), the protein can slowly be restored to its basal form. Protein BP removes the methyl group and the ligand will fall off of the receptor when it is no longer needed, leaving the cell with an inactive Protein A that can be recycled into the system through Protein R's methylation. Each of these proteins is recruited by the others in a manner similar to how the gears in a clock interact in such a way that the movement of one affects the movement of the others. The cyclical activity in this system of chemotaxis allows for the flagellar motor protein to return to its steady state after entering a part of its environment with a different level of its chemoattractant, so when it tumbles into another increase in the chemoattractant the bacterium can once again turn on its tumbling behavior.

In-class exercises

- What range of A is required to implement chemotaxis behavior?
    - Lowest quantity of A:
    - Highest quantity of A:

- How does the quantity of A affect the stability of the chemotaxis signaling?

- What range of doubling input signal (L) does the chemotaxis model respond to?
    - Lowest doubling range (e.g. 8nm to 16nm) :
    - Highest doubling range (e.g. 70nm to 140nm) :

- How does the quantity of L affect the stability of the chemotaxis signaling?

Chemotaxis Part II: Flagellar Rotation and Cell Motility

Ultimately this whole system will be affecting the movement of the *E. coli*'s flagellum, so we need to provide a flagellum in the program. Our input in the code is no longer a molecule, instead we use a boolean value. What it means is that the input, `bRun`, will have a value of either true or false. Biologically we can think of this as the flagellum rotating either counterclockwise or clockwise, so during periods of tumbling the program would consider the boolean value as clockwise = true, but during periods of running it would consider the value to be clockwise = false.

Model

Flagellar rotation code: It takes a simple boolean variable as an input and to move and rotate the physical container it is residing in.

Chemotaxis Code 2

Flagellar Rotation.lpp

```
function FlagellarRotation(bRun) // expecting a boolean input
{
    float distance = 2.5e-5; // velocity from BergBrown_1972
    float rot = rand(0, 2 * pi); // constant for rotation

    if (bRun)
    {
        position' = distance * orientation; // (x, y, z)
        orientation' = (0, 0, 0);
    }
    else
    {
        position' = dist / 3 * orientation;
        orientation'.phi = rot;
    }
}
```

There are some keywords in this code that we should cover. The **`if`**…**`else`**… conditional statements, meaning that one or the other execute based on whether or not the specified condition is met before they can determine the state of `bRun`. When the conditions for the **`if`** statement are met, the core of that statement will be executed by the program, or else the core function of the **`else`** statement will be executed instead. The **`position`** keyword will be determined by the distance constant that we included earlier in the code and the orientation of the *E. coli* using a x,y,z coordinate system. Notice the `'` operator that we include in **`position'`**. This operator is going to indicate the change in a variable or object, here we use it to tell our program about the change in the flagellum's position.

These functions will not be able to be executed until we incorporate this `FlagellarRotation` function into another code. We will be using this code in the following example where we talk about how we can implement the flagellar rotation and the adaptive signal at the cellular level.

Just like when you program a cell you can provide the petri dish with a shape and the dimensions that you want it to have. Here we chose to go with a standard 100 milimeter (mm) petri dish to house the *E. coli* as **`morphology.shape`** `= "cylinder"` and **`morphology.dimension`** `= (100mm, 100mm, 10mm)`. Include a concentration of 20 nM of the ligand, `L`. Once you have the petri dish described, add a simple line at the end to specify the quantity of petri dishes, here we use `P=1`.

Now let's make *E. coli* with both the flagellar rotation and adaptive signal required for chemotaxis. We import some of the previous code that we wrote for `BacterialChemotaxis` and `FlagellarRotation`.The critical components in this code provide a way by which the flagellum in our virtual *E. coli* will change given fluctuations in the protein system that we previously discussed. First we include a declaration for the chemotaxis program that we imported into the

code: `Chemotaxis = BC.BacterialChemotaxis`. Recall that CheAm is going to be driving the tumbling signal in this form of bacterial chemotaxis, so we provide a threshold of CheAm's concentration required to change the `bRun` value. We set the CheAm threshold value with `float threshold_CheAm = 0.982`. We add the threshold value back into `bool bRun = CheAm < threshold_CheAm` to tell our program that once the level of CheAm falls beneath the threshold value the boolean value for `bRun` will change. Now to incorporate the flagellar rotation code we previously wrote we specify that `Motility = FR.FlagellarRotation(bRun)`.

Chemotaxis Code 3

```
//Chemotaxis Code 3
import BacterialChemotaxis as BC;
import FlagellarRotation as FR;

petridish P
{
    morphology.shape = “cylinder”;
    morphology.dimension = (100mm, 100mm, 10mm);

    organism Ecoli
    {
        morphology.shape = “rod”;
        morphology.dimension = (1um, 2um, 1um);

        Chemotaxis = BC.BacterialChemotaxis;

        threshold_CheAm = 0.982nM;
        bool bRun = CheAm < threshold_CheAm;

        Motility = FR.FlagellarRotation(bRun);
    }

    L(0, 0, 0) = 20nmol;
    Ecoli(0.267, 0.1, 0) = 1;
}

P = 1;
```

If you want to expand on this code you can also add additional *E. coli* bacteria to the petri dish. Take a look at the code below:

Chemotaxis Code 4

```
// Instantiate 10 E. coli at random locations using a for loop

for (int i = 0; i < 10; i = i + 1)
{
    Ecoli(rand(-0.3, 0.3), rand(-0.3, 0.3), 0) = 1;
}
```

We can also look at chemotaxis in a petri dish that houses several virtual *E. coli* bacteria. By using iterative statements like a `for` loop here we instantiate 10 *E. coli* bacteria by declaring `(i=0; i<10; i=i+1)` where `i` represents the individual *E. coli*. Now each of the 10 *E. coli* will move through their environment in the petri dish searching for increasing levels of the chemoattractant ligand.

Not only can you use this statement to generate multiple cells, but you can spawn the cells in different positions by including coordinate information. The coordinate system is specified with the *E. coli* as `Ecoli (x, y, z)`. In this code we provide each new *E. coli* with a random space between x and y values of -0.3 and 0.3 with no specified z-value. We specify this as `Ecoli(rand(-0.3, 0.3), rand(-0.3, 0.3), 0) = 1` with the function `rand` serving as a built-in function that calls on a random value between the specified range parameters. In our petri dish coordinate system, points will fall between the range of -0.5 and 0.5 in the x and y axis, with the coordinate 0,0,0 serving as a midpoint.

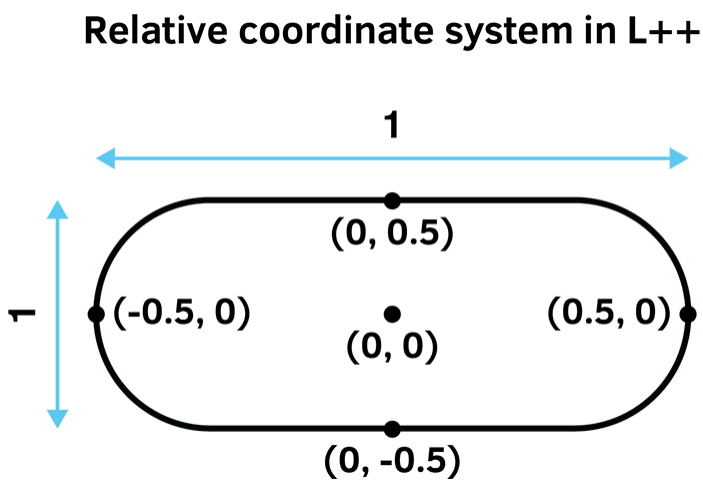


Figure 7. The coordinate system in L++ has a total range of 1 in each dimension with 0,0 serving as a midpoint.

In-class exercises

- Make 10 petri dishes with a single *E. coli* in it.
- Make a petri dish with four distinct *E. coli* with varying Threshold_Am values in it.
- What is the range of speed of an *E. coli* motility in the current chemotaxis model that would not break the current model?
    - Lowest speed:
    - Highest speed:
    - Why do you think the model breaks when you go out of the speed range? Is this a biological problem (L++ code) or an engineering problem (its output simulation software)?

Problem Set

- How can we change the model to show chemotaxis behavior in the presence of a repellent?
- How can we expand the model to show chemotaxis behavior in an environment with both chemoattractants and repellents?

References

1. Brian P. Ingalls, Mathematical Modeling in Systems Biology: An Introduction. The MIT Press. 1st edition (July 5, 2013)

Chapter 8. Cell Division

Learning Objectives

1. To understand the chemical equations are functions that process input and output analog signals in the form of molecular quantities.

Introduction

The purpose of every life form is to grow and pass on its genetic material to a new generation. This goal has yielded interest in many fields of biology ranging from the intricate mating behaviors in complex organisms to the complex molecular dynamics of division in simple organisms, with the cell encompassing the simplest form of life

When we think of division in single-celled organisms, we usually think of binary fission, a form of asexual reproduction in which one parent cell divides into two identical daughter cells. To accomplish this, the cell has to partition its genetic material equally to both poles of the parent cell before dividing at the midline. Here we will explore a model of *E. coli* binary fission and how we can implement this biological process into L++ code.

E. coli binary fission is controlled by only a couple of molecular components. The *E. coli* cell's objective during division is to keep the Z-ring, a ring of proteins that cinches around the cell to divide it, centered at its midline for even distribution of its genetic material. The Z-ring is maintained at the midline by oscillations of proteins throughout the cell that move from one cell pole to the opposite pole in twenty second time intervals. The oscillations create an inhibitory zone that prevents the Z-ring from deviating from the cell's midline. This system for positioning the Z-ring is called the Min System.

The Min System in *E. coli* Binary fission relies on three proteins: MinC, MinD, and MinE. MinC prevents the Z-Ring from forming at the cell poles by inhibiting the polymerization of FtsZ. MinC is recruited to the cell membrane by associating with MinD. Both MinC and MinD are pushed off of the membrane by MinE, which continually follows MinD.

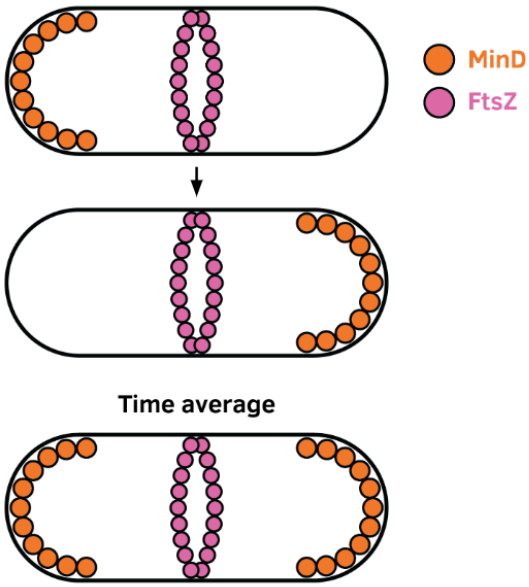


Figure 1. *E. coli* cell division relies on FtsZ to be more concentrated at the midline of the cell.

Throughout the tutorial we will look at questions addressing why this system of binary fission is used and why some of these components are necessary.

Cell Division Part I: Midline Determination

Determination of the midline in a cell prior to and during cell division is vital to the process being carried out correctly. Without a midline at which the Z-ring can form, the cell may generate daughter cells with uneven amounts of subcellular components or it may form multiple Z-rings at different locations of the cell. Any deviation from the midline will result in less viable daughter cells.

The model we will be using for *E. coli* binary fission is not completely biologically accurate, as we still have much to learn about the process of cell division. George Box, a British statistician, put it best when he said that “all models are wrong, but some are useful.” Despite the innate inaccuracies of models, we can still use them to gain some insights into the dynamics of various proteins and how we can code complex processes using L++.

Model

Add a plot for specification of Min system binary fission

As we discussed previously for bacterial chemotaxis, cell division has its own inputs and outputs that we have to consider. Within our program, consider adding MinD to the *E. coli*’s cytosol to serve as an input, as the activity of the other Min proteins will depend on MinD. Once it is in the cytosol we can think of either membrane-bound MinD, or MinC as the output, and fluctuations in this output can be measured to assess the molecular dynamics of cell division.

In the following exercise we will use this system of molecule inputs and outputs to design the Min system and the signals that it uses to regulate cell division. As we look at each component of this system we will add them to the code that designates the formation of the Z-ring and FtsZ to assess how the Min proteins affect cell division.

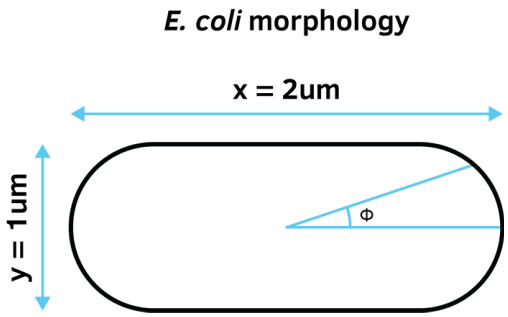


Figure 2. The Phi angle in a cell is used to model the trajectory of MinD as it gets recruited to the cell membrane.

Code 1

CellDivision .lpp

Tip: This CellDivision .lpp will be the file that you will copy and paste subsequent code into.

```
// Code 1
// CellDivision.lpp
organism Ecoli
{
    morphology.shape = “rod”;
    morphology.dimension = (1um, 2um, 1um);

    membrane Membrane
    {
        // reactions involving membrane can be added here

        cytosol Cytosol
        {
            MinC = 163;
            MinD = 4928;
            MinE = ?;
            FtsA = 792;
            FtsZ = 7898;
        }
    }
}
Ecoli = 1;
```

We can keep the description of the *E. coli* short, providing the program with a cell membrane and a generalized description of its rod shape. Additionally we provided the cell with some

additional layers– the cell `Membrane` and `Cytosol`. In our code these layers will serve as containers for the proteins involved in cell division. Notice that below `Cytosol` there are lines to specify the quantity of each protein that we will use in the program.

Now we can start describing the protein interactions in our code. We will start with `MinD`, which will serve as an input into the cytosol of our virtual cell.

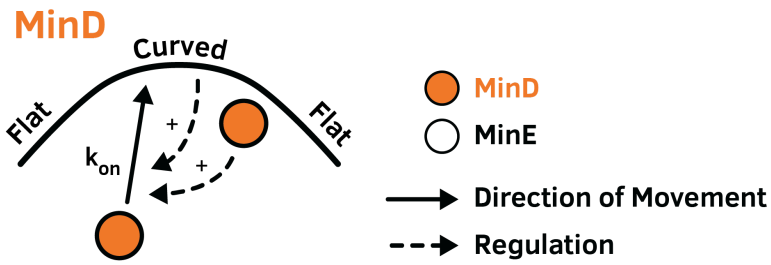


Figure 3. MinD is recruited to the *E. coli* cell membrane during cell division.

Code 2

```
// copy the following code into the CellDivision.Ecoli.Membrane code block

reaction MinD_MembraneRecruitment(Membrane~MinD_{n} + MinD{in} ->
Membrane~MinD_{n+1})
k_function
{
    MaxMembraneEffect = (0.5 * pi) ** 2;
    NewAngle = Membrane.phi; // phi: positional reference from the center of the
membrane shape
    if (NewAngle >= pi)
    {
        NewAngle = NewAngle - pi;
    }
    MembraneEffect = (NewAngle - 0.5 * pi) ** 2 / MaxMembraneEffect * 1.1;
    AggregationEffect = log(max(2, Membrane~MinD_{n}), 2); // log() assuming base
10
    return MembraneEffect * AggregationEffect;
}
```

Here we name the membrane binding reaction `MinD_MembraneRecruitment`. The reaction is described by `(Membrane~MinD_{n} + MinD {in} -> Membrane~MinD_{n+1}` and at first it may look like there is a lot going on here but it's quite simple. Notice the use of `~` operator in the code, this identifies two objects as being bound together so our first component of this reaction, `Membrane~MinD_{n}`, is telling us that `MinD_{n}` is binding to the cell membrane. What follows is additional MinD being recruited to the cell membrane which we designate as `MinD{in}`, thus the

product of the reaction is simplified as the addition of MinD by `Membrane~MinD_{n+1}`. As similarly found in our previous coding exercises, we'll use a `k_function` here to describe the reaction's kinetics.

We have several components of this code to discuss so we can get a nice picture of how our program is interpreting the code. First let's take a look at `MaxMembraneEffect`. The membrane effect in this code is describing the recruitment of MinD to the cell membrane. Note how the membrane effect is described as `(NewAngle - 0.5 * pi) ** 2`. MinD will only ever be bound to the membrane of one half of the cell at a time, therefore it is necessary to restrict it to half of the circumference of the cell. Our `Membrane.phi` value describes the angle of recruitment of MinD to the cell membrane measured as the angle taken from the center of the cell to the membrane. In our model, we assume MinD preferentially binds to the furthermost ends of the membrane, i.e. the pole. Thus MinD will travel along the poles where the angles are the highest degree, leading more molecules to prefer the cell pole than areas close to the midline.

In addition to the membrane effect, our model contains the `AggregationEffect`. MinD does not form a singular layer around the cell membrane, but instead it will also form aggregates most preferentially at the cell pole. The aggregation of MinD amplifies its activity near the polar region of the cell and further moves the activity of the Min system away from the cell's midline.

Now we need to bring MinE to the cell membrane to remove MinD.

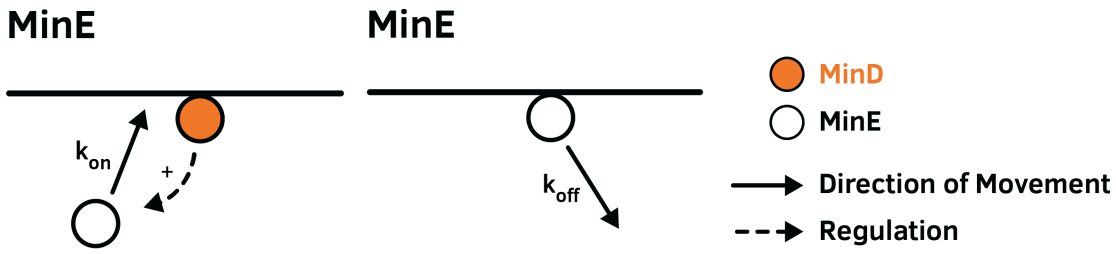


Figure 4. MinE is recruited to the cell membrane by MinD.

Code 3

```
reaction MinE_MembraneRecruitment(Membrane~MinD_{n}~MinE_{m} + MinE{in} ->
Membrane~MinD_{n}~MinE_{m+1})
k_function
{
    return MinD_MembraneRecruitment.k_function() * 0.2;
}
krev_function
{
    return max(0, Membrane~MinE_{n} - Membrane~MinD_{m});
}
```

We can describe that incoming MinE will be recruited to the membrane binding it to membrane-bound MinD as `Membrane~MinD_{n}~MinE_{m} + MinE{in}`. The products of this reaction could be described as the cell membrane bound to MinD with additional MinE bound to MinD as `Membrane~MinD_{n}~MinE_{m+1}`.

Now let's look at the energetic functions that we use to describe MinE's molecular effects. Think about how we can describe MinE's activity in the code. Because we provided a k function for MinD's membrane recruitment, we added a new krev function that describes MinE kicking MinD off of the membrane.

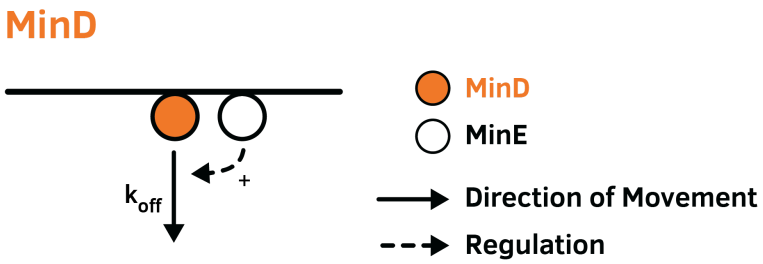


Figure 4. MinE follows MinD, kicking it off of the cell membrane.

Code 4

```
reaction MinD_MembraneRecruitment(Membrane~MinD_{n} + MinD -> Membrane~MinD_{n+1})
k_function
{
    ... // see Code 2
}
krev_function
{
    return k_function() * (Membrane~MinD_{n}~MinE_{m} / max(1, Membrane~MinD_{n}))
* random_normal(1.5, 0.2); // normal(mean,SD)
}
```

Given what we know about life programming, how should we implement the behavior of MinC into the program? Assume the membrane is already declared in our program because we will be using MinC alongside the rest of the Min system proteins.

We know that MinC is going to interact with MinD at the cell membrane so by now the reaction that we need to write to describe this interaction should come easily. (identical to MinE)

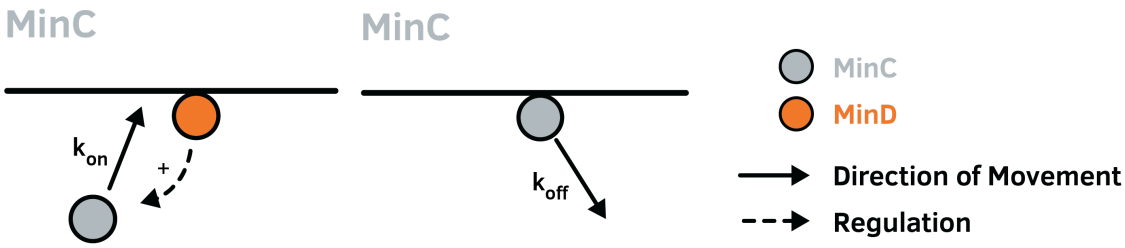


Figure 4. MinC's membrane recruitment depends on MinD at the cell membrane.

Code 4

```
// copy the following code into the CellDivision.Ecoli.Membrane code block

reaction MinC_MembraneRecruitment(Membrane~MinD_{n} + MinC{in} ->
Membrane~MinD_{n}~MinC)
k_function
{
    return MinD_MembraneRecruitment.k_function() * 0.2;
}
krev_function
{
    return max(0, Membrane~MinC_{n} - Membrane~MinD_{m});
}
```

Just like before we name the reaction, in this case `MinC_MembraneRecruitment`,will describe how we recruit MinC to membrane-bound MinD resulting in MinC binding to the complex like so: `Membrane~MinD_{n} + MinC{in} -> Membrane~MinD_{n}~MinC`. Once again, after specifying the reaction that will take place we provide k functions to give some energetic contexts to the reaction. First, our **`k_function`** describes MinD's membrane recruitment again because just like we specified with MinE, the function of MinC is not meaningful until it can bind to membrane-bound MinD. Second, we write a **`krev_function`** that describes MinC's dissociation from MinD as it will eventually fall off of MinD once MinE kicks MinD off of the membrane.

So far it's been pretty simple, right? Each of these proteins, their associated reactions, and the role that they play in *E. coli* binary fission can be described in only a few lines of code, allowing us to keep this system clear and succinct. We have described the ways by which the cell senses changes in its environment, but a model of chemotaxis also requires movement. Next we will look at how we can program the *E. coli*'s flagellum.

In-class exercises

- Play around with the number of molecules. How is oscillation behavior affected by their quantity?
    - Oscillation
    - How long it takes to start oscillating
    - Periodicity
    - Amplitude
- Another model is to statically keep MinC to the tips of E coli. Would it be a “viable” strategy in a real E coli? Why or why not?

Cell Division Part II: Cytokinesis

Finally we can see how binary fission will work in *E. coli* when we incorporate our program for Z-ring formation and the membrane recruitment of MinD, MinE, and MinC. By now we are familiar with the role of each of these four components in this process of cell division, how they function, and how they should come together to build the Min system. We have the Min proteins stored in another file called MinCDE that we can import along with the Divisome folder where we have our program for the function of the Z-ring. Because the Min proteins work together to prevent formation of ectopic Z-rings away from the midline, we designate the pathway with these proteins as `MidlineDetermination`. The midline determination pathway designates the recruitment of each of the Min proteins to the cell membrane.

Model

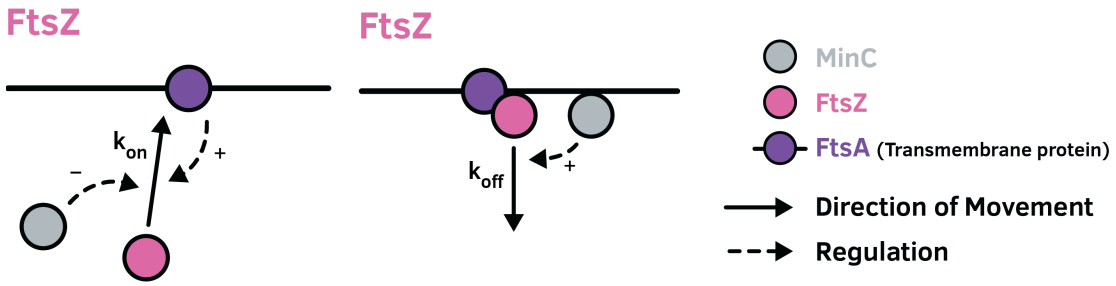


Figure 4. FtsZ is recruited to the cell membrane by a transmembrane protein, FtsA. MinC downregulates FtsZ’s membrane recruitment.

Code 5

```
// copy the following code into the CellDivision.Ecoli.Membrane code block

pathway Cytokinesis
{
    reaction FtsZ_Recruitment(Membrane~FtsA~(FtsZ_{n}) + FtsZ ->
                                    Membrane~FtsA~(FtsZ_{n+1}) // n can be 0
    k_function
```

```
    {
        return FtsA / max(1, MinCDE.MinCDE.MinC);

    }
    krev_function
    {
        return 0.5 + Membrane~MinCDE.MinCDE.MinC;
    }

    reaction Crosslinking(2 Membrane~FtsA~(FtsZ_{n}) -> Membrane~FtsA~(FtsZ_{2 *
n}));
    specification
    {
        float rate = 0;
        if (n <= 64)
        {
            rate = 1 * n;
        }
        Membrane~FtsA~(FtsZ_{2 * n})' = floor(rate);
    }

    reaction Constriction(Membrane)
    specification
    {
        float force = 0;
        for (int i = 4; i < 128; i = i * 2)
        {
            force = force + Membrane~FtsA~(FtsZ_{i}) * 0.00001 * i;
        }
        Membrane.radius' = -force;
    }
}
```

If we design the Z-ring there are two key proteins that we have to include, `FtsA` and `FtsZ`. `FtsA` will help anchor the Z-ring and recruit `FtsZ` to the Z-ring while `FtsZ` is going to be the primary component of the ring. We can reflect this in the reaction that we have designed to recruit FtsZ for assembly of the Z-ring which we have named `FtsZ_MembraneRecruitment`. We use the binding operator `~` to describe the binding of the proteins to each other and the membrane as:`Membrane~FtsA~(FtsZ_{n}) + FtsZ{in}` . The output of the reaction is described as the addition of a FtsZ protein which we annotate as n+1 in: `Membrane~FtsA~(FtsZ_{n+1}`.

We have two additional reactions that help form the Z-ring, `reaction Crosslinking` and `reaction Constriction`. `Crosslinking` describes the addition of additional FtsZ molecules to the Z-ring by the doubling of membrane-bound FtsZ after its recruitment by the reaction `2 Membrane~FtsA~(FtsZ_{n}) -> Membrane~FtsA~(FtsZ_{2 * n})`. The products of the reaction, `Membrane~FtsA~(FtsZ_{2 * n})`, are rounded down to the nearest integer using the `floor` keyword.

`Constriction` gives us a description of how the Z-ring exerts force on the cell membrane. We use a `for` loop to iterate this program over itself until a value of `i` exceeds an arbitrary value of `128`, doubling with each iteration described by `i = i * 2`. We designate a value called `force` that is amplified by the length of the Z-ring that we describe using `Membrane~FtsA~(FtsZ_{i}`. Multiplying this value by a small value like `0.00001` keeps the force at a small scale because we are working with subcellular molecular dynamics. Lastly the radius of the membrane will be shortened by subtracting the `force` value from it through each loop. In short, the force of the Z-ring is going to be greater as the Z-ring elongates, allowing for constriction of the cell membrane as the cell grows closer to dividing.

We include another pathway, `Cytokinesis`, that has the information that our program requires to construct the Z-ring as the *E. coli* divides, including the recruitment of FtsZ to the cell membrane and the crosslinking and treadmilling required to help facilitate the formation of the Z-ring. Lastly we include quantities of each protein that allow for simulation of natural binary fission, but you can change these quantities in your own code to observe how changes in each molecule may affect the cell division process.

Although the Min system of cell division is only one method by which cells divide, it is one that requires a delicate balance between each component to allow for the formation of two healthy daughter cells once the parent cell has divided. Cell division has changed throughout evolutionary history to have many methods that accommodate for differences in cells including eukaryotic cell division methods that use mitotic and meiotic pathways, or even reproductive methods in yeast in which the parent cell produces a smaller bud that will break off into a daughter cell.

In-class exercises

- Play around with the number of FtsZ molecules. How is oscillation behavior affected by their quantity?

Problem Set

- How can we achieve these same results with only two Min proteins?
- How does the new model challenge biological implementation of the design?

Chapter 9. *E. coli*, Population Growth and Perturbations

Learning Objectives

1. To understand the minimal set of biological functions to perform to have a self-sufficient lifeform, such as *E. coli.*
2. To understand that population growth of *E. coli* can be a readout of their overall biological performance and fitness.
3. To learn the ways to introduce perturbation and treatment of the virtual organisms.

Introduction

Population growth has long been an interest in modeling biological processes. Since the simplest model for population growth, the model of exponential growth, was developed, models of growth have been continually molded and developed to describe the change in a population over time with various limits, perturbations, and interactions. The growth of model organism populations in given settings has been a valuable tool for molecular biologists, as very often we do not understand how something works until we break it. Scientists have employed methods of introducing breaks into the genetic code of organisms like *E. coli* and measuring how deficiencies in those genes affect the growth of the organism.

E. coli are an interesting organism with respect to population growth. In optimal conditions a single *E. coli* will divide roughly every twenty minutes, and with such a short turnover in *E. coli* generations they can rapidly expand in their environment. This rate of growth has made them a critical model organism in the scientific community as a vector for replicating recombinant DNA. In this process, scientists also introduce perturbations to the population in the form of antibiotic compounds in the environment. Here we will discuss how we can model the population growth of virtual *E. coli* and how we can simulate perturbations to experimentally deter their population growth.

We can accurately model the population growth of wild-type *E. coli* in a petri dish containing the required nutrients to reproduce. In these simulations we can simulate the same time scale of each *E. coli generation* (~20 minutes per generation), while speeding up the computational time. The above figure shows us 30 generations of population growth of *E. coli* computed over 1 minute.

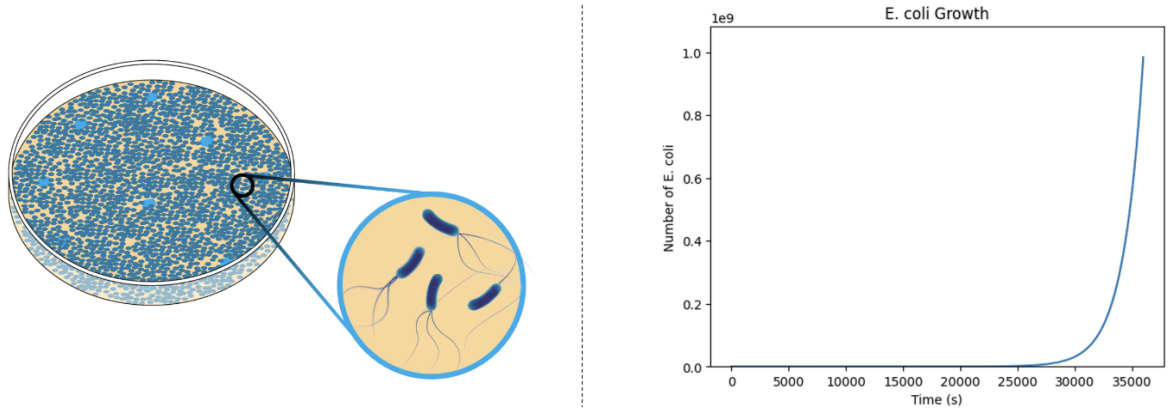


Figure 1. Experimental control of virtual *E. coli* population growth showing 30 generations of *E. coli.*

E. coli Part 1: Building the Cell

E. coli is a unicellular organism that can efficiently metabolize food, transforming it into energy and essential building blocks for biomass generation. This dynamic process underscores the organism's capacity to sustain itself. The biomass that *E. coli* accumulates is used to replicate itself (Figure 1). E. coli can double its cell population rapidly via binary fission, emphasizing the organism's profound adaptability and resilience, key attributes that have made it a vital model organism in microbiology and genetics. Essentially, writing L++ code for E coli will make a robust and self-sufficient single cell organism capable of all the basic cellular functions that is common to most prokaryotic organisms.

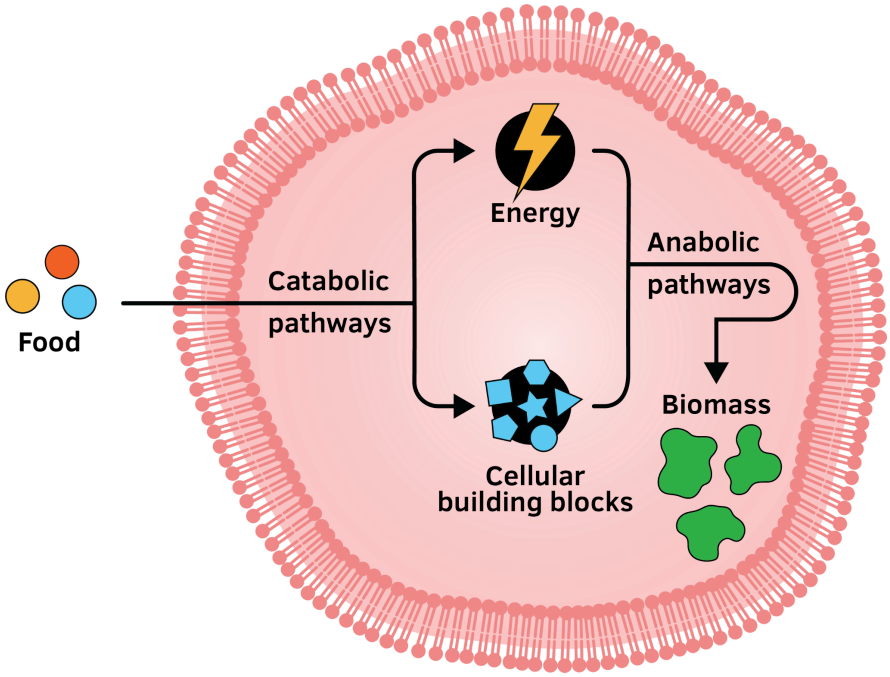


Figure 2. Basic cellular functions.

Model

Setting up structure and molecular compositions of *E. coli*

In the following code we are going to look at how we can thoroughly describe an *E. coli* cell in L++ code. To simplify things we import available programs for essential cellular processes, `Metabolism`, `CellDivision`, and `GeneExpression`. Before we get into the details, consider the layers, or scopes, present in the code. The first part of this code has four layers: `Ecoli`, `OuterMembrane`, `InnerMembrane`, and `Cytoplasm`. Within each scope we provide information that helps us describe various components of the cell. The `Ecoli` scope contains information about the genome (`genome = dsDNA(4.5e6)`) and morphology (`morphology.shape = "rod"; morphology.dimension = (2um, 1um, 1um)`).

Both membranes, `OuterMembrane` and `InnerMembrane`, contain information on the molecular composition of the membranes. Each description provides information about the molecular building blocks of the membrane and the percentage of the membrane that is composed of that molecule. `phosphatidylethanolamine = 0.78` tells us that 78% of the membrane is composed of phosphatidylethanolamine, followed by `phosphatidylglycerol = 0.12` and `cardiolipin = 0.06`. These values clearly don't make up the full percentage, so we fill the rest of the membrane in with `default = diacylglycerol_phosphate`. Lastly we have the `Cytoplasm` that we use to encapsulate the cellular processes that we imported earlier.

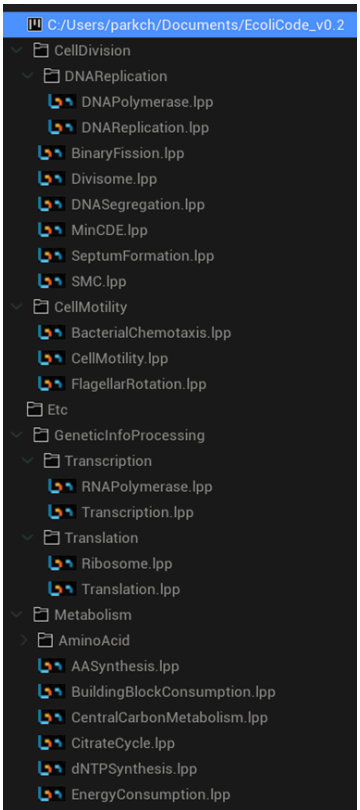
Code 1

```
import Metabolism;
import CellDivision.BinaryFission;
import GeneticInfoProcessing;

bacterium Ecoli
{
    genome = dsDNA(4.5e6);

    morphology.shape = "rod";
    morphology.dimension = (2um, 1um, 1um);
    default = H2O; // water to be used to fill the organism after dry molecules

    membrane OuterMembrane
    {
        phosphatidylethanolamine = 0.78;
        phosphatidylglycerol = 0.12;
        cardiolipin = 0.06;
        default = diacylglycerol_phosphate;

        membrane InnerMembrane
        {
```

```
            phosphatidylethanolamine = 0.78;
            phosphatidylglycerol = 0.12;
            cardiolipin = 0.06;
            default = diacylglycerol_phosphate;

            cytoplasm Cytoplasm
            {
                Metabolism = Metabolism.MetabolicPathway;
                CellDivision = CellDivision.BinaryFission;
                GeneExpression = GeneticInfoProcessing.GeneExpression;

                // reference for ThyA perturbation
                ThyA.DNA.promoter = 0.02;
                ThyA.RNA.degradation = 0.005;
                ThyA.RNA.RBS = 10.176;
                ThyA.degradation = 0.008;

            } = 1;
        } = 1;
    } = 1;
}

petridish P
{
    Ecoli(0, 0, 0) = 1;
} = 1;
```

Cellular processes E coli code

We have implemented essential cellular processes, `Metabolism`, `CellDivision`, and `GeneExpression`. Currently, there are a total of 45 files, 114 of reactions, 200 of molecules in the *E coli* code. More specifically:



In `Metabolism`, we included the following metabolic pathways:

- `CentralCarbonMetabolism` breaks down glucose as a main food source to generate energy (ATP) upon energy demand and intermediate metabolites and building blocks to support other cellular processes. The module includes `Glycolysis`, `PyruvateOxidation`, `CitricAcidCycle`, `OxidativePhosphorylation`, `PentosePhosphatePathway`, etc.
- `NucleotideSynthesis` generates purine and pyrimidine nucleotides from PRPP needed to support DNA replication, transcription and other nucleotide consuming processes.

- `AminoAcidSynthesis` generates 20 amino acids from their precursor metabolites to support translation and nitrogenous metabolism.
- `CofactorSynthesis` generates cofactors required in metabolic reactions, such as NAD, Flavin, FolicAcid, CoA, and acyl carrier proteins.
- `FattyAcidMetabolism` generates phospholipids from acetyl-CoA, which make up the membrane.

Most of the metabolic reactions are formulated with the high level description using **`specification`** syntax with product inhibition model to assume rapid growth of E coli every twenty minutes. We have incorporated enzymes that can facilitate the major metabolic pathways found in a recent preprint from the Reynolds lab at UTSW.

In `CellDivision`, we included cytokinesis by FtsZ ring and midline determination system by MinCDE discussed in the Cell Division tutorial, as well as DNA Replication. In `GeneExpression`, we included transcription and translation and RNA and protein degradation.

Regarding gene expression, we are able to control gene expression parameters for synthesis and degradation of protein and its RNA via accessing a few gene expression-related syntaxes. The original parameters are currently set in the ProteinConcentrations.lpp. Those values can be modified (overwritten) when importing them into the `Ecoli` code. For example, the experiment we are interested in is to perturb ThyA expression, so we will include information about the activity of thyA gene's promoter for controlling its rate of transcription (`ThyA.DNA.promoter = 0.02`), ribosome binding site (RBS) of ThyA mRNA for controlling its rate of translation (`ThyA.RNA.RBS = 10.176`), as well as the degradation of the its protein (`ThyA.degradation = 0.008`), and the degradation of the RNA that codes for the protein (`ThyA.RNA.degradation = 0.005`).

Lastly we need to put the *E. coli* in something, so we provide the code with a petri dish just as we did in the chemotaxis tutorial (**`petridish`** `P`), spawning the *E. coli* in the center of the dish (`Ecoli(0, 0, 0) = 1`).

With the cell described we can provide additional information that can let us assess the effects of ThyA on the cell, just specifying how ThyA is processed in the central dogma of biology will not be sufficient to see any effects of its perturbation. We import the modules `ProteinConcentrations` and `EcoliInfo` that contain additional information about *E. coli*.

By now we know that ThyA functions as an intermediate step in the synthesis of Thymine, a pyrimidine essential to DNA. We've incorporated a short code providing the program with information about pyrimidine metabolism, specifically thymine production in which deoxyuridine triphosphate (dUTP) and 5,10 methylenetetrahydrofolate (methylene-THF) are used to produce deoxythymidine triphosphate with a byproduct of DHF (`reaction dTTPSynthesisByThyA(dUTP + H2O + 2 ATP + 5,10-methylene-THF -> dTTP + PPi + H + DHF + 2 ADP)`). Specifying this reaction as one that is carried out by ThyA gives the program a way to functionally knockdown ThyA given the other cellular processes that we are including in the code.

Code 2

```
import ProteinConcentrations;
import EcoliInfo;

pathway PyrimidineMetabolism
{
    ...

    reaction dTTPSynthesisByThyA(dUTP + H2O + 2 ATP + 5,10-methylene-THF -> dTTP +
PPi + H + DHF + 2 ADP)
    specification
    {
        float rate = EcoliInfo.DNAReplicationRate * EcoliInfo.C2M * 0.5;
        rate = rate * (ThyA / ProteinConcentrations.ThyA);
        dTTP' = rate;
    }
}
```

The next code is going to be what we use to build a way by which our program can simulate essential cellular processes that are required for ThyA's activity in pyrimidine synthesis. After importing essential processes for DNA replication including several metabolic and synthetic pathways, the pathway in which they will be used can be specified. The pathway which is simply named `Metabolism` includes these imported modules to designate cellular processes essential to cell health and ThyA's function prior to the virtual knockout experiment.

Code 3
Metabolism.lpp

```
from MetaboliteConcentrations import *;
from ProteinConcentrations import *;
```

```
import MetaboliteConcentrations;
import CentralCarbonMetabolism.CentralCarbonMetabolism;
import Nucleotide.NucleotideSynthesis;
import AminoAcid.AminoAcidSynthesis;
import Cofactor.CofactorSynthesis;
import FattyAcid.FattyAcidMetabolism;
import EnergyConsumption;

pathway Metabolism
{
    CentralCarbonMetabolism =
CentralCarbonMetabolism.CentralCarbonMetabolism.CentralCarbonMetabolism;
    NucleotideSynthesis = Nucleotide.NucleotideSynthesis.NucleotideSynthesis;
    AASynthesis = AminoAcid.AminoAcidSynthesis.AminoAcidSynthesis;
    CofactorSynthesis = Cofactor.CofactorSynthesis.CofactorSynthesis;
    FattyAcidMetabolism = FattyAcid.FattyAcidMetabolism.FattyAcidMetabolism;
    EnergyConsumption = EnergyConsumption.EnergyConsumption;

    // nutrients
    glucose[:] = MetaboliteConcentrations.glucose; // source of carbon
    G6P[:] = MetaboliteConcentrations.G6P; // source of carbon
    sulfate[:] = MetaboliteConcentrations.sulfate; // source of sulfur
    ammonium[:] = MetaboliteConcentrations.ammonium; // source of nitrogen
    Pi[:] = MetaboliteConcentrations.Pi; // source of phosphorus

    // currently unbalanced metabolites
    nicotinamide[:] = MetaboliteConcentrations.nicotinamide;
    acetate[:] = MetaboliteConcentrations.acetate;
    formate[:] = MetaboliteConcentrations.formate;
    M_3_5_ADP[:] = MetaboliteConcentrations.M_3_5_ADP;
    ARP[:] = MetaboliteConcentrations.ARP;
    PPi[:] = MetaboliteConcentrations.PPi;
    glycoaldehyde[:] = MetaboliteConcentrations.glycoaldehyde;

    H2O[:] = MetaboliteConcentrations.H2O;
    H[:] = MetaboliteConcentrations.H;
    OH[:] = MetaboliteConcentrations.OH;
    H3O[:] = MetaboliteConcentrations.H3O;
```

```
    O2[:] = MetaboliteConcentrations.O2;
    CO2[:] = MetaboliteConcentrations.CO2;
}
```

In-class exercises

- ThyA 80% knockdown starts giving phenotype in experiments. How would you adjust the dTTPSynthesis reaction code to make it reflective of the experimental data?

E. coli Part 2: Virtual Perturbation Experiments

Before we get started on simulating population growth, we can discuss a gene that we will perturb. ThyA, a gene that codes for Thymidine synthase, is necessary for synthesis of Thymine, a nucleotide found in DNA. Thymidine synthase is required for the conversion of dUMP, deoxyuridine monophosphate, into dTMP, deoxythymidine monophosphate, prior to synthesizing Thymine. We should inuit then, that perturbations in ThyA will affect the genetic structure of the *E. coli* and hence dampen their growth.

Now that we have a good understanding of the effects of experimentally perturbing a gene in an L++ program, we can take a look at how we implement these experiments in the code.

Model

We have looked at how we can graphically represent *E. coli* population growth and the effects of genetic perturbations on the population, but now we are going to look at how we can design a program that can be used to generate a virtual *E. coli* population and alter it to satisfy experimental conditions. Throughout this exercise we will build a virtual bacterium and introduce defects in its genomes that may affect population growth.

With our virtual cell realized in the program and the pathways that affect and are affected by ThyA included to contextualize our experiment we are ready to perform our virtual experiment. Here we use a **for** loop to initialize 30 replicates of our experiment (`for (int i = 0; i < 30; i = i + 1)`) . In each of the replicates we will have a petri dish (`petridish P`) that contains our *E. coli* with varying perturbations of ThyA (`organism Ecoli`). The program additionally initializes the cell membrane and cytosol for each *E. coli*, as the *E. coli* will grow and divide so the individual organisms in the program require individual membranes to stay separated.

Code 4

```
from ProteinConcentrations import *;
from MetaboliteConcentrations import *;
for (int i = 0; i < 30; i = i + 1)
```

```
{
    petridish P
    {
        organism Ecoli
        {
            Metabolism = Metabolism.Metabolism.Metabolism;
            Transcription =
GeneticInfoProcessing.Transcription.Transcription.Transcription;
            Translation =
GeneticInfoProcessing.Translation.Translation.Translation;

            ThyA.DNA.promoter = 1.0 - i / 29;
        }(0, 0, 0) = 1;
    } = 1;
}
```

The experimental variation in this program is defined in a single line of code: `Ecoli.ThyA = ProteinConcentrations.ThyA * (1.0 - i / 29)`. In this line we tell the program that ThyA is expressed within the cell, and the concentration of ThyA will vary in each virtual replicate. The concentration of ThyA will be representative of its expression within the cell and we will be able to monitor the growth in the *E. coli* populations as ThyA is gradually repressed from its natural concentration.

There are various ways that you can perturb a gene. Depending on your method and expected outcome, you can completely remove a functional gene from the target's genome, a knockout, or you can repress it to varying degrees as a knockdown.

The knockout will be a perturbation with the strongest effect on a gene. A knockout will either completely disable the function of a gene, or using methods like CRISPR recombination remove the genetic sequence from the genome entirely. If the knockout is successful, the target gene will have no expression. These methods are typically used to test whether a gene or protein is necessary for a biological process.

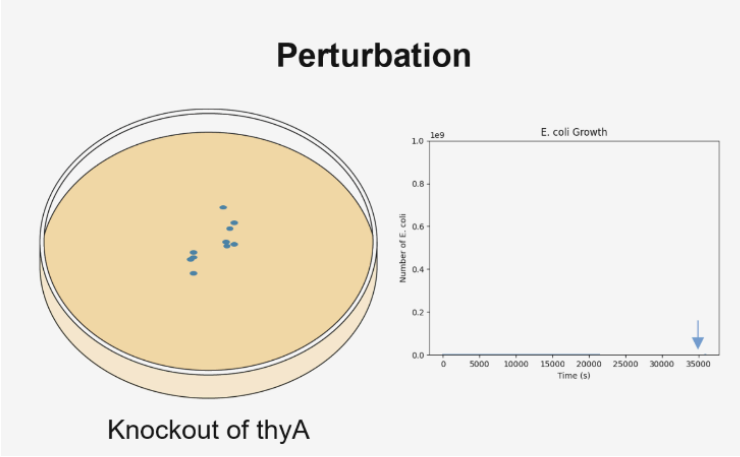


Figure 2. Complete knockout of ThyA in L++ code severely inhibits *E. coli* population growth.

Take a knockout of ThyA for example. If we compare this to our control for population growth it's safe to say that ThyA and hence, Thymine synthesis, is necessary for proper growth of *E. coli*.

Now consider how a knockdown will differ from a knockout. With a knockdown we can implement varying degrees of repression on a gene or protein to test not whether or not the gene is necessary to achieve a certain phenotype, but if it is sufficient to achieve the phenotype.

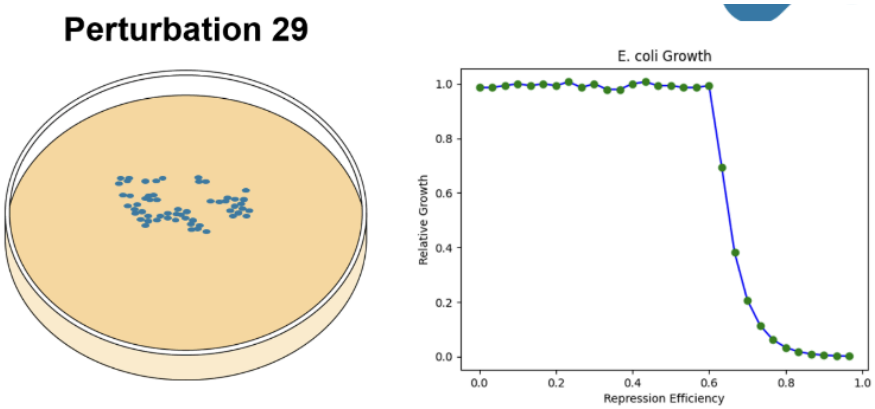


Figure 3. L++ supports many ways to track the effects of an experiment, here is the relative growth of the population plotted against the efficiency of repression in ThyA expression in virtual *E. coli.*

Here we have an example of both a measure of the repression efficiency of a knockdown of ThyA and its effect on a population of *E. coli.* Notice how the dish is labeled perturbation 29. Using our simulation software you can run replicates of experiments simultaneously and receive real-time data for each replicate as the virtual *E. coli* reproduces.

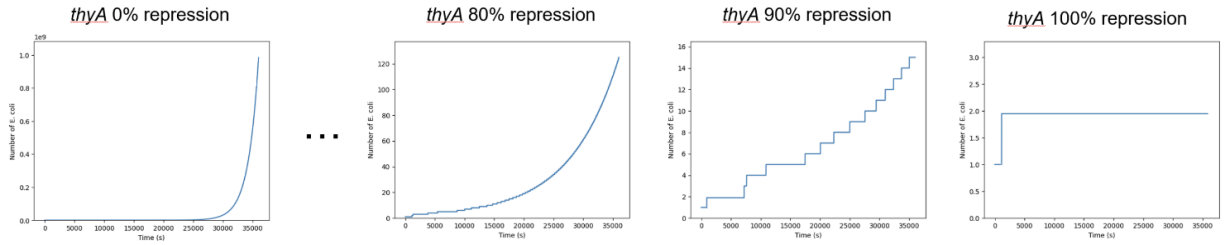


Figure 4. Perturbations can be gradually changed in L++. Here we show the gradual repression of ThyA to find the critical threshold where repression greatly alters the growth of the bacterial population.

With these methods you could run scores of trials of knockdown experiments to find a critical threshold in which your gene of interest loses most of its function. While these experiments would otherwise take days to experimentally alter the bacteria, culture them, then test for a knockdown. Using virtual *E. coli* would allow for the same experiments to be performed in mere minutes, this could involve perturbing another gene or it could involve perturbing other influences on the health of a cell including metabolites, glucose availability, and other molecular processes.

In-class exercises

Problem Set

- When programming a cell such as *E. coli* how can you combine two or more pathways within the cell and regulate them such that they do not interfere with each other?

E. coli properties

Consult the information below for additional information on how we can design virtual *E. coli* including concentrations of ribosomes, proteins, and other cellular components as well as structural features of *E. coli*.

Physical Properties		**Genetic Information**	
Cell Volume	0.3 ~ 3 μm	*Genome Size*	≈ 4.6 Mbp
Cell Dimension	2 μm × 1 μm × 1 μm (rod shape)	*Number of Protein Coding Genes*	≈ 4300
Total Weight	9.5×10^{-13} g	*Regulator Binding Site Length*	10 ~ 20 bp
Total Dry Weight (DW)	2.8×10^{-13} g (30% of total mass, where 70% of it being water)	*Promoter Length*	100 bp
		Gene Length	1000 bp

Chemical Composition		Molecular Dynamics	
Water	2×10^{10}	*Transcription Rate*	< 1 min (80 nts/s)
Inorganic Ions	1×10^{8} ≈ 4% of DW	*Translation Rate*	< 1 min (20 aa/s)
Lipids	5×10^{7} ≈ 10% of DW	*mRNA Lifetime*	≈ 3 min
DNA/Cell	4.6×10^{6} bp ≈ 3% of DW	*Protein Lifetime*	≈ 1 hr
RNA/Cell	2×10^{3} ≈ 20% of DW	*Minimal Doubling Time*	≈ 20 min
Proteins/Cell	3×10^{6} ≈ 55% of DW		
RNA Polymerases/Cell	2×10^{3}		
Ribosomes/Cell	1×10^{4}		

References

1. R.M. Otto, A. Turska-Nowak, P.M. Brown, K.A. Reynolds (2022). A continuous epistasis model for predicting growth rate given combinatorial variation in gene expression and environment. bioRxiv. https://www.biorxiv.org/content/10.1101/2022.08.19.504444v1
2. Milo, R., Phillips, R. (2015). Cell biology by the numbers. CRC Press.
3. Bennett, B., Kimball, E., Gao, M. et al. Absolute metabolite concentrations and implied enzyme active site occupancy in Escherichia coli. Nat Chem Biol 5, 593–599 (2009). https://doi.org/10.1038/nchembio.186
4. Bremer, H., Dennis, P. P. (1996) Modulation of chemical composition and other parameters of the cell by growth rate. Neidhardt, et al. Escherichia coli and Salmonella typhimurium: Cellular and Molecular Biology, 2nd ed. chapter 97, p.1559, Table 3 and note ’b’ beneath table
5. Frederick C. Neidhardt AND H. Edwin Umbarger, Chemical Composition of Escherichia coli, chapter 3 in Neidhardt F.C. Escherichia coli and Salmonella: Cellular and Molecular Biology. 2nd edition. Vol 1. American Society of Microbiology (ASM) Press 1996. p.2 table 1