Unit 3 Code Description

The experiments in this unit are more complicated than those in the previous unit. They involve collecting data and comparing it to existing experimental results and collecting vocal output from a model. Many of the ACT-R commands involved were described with the last unit's tasks. The new commands used will be described in this document along with more information on creating and running tasks, the parameters for adjusting finsts in the vision and declarative modules, buffer stuffing as it relates to the visual-location buffer, and a note about using the experiment files after changing them.

New ACT-R commands

open-exp-window and **open_exp_window**. This was introduced in the last unit. Here we see it getting passed a keyword parameter which was not done in the last unit. That parameter is a flag as to whether the window should be visible or virtual. If the visible parameter has a true value (as appropriate for the language) then a real window will be displayed, and that is the default if not provided (which is how the previous unit used it). If it is not true (again as appropriate) then a virtual window will be used and that will be demonstrated below in the section on running the experiments faster.

new-tone-sound and **new_tone_sound**. This function takes 2 required parameters and a third optional parameter. The first parameter is the frequency of a tone to be presented to the model which should be a number. The second is the duration of that tone measured in seconds. If the third parameter is specified then it indicates at what time the tone is to be presented (measured in seconds), and if it is omitted then the tone is to be presented immediately. At that requested time a tone sound will be made available to the model's auditory module with the requested frequency and duration.

schedule-event-relative and **schedule_event_relative**. This function takes 2 required parameters and several keyword parameters (only two of which will be described here). It is used to schedule ACT-R commands to be called during the running of the system. The first parameter specifies an offset from the current ACT-R time at which the function should be called and is measured in seconds (by default). The second parameter is the name of the ACT-R command to call specified in a string. The parameters to pass to that command are provided as a list using the keyword parameter params. If no parameters are provided then no parameters are passed to that command. If one wants to set the time using milliseconds instead of seconds then the keyword parameter time-in-ms in Lisp or time_in_ms in Python needs to be specified with a true value. By scheduling commands to be called during the running of the model it is possible to have actions occur without having to stop the running model to do so which often makes writing experiments much easier, and can also make debugging a broken model/task easier because the ACT-R stepper will pause on those scheduled actions in the same way it will for other model actions.

In the sperling experiment we use this to have the clear-exp-window command called to erase the display while the task is running. If you set the :trace-detail level of the model to high (not low which is how it is initially set) then you will actually see this command being executed in the trace on a line like this:

```
     0.941   NONE                    clear-exp-window (vision exp-window Sperling Experiment)
```

Since this was not generated by one of the ACT-R modules it specifies "NONE" where the module name is normally shown and then it shows the command which was evaluated and the parameter it was passed, which in this case is the representation of the experiment window.

correlation. This function takes 2 required parameters which must be equal length lists of numbers. This function computes the correlation between the two lists of numbers. That correlation value is returned. There is an optional third parameter which indicates whether or not to also print the correlation value. If the optional parameter is true or not specified then it is output, and if it is not true then it does not.

mean-deviation and **mean_deviation**. This function operates just like correlation, except that the calculation performed is the root mean square deviation between the data lists.

get-time and **get_time**. This function takes an optional parameter and it returns the current time in milliseconds. If the optional parameter is not specified or specified as a true value, then the current ACT-R simulated time is returned. If the optional parameter is specified as a non-true value then the time is taken from a real time clock.

output-speech. The output-speech command in ACT-R is very similar to the output-key command which we used in the previous unit. The output-speech command is called automatically by a microphone device (which is installed automatically when the AGI window device is installed) whenever a model performs a speak action using the **vocal** buffer. The command is executed at the time when the sound of that speech starts to occur i.e. it is essentially the same time that would be recorded if we were using a microphone to detect a person's speech output. It is passed two values which are the name of the model which is speaking and the string containing the text being spoken.

Response recording note

As was mentioned in the last unit, the monitoring functions are called in a separate thread from the main task execution. As will be the case throughout the tutorial we are not using any special protection for accessing the globally defined variables in the different threads to keep the example code simple, but in the subitizing experiment there is actually the possibility for a problem since we are setting two different variables in the monitoring function and both are used in the main thread. Without protecting them it is possible for the code in the main thread to try and use them both after only one has been set which could result in an error. To avoid that here we set the response-time variable first since the main thread is waiting for the response variable to change before it tries to use the response-time value. That is sufficient to avoid problems in this task (we could have also waited for both to be set before continuing as an alternative), but the best solution would really be to use appropriate thread protection tools.

Buffer stuffing

The buffer stuffing mechanism was introduced in this unit, and with regard to the **visual-location** buffer it mentioned that one can change the default conditions that are checked to determine which item (if any) will be stuffed into the buffer. The ACT-R command which can be used to change that is called **set-visloc-default**, and that is typically placed into the model definition when a different specification is needed. The specification that you pass to it is the same as you would specify in a request to the **visual-location** buffer in a production. Here are a few examples:

```
(set-visloc-default :attended new screen-x lowest)

(set-visloc-default screen-x current > screen-y 100)

(set-visloc-default kind text color red width highest)
```

set-visloc-default – This command sets the conditions that will be used to select the visual feature that gets stuffed into the **visual-location** buffer. When the visual scene changes for the model (denoted by the proc-display event which is only shown when the :trace-detail parameter is set to high) if the **visual-location** buffer is empty a visual-location that matches the conditions specified by this command will be placed into the **visual-location** buffer. Effectively, what happens is that when the proc-display event occurs, if the **visual-location** buffer is empty, a **visual-location** request is automatically executed using the specification indicated with set-visloc-default.

Visual and declarative finst parameters

To adjust the number of finsts that a model's vision module has you can use the :visual-num-finsts parameter. It can be set to any positive integer and the default value is 4. To change the duration of finsts the :visual-finst-span parameter can be used. It can be set to any positive number which specifies how long the vision module can maintain a finst in seconds. The default value is 3 seconds. The starting model for the subitizing assignment sets both of those parameters to 10 to make the modeling task easier – the model has ten finsts available and each will last for ten seconds.

The declarative module has similar parameters that work the same way. The number of declarative finsts is set using the :declarative-num-finsts parameter, and it has a default value of 4. The duration for declarative finsts is set using the :declarative-finst-span parameter, and the default value is 3 seconds.

Speeding up the experiments

Because it is often necessary to run a model multiple times and average the results for data comparison, running the model quickly can be important. One thing that can greatly improve the time it takes to run the model (i.e. the real time that passes not the simulated time which the model experiences) is to have it interact with a virtual window instead of displaying a real window on the computer. The virtual windows are an abstraction of a window interface that is internal to ACT-R, and from the model's perspective there is no difference between a virtual window and a real window which is generated by the AGI commands. Another significant factor with respect to how long it takes is whether or not the model is being run in real time mode. In real time mode the model's actions are synchronized with the actual passing of time, but when it is not in real time mode it uses its own simulated clock which can run much faster than real time. Typically, when the model is running with a real window it is also running in real time so that one can actually watch it performing the task and to make sure the task display is updating appropriately.

Since it is usually very helpful to use a real window when creating a model for a task and debugging any problems which occur when running it all of the AGI tools for building and manipulating windows work exactly the same for real windows and virtual windows. All that is necessary to switch between them is one parameter when the window is opened. Thus, you can build the experiment and debug the model using a real window that you can see, and then with one change make the window virtual and run the model more quickly for data collection.

The experiment code as provided for both of the tasks in this unit uses a real window and the model is run in real time mode so that you can watch it interact with that window.

To change the windows in those tasks to virtual windows requires changing the call to open the window to specify the visible parameter as not true (**nil** in Lisp and **False** in Python).

Here are the current lines from the sperling experiment files:

```
          (window (open-exp-window "Sperling Experiment" :visible t))

     window = actr.open_exp_window("Sperling Experiment", visible=True)
```

Here in red is what would need to be changed to make those virtual windows instead:

```
          (window (open-exp-window "Sperling Experiment" :visible nil))

     window = actr.open_exp_window("Sperling Experiment", visible=False)
```

A similar change could be made in the subitize experiment from this unit.

To change the code to run the models in simulated time requires not specifying the second parameter as true in the call to run.

Here ares the calls that currently exists in the sperling files:

```
     (run 30 t)

     actr.run(30,True)
```

Removing that second parameter will run the models in simulated time instead of real time:

```
    (run 30)

    actr.run(30)
```

Something else which can improve the time it takes to run a task is to turn off any output which it generates. As was mentioned in the unit you can turn the model trace off by setting the :v parameter to **nil**. If there is any other output in the experiment code as the task is running you will also want to disable that.

For the sperling experiment the given code prints out the correct answers and model responses for each trial, and turning that off will help if you want to run a lot of trials to see the average performance. In that code we have already added a variable which is used as a flag to control that output. The global variables *show-responses* and show_responses control whether that information is printed. In the given code they are set to true values, but if you change those to **nil** or **False** (respectively) it will suppress the output.

Adding a variable to control whether any output from the experiment is printed can be a useful thing to add when creating tasks for models that you want to run many times so that you can see the output when needed and quickly turn it off when you no longer need it.

Changing Experiment files

One last thing to note is that if you change the code in the experiment file then you will need to load that file again to have the changes take effect. For the Lisp versions of the task you can simply load the file again to have the changes take effect, but for Python just importing the module again in the same session will not work. One way to deal with that is to also import the importlib module. That provides a reload function which can be used to force the module to be reimported to reflect the changes. Here is an example of using that to load the sperling module again after changing it:

```
>>> import importlib
>>> importlib.reload(sperling)
```

Alternatively, if you use the “Import Python module” button in the Environment then it will automatically use importlib.reload if the module has been imported previously.