Documentation for the BioMotion Lab (BML) Experiment Toolkit

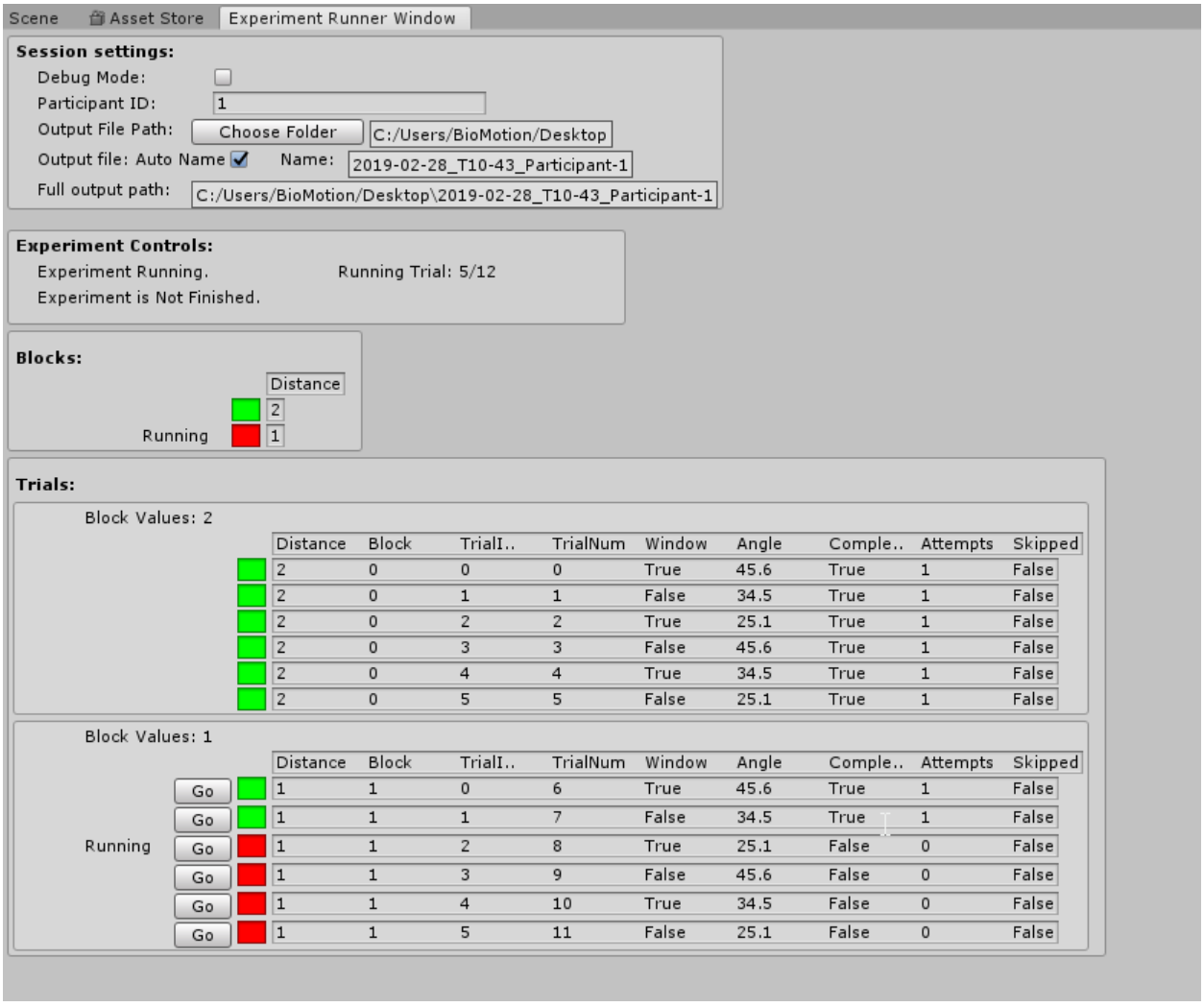
Author: Adam Bebko

Contents

About

The BML Experiment Toolkit helps you design and run experiments in unity quickly and iteratively without fussing over coding details. You define your variables and experiment structure, and the toolkit will automatically create a table of trials to run. You can customize what happens before, during, and after a trial, block of trials, or the experiment itself, while not having to worry about the details of setting up your experiment. The toolkit will go through all the trials, outputting the results automatically to a file.

The toolkit adds a menu called BML, which contains several useful items for running your experiments.



Overview

The main structure of an experiment is defined with a variable configuration file. In this file you define your independent and dependent variables, and any blocking or randomization that needs to occur.

Once the variable structure has been defined, you can write scripts inheriting from the toolkit's base classes, customizing functions that are called automatically during the experiment. For each "stage" of the experiment (Experiment, Block, Trial), you can define custom behavior that interacts with your Unity scene. Examples of such customization might include a "welcome screen" before the experiment, a "thank you" screen after the experiment, a pause between each block of trials, and instructions before each trial. These customizations can be written very quickly.

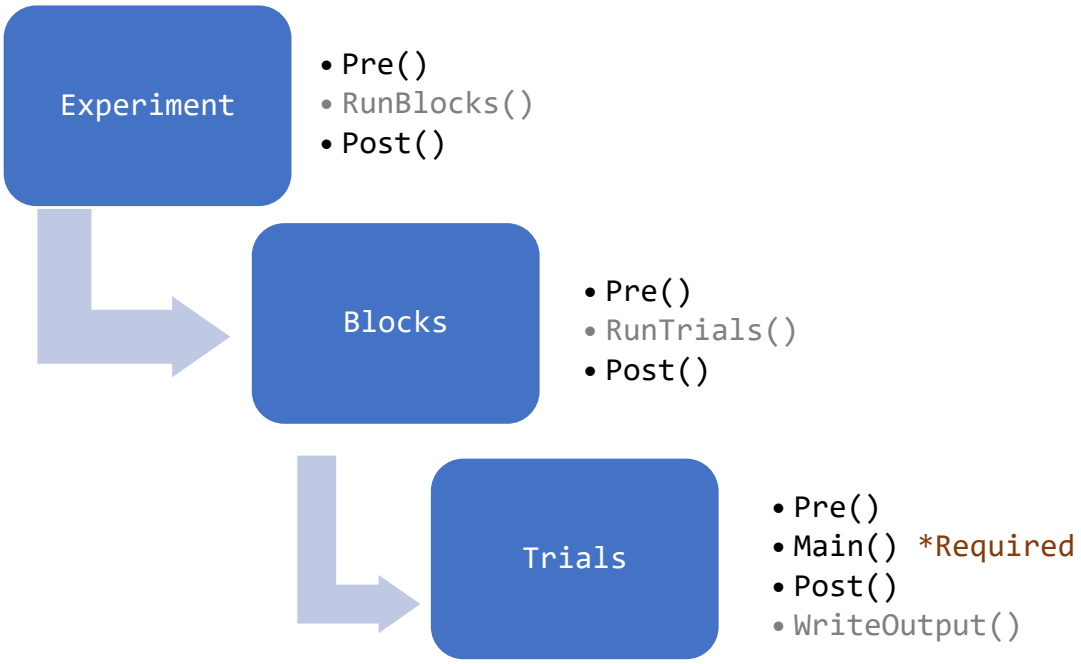


Figure 1: Basic structure of an automatically generated experiment. You can customize functionality for all functions except grey ones. You must define the Main() function for your trials to define what should happen in each trial.

The main requirement for setting up an experiment is defining what occurs during a trial. A trial script has access to each of the variables you defined in the config file, so you can set up objects in your scene using values from the independent variable, or write results to the dependent variables.

After hitting play in unity, a new experimental session is created, and a window appears prompting you for session details such as participant ID, and which block order to use.

The trial structure is displayed to double-check correctness, and then you can begin the experiment. The experiment runs, automatically running the custom code you defined above at the appropriate time.

Getting started

Please check out the tutorials, which should give you a handle on how to get up and running using the toolkit to run your experiments.

[not yet written]

Configuring your experimental design

With the toolkit, you need only define your variables, and your experimental design will be created automatically for you.

To set up your experimental design you need to define your variables. Create a new Config file from the BML menu. It will appear in your project's Assets folder.

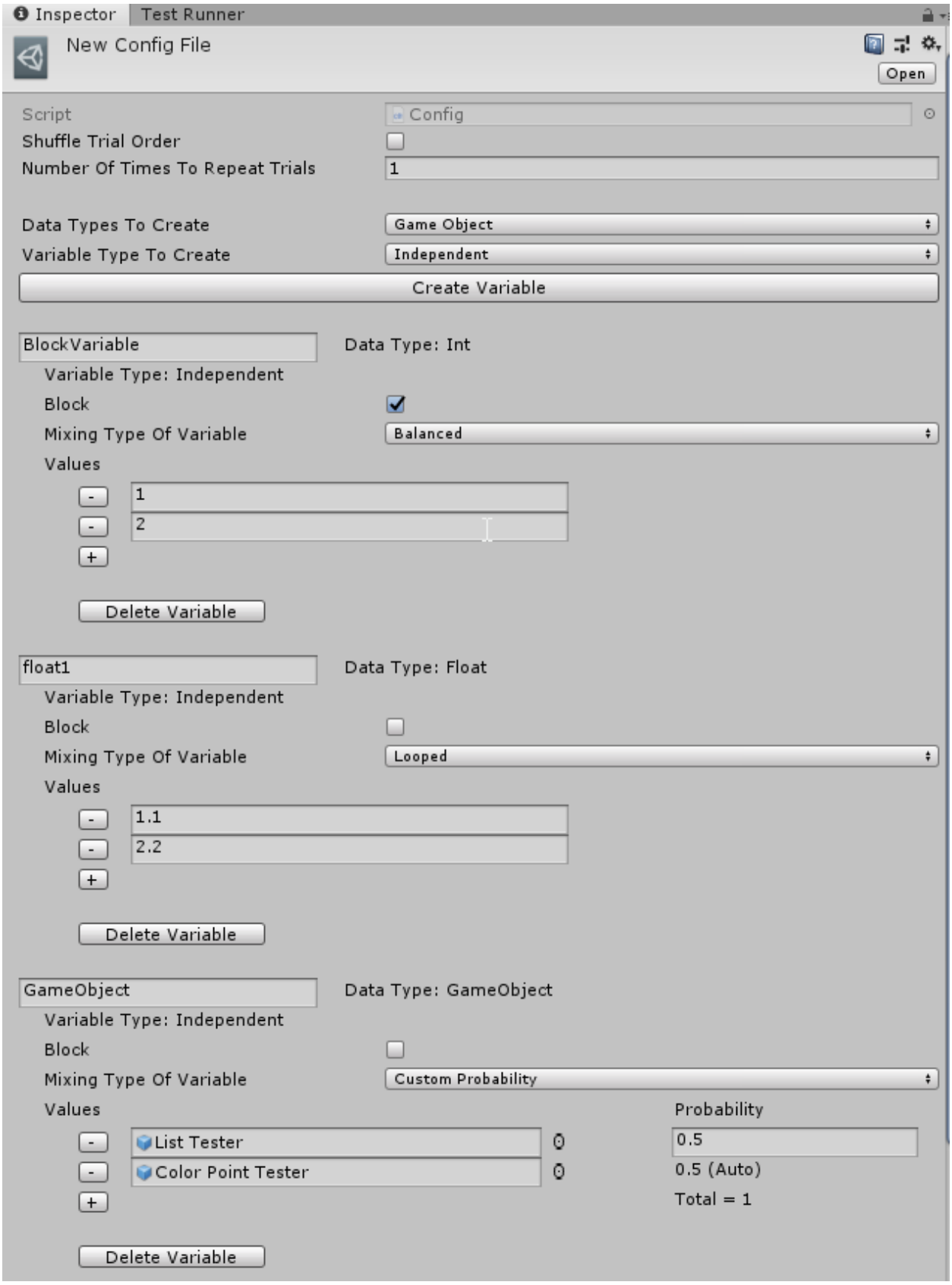

Inspector
Test Runner
New Config File
Open
Script
Config
Shuffle Trial Order
Number Of Times To Repeat Trials
1
Data Types To Create
Game Object
Variable Type To Create
Independent
Create Variable
BlockVariable
Data Type: Int
Variable Type: Independent
Block
Mixing Type Of Variable
Balanced
Values
1
2
Delete Variable
float1
Data Type: Float
Variable Type: Independent
Block
Mixing Type Of Variable
Looped
Values
1.1
2.2
Delete Variable
GameObject
Data Type: GameObject
Variable Type: Independent
Block
Mixing Type Of Variable
Custom Probability
Values
Probability
List Tester
0.5
Color Point Tester
0.5 (Auto)
Total = 1
Delete Variable

Figure 2: A screenshot of the variable creation inspector.

To add a variable, select your desired data type (int, string, etc.), type of variable (independent, dependent), and then click on Create Variable. You should see your variable added to the list below. Configure your variable by naming it, adding values, selecting whether the variable will be used to create blocks of trials, etc.

To delete a variable from the list click the Delete Variable button

Your changes will be saved automatically.

When complete, drag your config file into your Custom Experiment GameObject Prefab, and it will run.

The power of the configuration files is that you can run the same experiment with multiple saved configurations. For example, you can select different values for your experiment in different config files and drag your desired configuration into the experiment object prior to running participants. This also allows you to iteratively design your experiments while saving previous configurations.

Variable options

Name

You must name your variables. This will be how you access their values within your trials. I recommend using one-word names, or joining words using underscores. Do not use any special characters or punctuation.

Datatype

Once created, you cannot modify this variable's datatype. This lets Unity know what kind of data to expect. To get a value from a variable in a trial you must know its datatype to avoid errors. Currently, supported datatypes include:

- Int
- Float
- String
- GameObject
- Vector3

Variable Type

Independent and dependent variables are treated differently. Independent variables define your experimental structure, and dependent variables are values that determined from responses or measurements during each trial

Independent variable options:

Block

Enabling this option will create blocks of trials for each value defined. Useful for counterbalancing or variables that require setup in real life (i.e. participants need to move between values)

Mixing Type of Variable

This option will define how the variable is mixed with the other variables when creating your experimental design.

Balanced:

This will create trials for every combination of values of each balanced variable. Example: for two 3-value balanced variables, there will be 3x3=9 trials.

Trial Number	Balanced Variable 1	Balanced Variable 2
1	1	1
2	1	2
3	1	3
4	2	1
5	2	2
6	2	3
7	3	1
8	3	2
9	3	3

Looped:

This will loop through the values such that there is an equal number of trials with each value. Lowest common multiple.

Eample1: For a variable with 2 values, and another variable with 4 values, the following table will be created. In this case the lowest common multiple is 4 trials.

Trial Number	Looped Variable 1	Looped Variable 2
1	1	1
2	2	2
3	1	3
4	2	4

Example2: For a variable with 2 values, and another variable with 3 values the following table. In this case the lowest common multiple is 6 trials.

Trial Number	Looped Variable 1	Looped Variable 2

1	1	1
2	2	2
3	1	3
4	2	1
5	1	2
6	2	3

Even Probability:

Each trial will have a randomly selected value for this variable, with the same probability for each value.

Example, A balanced variable with 6 levels, and an even probability variable with 10 values (numbers 1-10)

Trial Number	Balanced Variable 1	Even Probability Variable 2
1	1	4
2	2	8
3	3	9
4	4	4
5	5	7
6	6	2

Custom Probability:

Each trial will have a randomly selected value for this variable, with a defined probability of being selected. You define the probability to the right of each value (as a decimal). The final probability is automatically calculated to ensure they add up to 1.

Example, A balanced variable with 6 levels, and a custom probability variable with 2 values with the first value having 0.2 probability, and the second value having 0.8 probability

Trial Number	Balanced Variable 1	Custom Probability Variable 2
1	1	2
2	2	2
3	3	2
4	4	1
5	5	2
6	6	2

Values

Define the levels of each variable. You must define at least one value.

Probability

Only visible when custom probability mixing type is selected. This is where you define your probability of each value being picked.

Dependent variable options:

Default value

This is the value given to your dependent variables in case there is no response given, or the variable is not updated in your trial. This can be left blank if desired.

Controlling an experiment

The Experiment Runner Window

Once the experiment's variables have been configured, the experiment can be run from the Experiment Runner Window. You can open this window from the BML menu. This window can be docked and moved around just like any other editor window.

Once the window is open, there will be an error explaining that it can only display when Unity is in play mode.

Press play in the unity editor. The unity scene will begin, but the experiment will not yet run. It will prompt you for settings relevant to the session, including picking which block order to use.

When all required settings have been selected, the Experiment controls will show, allowing you to begin the experiment. The experiment will show the trial structure of your experiment and track your progress through the experiment.

In-Trial Experimenter Controls

During the experiment you can jump between trials by pressing the "go" button next to the trial in the Experiment Runner Window. Any incomplete trials will be revisited at the end of the block. It is not currently possible to jump between blocks. This might be useful if your participant says they made an error. Additional attempts on a trial will be recorded in the output in the "Attempts" column.

The experimenter can also skip between trials using the keyboard using the following controls

- Left, Up Arrow keys: Go back one trial (current trial will be revisited at end of block)
- Right, Down Arrow keys: Go to next trial (current trial will be revisited at end of block)
- Delete: Skip trial completely (will NOT be revisited at end of block)

Structure of an experiment

Session

The session class defines one time through an experiment. It stores options that affect the whole experiment. You can set options such as debug mode for testing out your experiment, the participant's ID, block order number and the output file.

You can change your session settings when running your experiment program. By default, the session settings will display your previously used settings.

Debug mode will save your output into a debug file created in your project's Assets/Debug/debugFile. It can also be used to change functionality in your trials for testing purposes.

You can manually name your output files, or let the toolkit do it for you based on the current date, time, and participant ID.

This is where you choose your desired block order or have it randomly select the block order for you [Not yet implemented].

Experiment

The Experiment class is your main window into running experiments with the toolkit. To Define an experiment, you need to create a new script and have it inherit from the Experiment class.

```
1.  public class MyExperiment : Experiment {
2.
3.  }
```

The Experiment class provides the main functionality for running experiments in the toolkit. You need to attach your custom experiment script to a GameObject in your scene. I suggest an empty GameObject Called ExperimentManager or something similar.

The above code attached to a GameObject is all you need to do to have a functional experiment. However, it is also easy to add custom functionality.

Within your custom Experiment class, you can define custom behavior to occur pre- and post-experiment. To do so, you write the following

```
1.  public class MyExperiment : Experiment {
2.      protected override IEnumerator Post() {
3.          //Your code here
4.          return null;
5.      }
6.
7.      protected override IEnumerator Pre() {
8.          //Your code here
9.          return null;
10.     }
11. }
```

Lets try to understand this a bit.

A Note on Coroutines and IEnumerators

Normal loops in normal code keep looping until done, and don't allow for you to extend code across multiple frames in a unity program.

For example, the following code will hang your program since it never lets the program get to the next frame.

```
1.  void NormalKindOfMethod() {
2.      while (true) {
3.          //do something
4.      }
5.  }
```

What Coroutines and IEnumerators let you do is have a loop, but then pause at certain points to let the rest of the program run. Then, during the next frame of your program, it will pick up where it left off. Therefore, your program will run normally, but the loop is updated each frame.

```
1.  IEnumerator CoroutineMethod() {
2.      while (true) {
3.          //do something
4.
5.          yield return null; //continue below this after next frame
6.
7.          Debug.Log("this will print to console the next frame");
8.      }
9.  }
```

To begin a coroutine method, you cannot call it like a normal method. You need to use a special unity function called StartCoroutine(). This functionality is already written for you in the toolkit. For a Coroutine function, you need only tell Unity where to pause to wait for the next frame using a "yield return" statement. You can call normal methods inside coroutine functions, and everything else behaves as you would expect it to, with the added power of being able to have behavior that occurs over many frames.

In addition to waiting for a frame, you can have unity wait for a specified amount of time before continuing. This is great for displaying instructions. Or for creating time delays in your code. You can have unity display some instructions to a participant, wait for a few seconds, then stop displaying the instructions. Below is an example of displaying instructions at the start of your program for 5 seconds.

```
1.  protected override IEnumerator Pre() {
2.
3.      DisplayInstructions(); //called right away
4.
5.      //the rest of your program will run normally while it waits.
6.      yield return new WaitForSeconds(5);
7.
8.      //will only get called after 5 seconds
```

```
9.      StopDisplayingInstructions();
10. }
11.
12.
13. void DisplayInstructions() {
14.     //your code for displaying
15. }
16.
17. void StopDisplayingInstructions() {
18.     //your code for stopping to display
19. }
```

As a rule of thumb, it's a good idea to follow the following structure:

- In your Pre() method:
  - Show instructions, welcome screen.
  - Initialize your experiment (calibration, etc.)
- Experiments do not have custom access to the Main() method.
- In your Post() method:
  - Thank you screen.
  - Finalize your experiment (confirm calibration still valid, etc.)

Blocks

If you flag any of your independent variables as blocking variables, the toolkit will automatically create blocks for you, and run them.

However, If you'd like to customize what happens when a block starts or ends, you can use similar functionality to your custom experiment class.

```
1.  public class MyBlock : Block {
2.
3.  }
```

However, this will give you an error. In most editors you can automatically solve this by right-clicking on MyBlock and selecting "implement missing members". If you can't do that you can type the following in manually

```
1.  public class MyBlock : Block {
2.
3.      public MyBlock(DataTable trialTable,
4.                     string identity,
5.                     Type CustomTrialType)
6.          : base(trialTable, identity, CustomTrialType) { }
7.  }
```

This is a Constructor that creates Blocks. All it does is forward that job up to the main Block class (base), so you don't have to worry about it. Now you can override Pre() and Post() as above to implement behaviour before and after a block runs.

For example, in one of my experiments, one of my blocking variables was where the participants stood. So, before each block I gave them instructions to move to a location, and only allowed the program to continue if they were standing in the correct location.

Remember that the code defined in your custom Block class is general for all blocks, but the behaviour changes based on the values of your variables. Therefore, you only need to code the behaviour for all blocks in one place and set up each one based on the values of your variables for that block.

Accessing your variables is easy. Their values are stored in a field of the Block class called "data". This "data" object will be updated and set to the correct values for that block automatically by the toolkit. Note that only independent variables flagged for blocking will be accessible, and an error will be given otherwise.

Let's say you defined an integer-type independent block variable named "MyIntBlockVariable" with values 1 and 2. To access it from a Block, write the following:

```
1.  int blockVariable = (int) data["MyIntBlockVariable"]
```

This stores each Block's value for that variable into an int called blockVariable which can be manipulated normally like any other c# variable. You need the (int) at the start to remind c# that your variable is type int.

Now you can modify things for each block based on the value. For example, you could move an object based on its value:

```
1.  Vector3 positionToMoveTo = new Vector3(blockVariable, 0, 0);
2.  someGameObject.transform.localPosition = postionToMoveTo;
```

now, each block will move someGameObject to the correct position for that trial.

See the section on Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts For more information on how to customize your Blocks.

As a rule of thumb, it's a good idea to follow the following structure:

- In your Pre() method:
    - Set up your environment for each block.
    - Show instructions.
- Blocks do not have custom access to the Main() method.
- In your Post() method:
    - Reset everything in preparation for next Block.

Trial

This is similar to defining blocks with Pre() and Post(), except now we can also override the Main() function, which is your custom behaviour of what occurs during a trial. Also, your trial class will have access to all the variables of your program.

Remember that the code defined in your custom Trial class is general for all trials, but the behaviour changes based on the values of your variables. Therefore, you only need to code the behaviour for all trials in one place and set up each trial based on the values of your variables for that trial.

Accessing your variables is easy. Their values are stored in a field of the Trial class called “data”. This “data” object will be updated and set to the correct values for that trial automatically by the toolkit.

Let’s say you defined an integer-type independent variable named “MyFirstVariable” with values 1 and 2. To access it from a trial, write the following:

```
1.  1.  int myFirstVariable = (int) data[“MyFirstVariable”]
```

This stores each trial’s value into a new int variable called myFirstVariable which can be manipulated normally like any other variable. You need the (int) at the start to remind c# that your variable is type int.

Now you can modify things for each trial based on the value. For example, you could move an object based on its value:

```
1.  1.  Vector3 positionToMoveTo = new Vector3(myFirstVariable, 0, 0);
2.  2.  someGameObject.transform.localPosition = postionToMoveTo;
```

Now, each trial will move someGameObject to the correct position for that trial.

To write responses or results to your dependent variables, write the following

```
1.  1.  float response = 5.6f;
2.  2.  data[“MyFloatDependentVariable”] = response;
```

Note: Any updated values for the dependent variables will be automatically added to the output CSV file.

See the section on Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts for more information about customizing your trials.

As a rule of thumb, it’s a good idea to follow the following structure:

- In your custom Pre() method:
    - Set up your environment for each trial.
    - Set up stimuli, other variables.

    - Access your independent variables to set up trial.
    - Show instructions.
- In your Main() method:
    - Present stimuli.
    - Collect responses.
    - Take measurements.
- In your custom Post() method:
    - Finalize any measurements.
    - Write data to your dependent variables.
    - Reset everything in preparation for next trial.
- WriteOutput() method:
    - This cannot be customized directly, but is called automatically after your Post() method completes. It writes each trial's data to a csv file.
    - See the Output section for more information.

Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts

The Trial class resides in the voids of C# and doesn't really interact with the unity system at all. For this reason, you need to create a link between your unity project and your custom trial script.

To do this, you need to create a script that inherits from the ConfigOptions class. Then, create an empty GameObject in your scene called "ConfigOptions" and drag your new script onto this GameObject. Now, inside this script you can add any public fields that need to be used as options in your experiment. For example, you could add a float to set the speed of something that moves in your trial, or add a field that references another GameObject in your scene.

In you custom ConfigOptions which you called "MyConfigOptions", you could add a line:

```
1.  public GameObject gameObjectThatMoves
```

Then, in your custom Trial script you can load this using the following

```
1.  MyConfigOptions opts = experiment.ConfigOptions as MyConfigOptions;
```

This line loads the options not as the base ConfigOptions, but instead as your custom class.

Then you can access the gameobject with:

```
1.  opts.gameObjectThatMoves
```

This allows you to use the config options to drag and drop GameObjects and other things in your unity scene that need to be controlled or accessed during your experiment.

You can extend your custom options class as much as you want to add as much functionality to your trials as you need.

Debug Mode

You can run your experiment in Debug Mode during testing. This is useful for quickly prototyping changes without having to set everything up each session.

You can run in debug mode from the Experiment runner window (in the BML menu)

Selecting debug mode in the session settings has several advantages.

- It allows you to save output without the possibility of overwriting your real output files
- You can change the functionality of your experiment when debug mode is on.
    - You code special functionality for debug mode using something like the following:

```
1.  if (experiment.session.DebugMode) {
2.      //do something for debug
3.  }
4.  else {
5.      //do something normally
6.  }
```

- Output from debug mode is saved in Assets/BML_Debug folder.

Output

By default, the toolkit outputs your experiment results as a .CSV file. It automatically numbers your trials by block and by trial number inside each block. It adds a column for each variable including your dependent variables. To add a custom column, create a dependent variable of the appropriate type, and set its value in each trial.

The best place to write data to your dependent variables is in the Post() method of your custom Trial script.

Logging

The BML Experiment Toolkit keeps a log of all sessions. You can access it in the Assets > BML_ExperimentToolkit > Data Folder

Customizing further

The package is provided open source, so you can make any modifications you like. If you notice a missing feature, or write something other people might use, consider letting us know so we can add it to the package to be enjoyed by everyone.