

Randomness in AnyLogic models

The world we live in is non-deterministic. If today it took you 35 minutes to get to work, tomorrow it may be an hour, so when you set off in the morning you never know exactly how long it will take. You may know that on Friday evening on average 30 people come to your restaurant for dinner, but the time the first customer comes in tells you nothing about when you should expect the next one. John Smith who works for you as a salesman may have excellent skills, but when he is dealing with a particular prospect you never know for sure if he will be able to close the deal. When your company starts a new R&D project, you always hope it will bring you revenue in the end, but you always know it can fail. If you contact a person with flu you can get infected or you can successfully resist it. You are alive today, but nobody knows if you will be alive tomorrow.

Uncertainty is an essential part of reality and a simulation model has to address it to reflect this reality correctly. The only way of doing that is to *incorporate randomness into the model*, i.e. include the points that would give random results each time you pass them during the model execution.

This chapter describes possible ways to create sources of randomness in the model; namely the probability distributions, the random number generators and where and how you can use them in different kinds of models. The related topics of how to design experiments with a stochastic model and how to analyze the output of stochastic models are addressed in the sections on [experiment design](#) and [output analysis](#).

Probability distributions

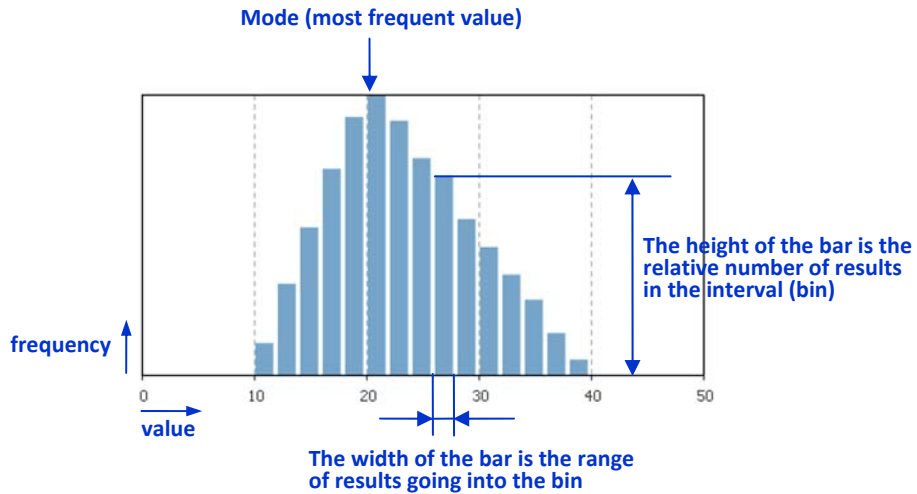
Suppose you are modeling a business process in a bank, in particular the operation of opening a new bank account. You know from your observations that it takes a minimum of 10 minutes, most likely 20 minutes, and a maximum 40 minutes, but you did not bother to make serious measurements. How do you model the delay time associated with this operation? AnyLogic offers you a set of [probability distribution functions](#) that will return random values distributed according to various laws each time you call them. For the purposes of this example we can use the function **triangular(min, mode, max)** with these parameters:

triangular(10, 20, 40)

If you call that function several times you will get a sequence of results like these:

Result	11.555	18.592	30.945	24.867	21.346	31.423	22.741	28.350
--------	--------	--------	--------	--------	--------	--------	--------	--------

and so on. As you can see, the results appear to be random and more or less consistent with your observations. To explore the function `triangular()` more thoroughly you can call it very many times and build a histogram of the results distribution. The histogram will look like the one in the Figure.

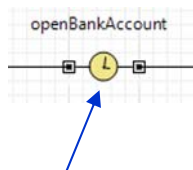


The distribution of 10,000 results returned by the function `triangular(10, 20, 40)`

The shape of the distribution (also called the *probability density function, PDF*) is a triangle with a minimum at 10, a maximum at 40 and a peak at 20 – the most frequent value also known as the *mode*. Indeed, the function `triangular(min, mode, max)` draws its results from the Triangular probability distribution with a given minimum, maximum, and most frequent values. We recommend using the triangular distribution function if you have limited sample data as in our bank example.

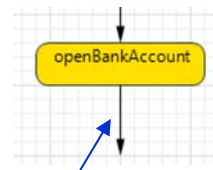
Depending on the type of your model you put a call to a probability distribution function, for example, into the **Delay time** parameter of a **Delay** or **Service** object, or into the **Timeout** field of a transition exiting the corresponding state, see the Figure. Each entity passing the object (or each agent coming to the state) will get a new sample of the distribution.

Process (discrete event) model



Delay time: `triangular(10, 20, 40)`

Agent based model

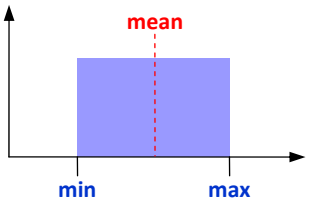
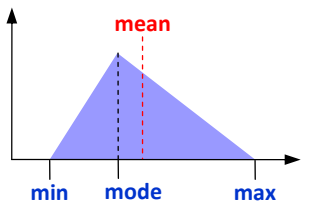
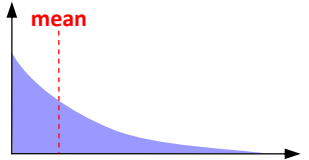


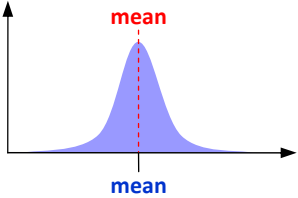
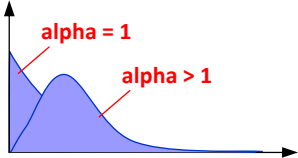
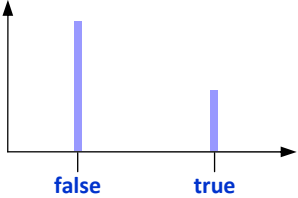
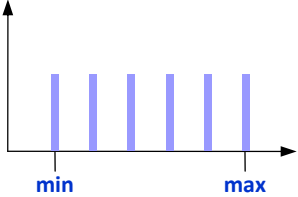
Timeout: `triangular(10, 20, 40)`

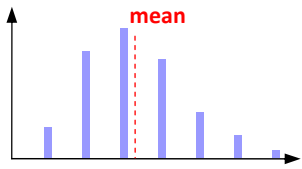
Using a probability distribution to model the duration of opening a bank account

Probability distribution functions

All randomness in AnyLogic models is based on calls to probability distribution functions. In total, AnyLogic supports about 25 distributions and offers over 50 corresponding functions. In the Table below we describe the most frequently used distributions. The complete information can be found in the [AnyLogic API reference](#).

Distribution name, PDF form, and AnyLogic functions*,**	Primary use
<p>Uniform</p>  <p> <code>uniform()</code> <code>uniform_pos()</code> <code>uniform(max)</code> <code>uniform(min, max)</code> </p>	<p>You know the minimum and the maximum values but have zero knowledge about how the values are distributed in between (i.e. you do not know if there are any values more frequent than others and assume a constant likelihood of a value being in any place between min and max).</p> <p>Used, for example, to generate coordinates of objects that are evenly spread over a rectangular area.</p>
<p>Triangular</p>  <p> <code>triangular(min, max)</code> <code>triangular(min, max, mode)</code> <code>triangular(min, max, left, mode, right)</code> </p>	<p>You know the minimum, the maximum, and have a guess about the most likely (modal) value.</p> <p>Used, for example, for service times, travel times, or, in general, for the duration of operations in conditions of limited sample data (too few samples to build a meaningful distribution shape).</p>
<p>Exponential</p>  <p> <code>exponential()</code> <code>exponential(lambda)</code> <code>exponential(lambda, min)</code> <code>exponential(min, max, shift, stretch)</code> </p>	<p>Describes the times between events in a Poisson process, i.e. when events occur independently at a constant average rate.</p> <p>Used as the inter-arrival time for input streams of customers, parts, calls, orders, transactions or failures in process models.</p> <p>In agent based models it is used as timeout for rate transitions that model independent events in agents that are known to occur at a certain global average rate.</p>

<p style="text-align: center;">Normal</p>  <p>A bell-shaped curve representing a normal distribution. The peak is labeled 'mean' in red, with a dashed vertical line extending down to the x-axis, which is also labeled 'mean' in blue.</p> <p> <code>normal()</code> <code>normal(sigma)</code> <code>normal(sigma, mean)</code> <code>normal(min, max, shift, stretch)</code> </p>	<p>Gives a good description of data that tend to cluster around the mean.</p> <p>For example, the height of an adult male person, the observation error in an experiment, etc.</p> <p>Note that the normal distribution is unbounded on both sides, so if you wish to impose limits (e.g. to avoid negative values) you have to either use its truncated form, or use other distributions such as Lognormal, Weibull, Gamma or Beta.</p>
<p style="text-align: center;">Gamma</p>  <p>Two curves representing gamma distributions. The first curve, labeled 'alpha = 1' in red, starts at the origin and decreases monotonically. The second curve, labeled 'alpha > 1' in red, starts at the origin, rises to a peak, and then decreases monotonically.</p> <p> <code>gamma(alpha, beta)</code> <code>gamma(alpha, beta, min)</code> <code>gamma(min, max, alpha, shift, stretch)</code> </p>	<p>A distribution bounded on the lower side. If alpha (the shape parameter) is 1 it reduces to the exponential distribution; for larger values of alpha it starts at 0, then has a peak and decreases monotonically.</p> <p>Used to model, for example, lifetimes, lead times or personal income data.</p>
<p style="text-align: center;">Random boolean</p>  <p>A bar chart representing a random boolean distribution. There are two bars: one at 'false' and one at 'true' on the x-axis. The 'false' bar is significantly taller than the 'true' bar, indicating a higher probability for the 'false' outcome.</p> <p> <code>randomTrue(p)</code> <code>randomFalse(p)</code> </p>	<p>Used to make a random decision between two alternatives with a given probability.</p> <p>For example, in process models used to divide the flow of entities into two, for example, economy and business class passengers or regular and urgent orders.</p> <p>In agent based models may be used in transition branches to model, for example, success or failure, and so on.</p>
<p style="text-align: center;">Discrete uniform</p>  <p>A bar chart representing a discrete uniform distribution. There are six bars of equal height, spaced evenly along the x-axis. The first bar is labeled 'min' and the last bar is labeled 'max' in blue.</p> <p> <code>uniform_discr(max)</code> <code>uniform_discr(min, max)</code> </p>	<p>Used to model a finite number of outcomes that are equally probable, or when you have no knowledge about which outcomes are more likely to occur.</p> <p>Example: a person (an agent) chooses a friend to communicate an idea.</p> <p>Note that both the minimum and maximum values are included in the set of possible results, so a call of <code>uniform_discr(3, 7)</code> may return 3, 4, 5, 6, or 7.</p>

<p style="text-align: center;">Poisson</p>  <p><code>poisson(lambda)</code> <code>poisson(min, max, mean, shift, stretch)</code></p>	<p>Discrete distribution describing the number of events occurring in a fixed period of time if these events occur independently and at a constant rate (lambda)</p> <p>Used to model, for example, the number of defects in a product or the number of calls in an hour, etc.</p>
<p>* There are many more probability distribution functions in AnyLogic. The full list with descriptions can be found in the AnyLogic Help section on Functions.</p> <p>** Note that each distribution function has also a form <code>name(..., Random r)</code> which enables you to use a custom random number generator.</p>	

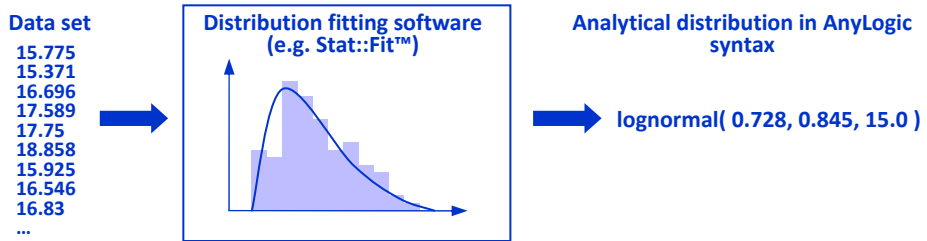
As you can see, in general there may be more than one function for a distribution: a short form that assumes default parameter values, and longer forms that allow you to tune the distribution for your particular problem. For example:

- `normal()` – the Normal distribution with mean at 0 and sigma (standard deviation) = 1
- `normal(sigma)` – mean at 0, custom standard deviation
- `normal(sigma, mean)` – both mean and standard deviation can be customized
- `normal(min, max, shift, stretch)` – same as above (**shift** is **mean**, **stretch** is **sigma**), but truncated to return values between **min** and **max**.

The latter form is provided for compatibility with Vensim™. *Truncation* of a distribution is performed in the following way: a draw from the original (not truncated) distribution is made, if the sample is outside the [min..max] interval, another try is made and so on until the sample is within the specified bounds.

Distribution fitting

If you are choosing the distribution in a state of limited information you can follow the advice in the previous section. However, if you have a data set (a set of observations) which well characterizes the random values in the system you are modeling, you can choose the right distribution by *fitting* it to the data set. *Distribution fitting* is the process of finding the analytical formula of a distribution that describes the random value as closely as possible to a given data set. There are various fitting heuristics and goodness-of-fit tests (e.g. the [Kolmogorov-Smirnov test](#)) and a number of software packages that would automatically perform distribution fitting and suggest one or several analytical distributions.



Input and output of distribution fitting software

Distribution fitting software would typically give you comprehensive statistics on your data set, display its histogram along with the PDF of the fitted distributions and rank the distributions according to several goodness-of-fit tests.

Should you be choosing distribution fitting software, it is recommended to use the one that directly supports the syntax of AnyLogic probability distribution functions, e.g. **Stat::Fit™**. The output of such software can be directly copied into an AnyLogic model.

Custom (empirical) distributions

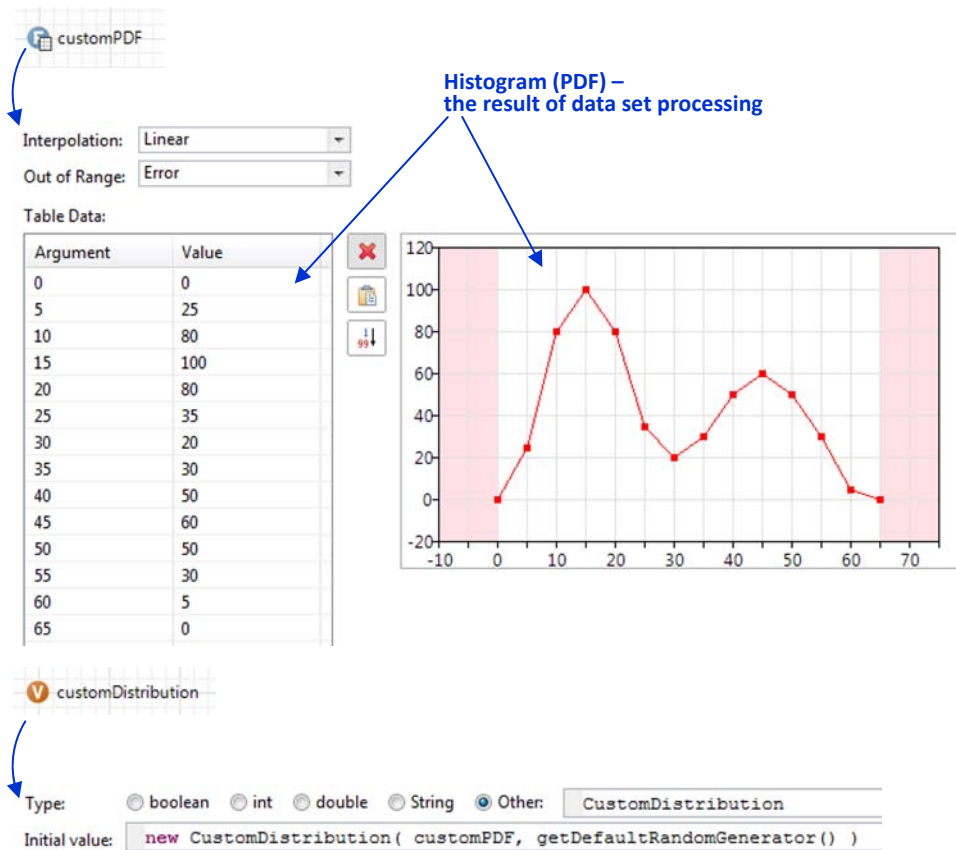
It may happen that no standard probability distribution can well fit the set of observations. In this case you can create a *custom* (also called *empirical*) distribution and use it in your model. As a first step you should build the PDF of your data set in the form of a histogram. This can be done using any statistical package, any [distribution fitting software](#) (which typically supports exporting in the form of an empirical distribution), or even an AnyLogic [Histogram data object](#). Then you should construct an instance of the **CustomDistribution** class giving it the PDF as a parameter, for example in the form of a table function. After that you can draw random values from your distribution by using the function **get()**.

► To create an empirical distribution from a data set:

1. Process your data set and obtain its histogram in the following form:

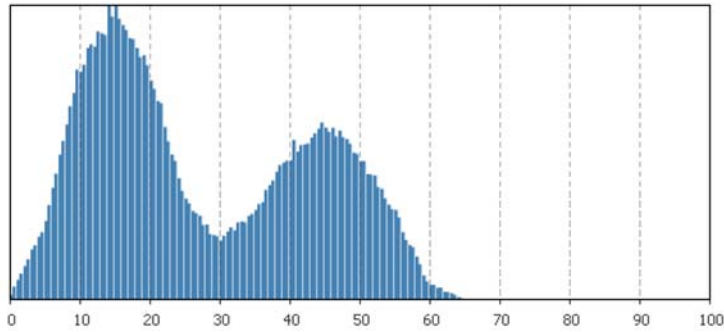
Interval bounds	Frequencies (number of samples in the interval)
Start of the first interval	Frequency of the first interval
Start of the second interval	Frequency of the second interval
...	...
Start of the last interval	Frequency of the last interval
End of the last interval	0

2. Open the **General** palette and drag the **Table** function object to the canvas. Set the name of the table function to **customPDF**.
3. Copy the histogram data in the text form to the clipboard.
4. In the **General** property page of the **customPDF** table function press the **Paste from Clipboard** button located between the table and the graph.
5. Check the graph shape. Leave the **Interpolation type** of the table function set to **Linear** (you can also use **Step** interpolation if you wish).
6. Drag the **Plain variable** object from the **General** palette to the canvas. Set its name to **customDistribution**.
7. In the **General** page of the **customDistribution** properties set:
Type: Other: CustomDistribution
Initial value: new CustomDistribution(customPDF, getDefaultRandomGenerator())
8. The custom distribution is ready to use. To draw a value call:
customDistribution.get()



Two objects used to construct a custom (empirical) distribution

A [random number generator](#) should be provided either in the constructor of the **CustomDistribution** (as in the example above), or as a parameter of the `get()` method. The custom distribution constructed from the table in the Figure will have a PDF as in the Figure below.



Histogram of 200,000 samples generated by the custom distribution

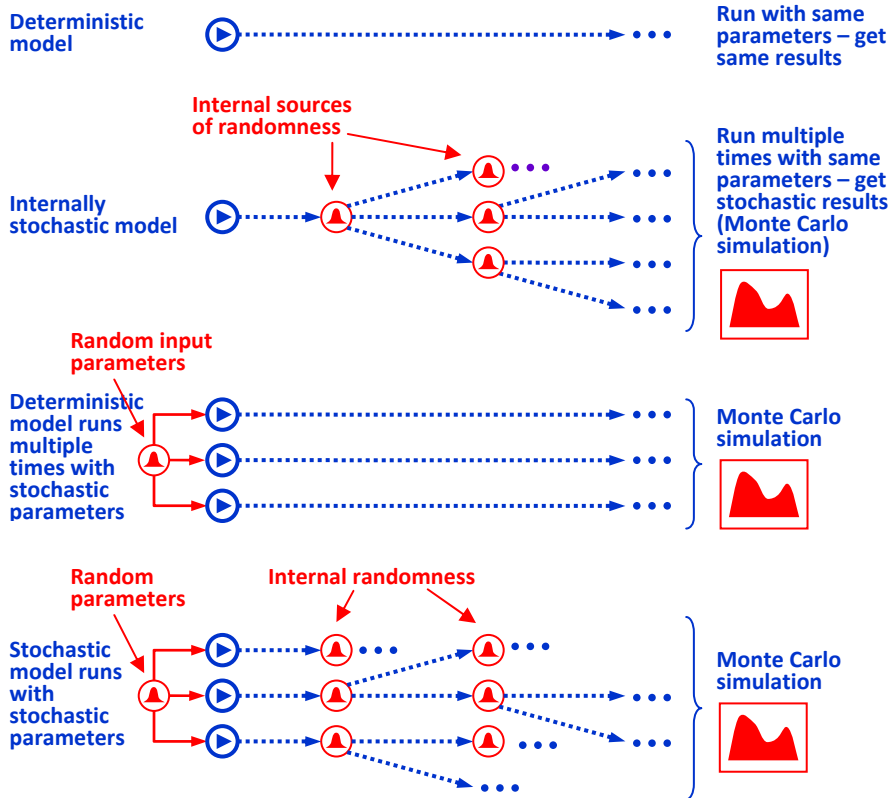
Sources of randomness in the model

There are stochastic and deterministic models. A *Deterministic model* has no internal randomness and, being run with the same set of input parameters, always drives the system through the same trajectory (the sequence of state changes) and gives the same output results. A *Stochastic model* has internal sources of randomness, and each run (even with the same parameters) may give a different trajectory and output.

In general there are more stochastic models than deterministic, especially among [process](#) (discrete event) models and [agent based](#) models. System dynamics models are mostly deterministic. This is explained by abstraction level: process and agent based models typically deal with individual objects, whose behavior has variations, while system dynamics deals with aggregates of large numbers of objects where individual variations are replaced (consumed) by averaging.

For a stochastic model you may need to perform multiple runs to get a meaningful picture of the system's behavior. The deterministic models are also often run multiple times with random variation of input parameters. A series of simulation runs with any kind of randomness (internal, at the level of input parameters, or both, see the Figure) is called *Monte Carlo simulation*. The results of Monte Carlo simulation are statistically processed and typically have the form of probabilities, histograms, scatter plots or envelope graphs, etc.

Alternatively to performing multiple runs you can explore a stochastic model in a single run if each point of randomness is passed many times in a loop and the model enters a stochastically stable mode.



Sources of randomness inside and outside the model. Monte Carlo simulation

In this section we focus on internal sources of randomness in different types of AnyLogic models. Stochastic variation of input parameters is described in the chapter on [experiment design](#) and analysis of stochastic models is covered in the chapter on [output analysis](#).

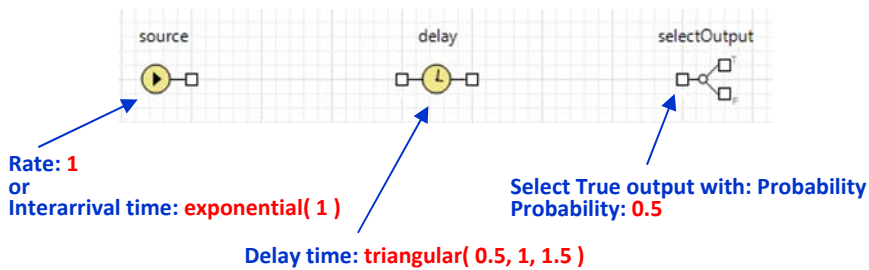
Randomness in process models

Randomness is an essential part of virtually any process model, be it a model of a manufacturing site, call center, warehouse, hospital, airport or a bank. The durations of operations, the arrivals of clients, orders or patients, the human decisions and errors, the equipment failures, the delivery times, etc. all vary randomly from one instance to another. Therefore the process flowcharts typically contain plenty of calls to probability distribution functions.

By default, the fundamental process flowchart objects of the Enterprise Library have built-in randomness. These are (see the Figure):

- **Source**: generates new entities with exponentially distributed inter-arrival time.
- **Delay** (and **Service** object based on it): has triangularly distributed delay time.
- **SelectOutput** (and its 5-exit version **SelectOutput5**): routes entities to different outputs with equal probability.

This means that, unless you explicitly eliminate the randomness from these objects, any process model you build is stochastic.

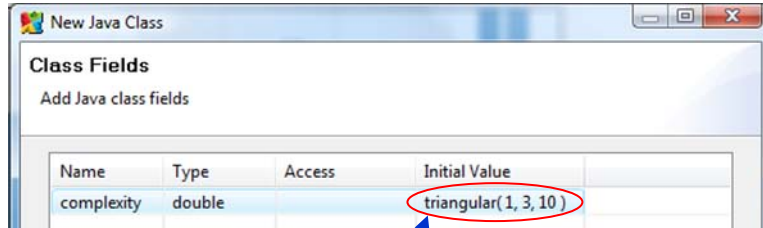


Randomness in the Enterprise Library objects

You may have noticed that if you run the simplest queuing model built of four objects **Source-Queue-Delay-Sink** with default parameters for a longer period of time, **Queue** overflow will occur. This is caused by the fact that mean values of entity inter-arrival time and delay time both equal 1; see the Figure. In these conditions the length of the queue has no limited mean.

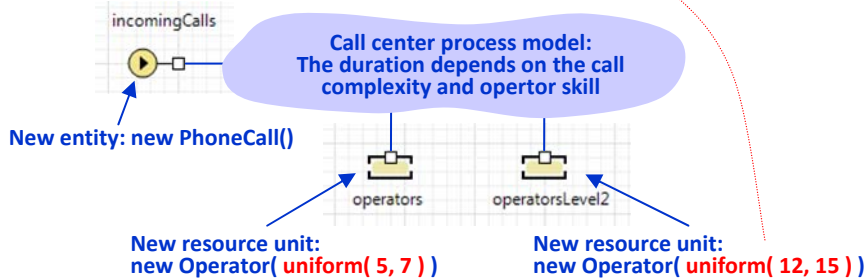
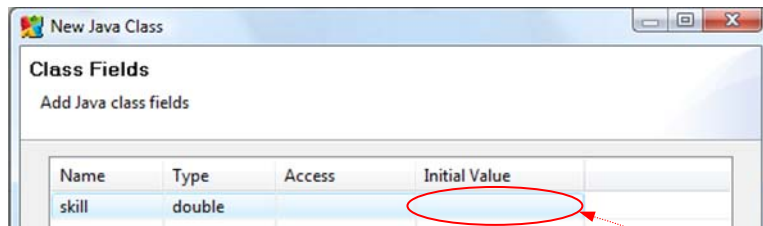
Other typical sources of randomness in process models are randomly distributed properties of entities and resource units. For example, in the model of a call center an entity class **PhoneCall** may have a field **complexity** with a randomly assigned value and the call center operators (resource units of class **Operator**) may have different random skills in different groups; see the Figure. Then the time it takes an operator to answer the call and the algorithm of call redirection may depend on the complexity and skill values.

Wizard (page 2): New Java class PhoneCall (base class ...Entity)



The complexity of the new PhoneCall instances will be distributed triangularly **INCORRECT!!!**

Wizard (page 2): New Java class Operator (base class ...ResourceUnit)



Random properties of entities and resource units in a call center model

Randomness in agent based models

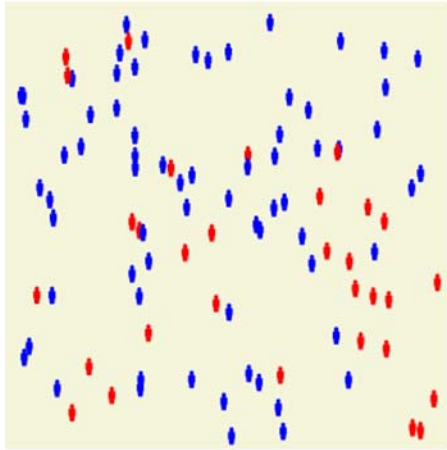
As well as process models, most agent based models are stochastic too. In particular, randomness may be present in:

- The initial locations of the agents (if [space](#) is used)
- The [network](#) of agent connections
- The properties of agents
- The agent behavior; in particular in agent communication

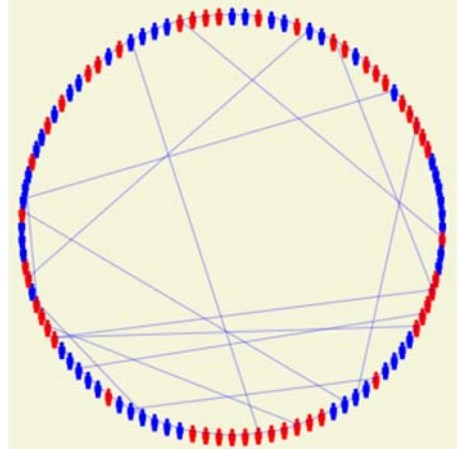
A random layout is frequently used to evenly distribute the agents over a rectangular area; this is done using uniform distribution. **Random, Small world** and **Scale free** networks are constructed using link probabilities. Probability distributions are often

used to set up the randomly distributed parameters over a population of agents, like the **income** parameter in the Figure. While communicating with other agents a random friend or any random agent may be chosen. The transitions in the agents' statecharts may fire at random times and have nondeterministic results.

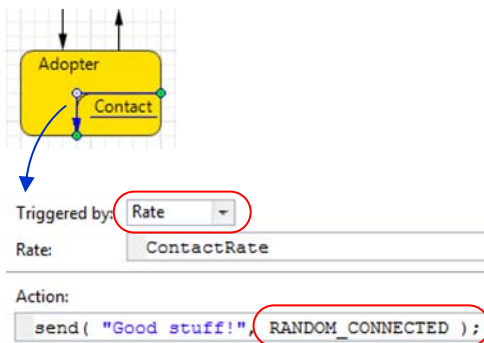
Layout type: Random
X and Y are uniformly distributed



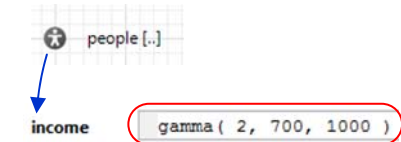
Network type: Small world
used neighborhood link probability



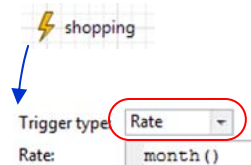
The transition fires at random times and sends a message to a randomly chosen friend



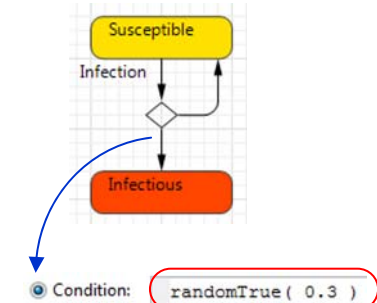
A collection of agents. The income of a person is drawn from a Gamma distribution



A particular kind of shopping event occurs randomly, on average once a month



In a contact between infectious and susceptible agents the disease is transmitted with a probability of 30%



Randomness in agent based models

Example: Agents randomly distributed within a freeform area

Suppose in an agent based model the agents need to be randomly distributed in an area bounded by a closed curve or a polyline (which may mean a city limit, for example). As the standard layouts only distribute agents over rectangular areas or rings only, you will need to create your own layout.

► **Follow these steps:**

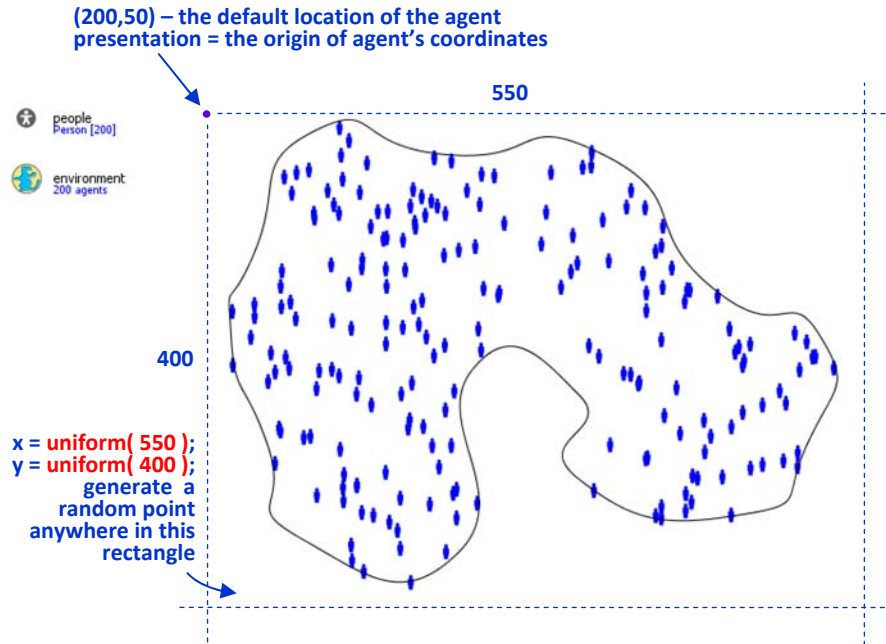
1. Press the **New** button on the toolbar. In the **New model** wizard enter the model name and on the next page choose the **Use template to create model** option and the **Agent Based** model template. Press **Next**.
2. On the next page of the wizard leave the default settings and press **Finish**. A new agent based model is created and the editor of its **Main** object opens.
3. Select the **environment** object and open the **Advanced** page of its properties. Set the **Layout type** to **User defined**.
4. Open the **Presentation** palette, double-click the **Curve** object and draw a curve like the one in the Figure by clicking at node points. Use double-click to finish drawing. Make sure the curve is fully on the right and fully below the presentation of the agent and that its width is less than 550, and its height is less than 400.
5. On the **General** page of the curve properties check the checkbox **Closed curve**.
6. Click the empty space of the editor to display the properties of the **Main** object.
7. On the **General** page write the following code in the **Startup code** field:

```
for( Person p : people ) { //for each agent
    double x, y;
    do {
        x = uniform( 550 );
        y = uniform( 400 );
    } while( ! curve.contains( x+200, y+50 ) ); // (200,50) is the origin of agents coords
    p.setXY( x, y ); //set the coordinates of the agent
}
```

8. Run the model. All agents should be randomly distributed across the area bounded by the curve.

In the startup code we are setting the initial location for each agent. We first generate a pair of random numbers **x** and **y** so that **x** is within 0..550 and **y** is within 0..400. Then we test if the point (**x,y**) is contained in the curve. (As the agent coordinate origin is where the agent presentation is located (the wizard places it at 200,50) and the **contains()** method accepts the global coordinates, we need to shift them by adding 200 and 50 correspondingly.) If the random point is not contained inside the curve,

another point is generated. Sooner or later the point will be within the curve bounds. Once this happens, we set up the agent coordinates and proceed to the next agent. The performance of this algorithm (i.e. the percent of successful tries) depends on the ratio of the area inside the curve to the enclosing rectangular area where we generate points. The same method would also work for shapes other than a curve.



Agents randomly distributed within an area bounded by a curve

❖ Example: Agents randomly distributed over a finite set of locations

Another similar task is to randomly distribute agents over a finite set of locations; for example seats in a movie theater or in an airplane. In this example we will use a discrete uniform probability distribution to generate a location. To define the set of locations we will use the points of a polyline.

► Follow these steps:

1. Press the **New** button on the toolbar. In the **New model** wizard enter the model name and on the next page choose **Use template to create model** option and the **Agent Based** model template. Press **Next**.
2. On the next page of the wizard leave the **Initial number of agents** set to **25** and press **Finish**. A new agent based model is created and the editor of its **Main** object opens.

3. Select the **environment** object and open the **Advanced** page of its properties. Set the **Layout type** to **User defined**.
4. Open the **Presentation** palette, double-click the **Polyline** object and draw a curve like the one in the Figure by clicking at node points. Create at least 30 points. Use double-click to finish drawing.
5. Click the empty space of the editor to display the properties of the **Main** object.
6. On the **General** page write the following code in the **Startup code** field:

```
int n = polyline.getNPoints(); //total number of locations = number of points
boolean[] occupied = new boolean[n]; //remember which are occupied
int freeplaces = n; //number of free locations
for( Person p : people ) { //for each agent
    int freeindex = uniform_discr( freeplaces-1 ); //random free location
    int ptindex = 0; //we will look for the index of the polyline point, start at 0
    while( true ) {
        if( ptindex >= n )
            error( "All points are occupied. Cannot find a place for the agent" );
        if( ! occupied[ptindex] ) //if location is free
            if( freeindex == 0 ) { //if this is the index we are looking for
                //place the agent there
                double x = polyline.getX() + polyline.getPointDx( ptindex ) - 200;
                double y = polyline.getY() + polyline.getPointDy( ptindex ) - 50;
                p.setXY( x, y );
                occupied[ ptindex ] = true; //mark location as occupied
                freeplaces--; //decrement the number of free locations
                break; //exit the while loop
            } else {
                freeindex--; //not yet at the desired free location - decrement index
            }
            ptindex++; //will look at the next point of the polyline
    }
}
```

7. Run the model. All agents should be randomly distributed over the points of the polyline.

The algorithm above is fairly long and differs from the one used in the previous example to find a point within the curve bounds. We could use a similar method; namely we could be directly generating the random index of the location, then test if it is free, and, if not, make another draw from the discrete uniform distribution. However, for a large number of locations densely occupied by agents it would be very inefficient: suppose you have already placed 9,990 football fans into a stadium with 10,000 seats. To place the next person you would need to make hundreds of tries

before the result of `uniform_discr(9999)` gives you one of the free seats left. In our algorithm we are directly choosing a random seat among the 10 free seats.

Randomness in system dynamics models

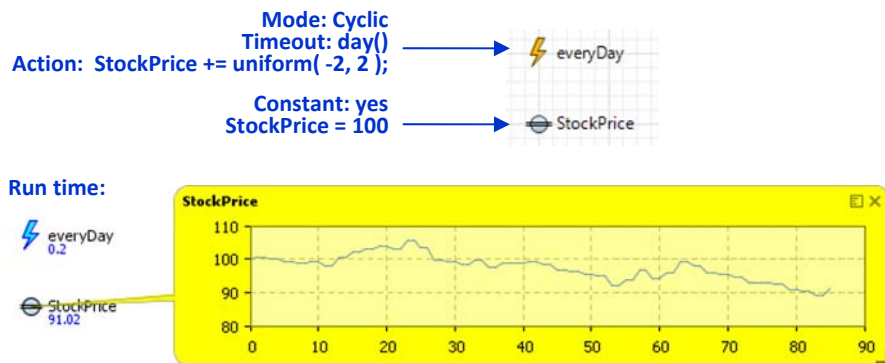
A system dynamics model built of standard elements, i.e. stocks, flows and feedback loops has no internal randomness (is deterministic) unless you explicitly insert it into the model. When doing that please follow the guidelines below:

In AnyLogic each new call to a probability distribution function generates a new value. Therefore you should not use those functions in the formulas of [system dynamics models](#): the numeric solver evaluates the formulas *several times in one time step* and will be confused if it gets different results.

To model values randomly changing in time in system dynamics models you should create a plain variable or SD constant and assign a random value to it at each time step (or less frequently, depending on the model logic) using, for example, a [cyclic event](#).

Example: Stock price fluctuations in a system dynamics model

Suppose in a system dynamics model you wish to have a variable for a stock price that randomly changes every day. A daily change is random and is not larger than 2 in either direction.



Random fluctuations of a stock price in a system dynamics model

► Follow these steps:

1. Open the editor of the active object. Open the **General** palette and drag the **Event** object to the canvas. Set the name of the event to `everyDay`.
2. In the **General** page of the event properties set **Mode** to **Cyclic** and set the **Recurrence time** to `day()`.

3. Open the **System Dynamics** palette and drag the **Flow Aux Variable** object to the canvas. Set the name of the variable to **StockPrice**.
4. In the **General** page of **StockPrice** check the **Constant** checkbox and set the value to **100**.
5. Open the **General** property page of the event **everyDay**. In the **Action** field write: **StockPrice += uniform(-2, 2);**
6. Run the model. Click the **StockPrice** and watch the changes in the **Inspect** window.
7. You can now use the random variable **StockPrice** in any formula in the system dynamics model.

By marking the **StockPrice** as **Constant** we just tell AnyLogic that the value specified in its properties (in our case **100**) *should be treated as an initial value only and should not be treated as a formula and evaluated by the numeric solver*. Instead we are explicitly assigning a new value to the **StockPrice** every day by the event **everyDay**. (As an alternative to a constant system dynamics variable we could also use a [plain variable](#).)

The uniform distribution was used as we do know the bounds of a daily change but do not know if any changes within those bounds are more likely than others.

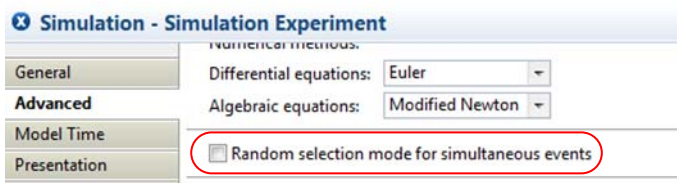
Randomness in AnyLogic simulation engine

Besides the sources of randomness at the model level as discussed above, there is only one internal source in AnyLogic simulation engine, namely the ordering of simultaneous events. This topic is extensively addressed in the [chapter on discrete events](#), here we would like to mention that:

- The engine uses the same [default random number generator](#) as the probability distribution functions do, and
- You can turn that randomness on and off:

► To set the ordering mode for simultaneous events:

1. Select the experiment and open the **Advanced** page of its properties.
2. Depending on what you want, check or uncheck the checkbox **Random selection mode for simultaneous events**.



You can turn random ordering of simultaneous events on and off

Random number generators. Reproducible and unique experiments

The computer (or, at least, the processor executing a program) is a completely deterministic device: its next state is fully determined by the current state. So, when we talk about randomness in simulation models, did you ever wonder where that randomness comes from?

The truth is that, unless a software application accesses an external physical random number generator, there is no real randomness in it; however a computational random number generation may be used to create pseudo-randomness.

Random number generators

A *random number generator* (RNG) is a device that generates a sequence of numbers that lack any pattern, i.e. appear random [wikipedia]. There are two types of RNGs: physical and computational. *Physical RNGs* have been known from ancient times: dice, coin flipping, shuffling of playing cards, roulette wheel and other techniques. The modern ones use atomic and subatomic physical phenomena, such as radioactive decay or thermal noise, or (like random.org) atmospheric noise, or radio noise. Physical RNGs generate “true” random numbers, i.e. those that are completely unpredictable.

Computational RNGs are based on deterministic algorithms that produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a *seed*, and are periodic. They are therefore called *pseudo-random number generators*. Pseudo-random number generators can be used instead of “true” ones in many applications; in particular, in the majority of simulation models. Moreover, their predictability feature is used to create [reproducible stochastic experiments](#).

By default, all probability distribution functions in AnyLogic, the Enterprise Library objects, the random transitions and events, the random layouts and networks and the AnyLogic simulation engine itself – in other words, all randomness in AnyLogic, is based on the *default random number generator*. The default random number generator is an instance of the Java class `Random`, which is a [Linear Congruential Generator](#) (LCG). The LCG is one of the oldest and best known pseudo-random generators. It is very simple: the stream of random numbers is determined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \bmod m$$

where a is a multiple, c is increment, and m is modulus. The period of LCG is at most m , and in the class **Random** $m = 2^{48}$. The initial value X_0 is a seed.

If for any reason you are not satisfied with the quality of **Random**, you can:

- Substitute your own RNG instead of the AnyLogic default RNG.
- Have multiple RNGs and explicitly specify which RNG should be used when calling a probability distribution function.

► **To substitute the default RNG with your own RNG:**

1. Prepare your custom RNG. It should be a subclass of the Java class **Random**, e.g. **MyRandom**.
2. Select the experiment and open the **General** page of its properties.
3. Select the radio button **Custom generator (subclass of Random)** and in the field on the right write the expression returning an instance of your RNG, for example:
`new MyRandom()` or `new MyRandom(1234)`

Random number generation:

<input type="radio"/> Random seed (unique simulation runs)	
<input type="radio"/> Fixed seed (reproducible simulation runs)	Seed value: <input type="text" value="1"/>
<input checked="" type="radio"/> Custom generator (instance of java.util.Random class):	<input type="text" value="new MyRandom(1234)"/>

Setting a custom random number generator as default RNG

The initialization of the default RNG (provided by AnyLogic or by you) occurs during the initialization of the experiment and then before each simulation run.

In addition you can substitute the default RNG at any time by calling:

`setDefaultRandomGenerator(Random r)`

However you should be aware that before each simulation run the generator will be set up again according to the settings on the **General** page of the experiment properties.

► **To use a custom RNG in a particular call of a probability distribution function:**

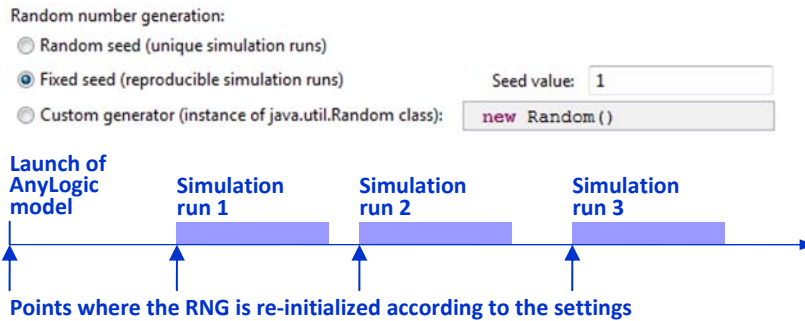
1. Create and initialize an instance of your custom RNG. For example, it may be a plain variable **myRNG** of class **Random** or its subclass.
2. When calling a probability distribution function, provide **myRNG** as the last parameter, for example:
`uniform(myRNG)` or `triangular(5, 10, 25, myRNG)`

If a probability distribution function has several forms with different parameters, some of them may not have a variant with a custom RNG, but the one with the most complete parameter set always has it.

The seed. Reproducible and unique experiments

Although pseudo-random number generators do not produce “truly random” streams of numbers, they have one feature which is very important in simulation modeling: having been initialized with a particular seed they generate exactly the same sequence of numbers each time. This enables you to create *reproducible experiments* with stochastic models, which would be impossible with a “true” RNG. Reproducibility, i.e. the ability to run the model along the same trajectory of state changes, is useful when you are debugging the model, or when you wish to demonstrate a particular scenario.

In AnyLogic you have two options for the standard RNG, see the Figure: you can choose **Random seed** to run unique experiments, or **Fixed seed** to run reproducible experiments. The seed is set during the initialization of the experiment and then at the beginning of each simulation run (replication). If a custom RNG is provided, AnyLogic at those points just sets the default RNG to what is specified in the corresponding field.



AnyLogic RNG seed settings and the points when they are applied

A valid question is: where is the “random seed” taken from? When the random seed option is chosen, AnyLogic calls the default constructor of Java class `Random()`, which sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor. In the earlier versions of Java the computer system time was used as a seed in that case.

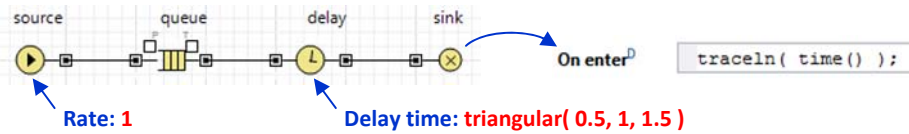
Example: Reproducible experiment with a stochastic process model

We will create the simplest queuing system: a source of entities, a queue, a delay with capacity 1, and a sink object. The model will have two sources of randomness: the arrival times of the entities and the delay time: under the default settings the inter-

arrival times are distributed exponentially and the delay times triangularly. (See [Randomness in process models](#).) We will record the entity exit times and compare them for different runs.

► **Create and run the model with default seed settings (random seed)**

1. Press the **New** (model) button on the toolbar. In the **New model** wizard enter the model name and on the next page choose the **Use template to create model** option and the **Discrete Event** model template. Press **Next** and press **Finish** on the next page. A new process model is created and the editor of its **Main** object opens.
2. Select the **sink** object. In the **On enter** field of its properties write: `traceln(time());` – each time the entity exits the model we will write the current time to the model log.
3. In the **Projects** tree select the **Simulation** experiment. On the **Model time** page of the experiment properties set:
Stop: Stop at specified time
Stop time: 20
4. Run the model up to completion. Look at the model log in the **Console** window of the AnyLogic model development environment: there should be about 20 records.
5. Press the **Stop** button on the model toolbar and run the model again. Another portion of records should appear in the log. The entity exit times are different from the ones in the first run.
6. Close the model window and run the model again. The output should again be different.



Execution results (random seed):

Execution results (fixed seed = 1):

Run 1:

```
anylogic config [Java Applica
3.52141717979589
4.849374099635038
6.096712763798887
7.362107075477317
8.144411189871564
9.140158222025375
10.164109447836058
10.743006040647245
11.978769537476559
13.04559083469592
13.806518948794587
14.54890652717644
15.500840137353903
16.51542401274163
18.02386664482178
19.410723117937547
```

Run 2:

```
anylogic config [Java Applica
1.5604550761627234
3.8290354125946218
4.901375301972588
5.883336640924297
6.752258481266244
7.64847240471175
8.940080257525333
9.745630479947621
10.808863792649475
11.751260985276854
12.936698075529803
14.228490703882237
15.49657309735129
16.273563278376308
17.452507251908955
18.060949945057217
19.331362549234818
```

Run 1:

```
anylogic config [Java Applica
1.1357776411258893
2.5779368881156968
3.9432238542682025
8.739537451159041
9.622980427365437
10.363778151641082
11.451402539300476
12.38626001326238
13.49571422823999
14.683646744731785
15.695351319701166
16.8375696028942
```

Run 2:

```
<terminated> anylogic confi
1.1357776411258893
2.5779368881156968
3.9432238542682025
8.739537451159041
9.622980427365437
10.363778151641082
11.451402539300476
12.38626001326238
13.49571422823999
14.683646744731785
15.695351319701166
16.8375696028942
```

Identical

Unique and reproducible experiments with a stochastic process model

You see that under the default settings of AnyLogic RNG each run of the stochastic model is unique. Now let us change the RNG settings.

► Change the RNG settings to run reproducible experiments

7. In the **Projects** tree select the **Simulation** experiment. On the **General** page of its properties choose **Fixed seed (reproducible simulation runs)** and leave the default seed value.
8. Run the model several times and compare the outputs. They should be exactly the same.

The important thing is that the results will be the same every time and everywhere: you may export your model, send it to a client, or publish it as a Java applet on the web – and anybody who runs it will observe exactly the same behavior.