

UFIEYD-20-1

Introductory Programming for Audio I  
Max/MSP Programming Handbook



Dr Thomas Mitchell

2010/11



# Contents

Contents .....	3
Chapter 1: Introduction .....	5
1.1 About IPA I .....	5
1.2 Weekly tasks .....	5
1.3 Module Schedule .....	6
1.4 Getting Started .....	6
1.5 Working at Home .....	7
Chapter 2: Practical Sheets .....	9
2.1 Practical 1 – Familiarisation .....	11
2.2 Practical 2 – Message Ordering and MIDI .....	23
2.3 Practical 3 – Numerical Operations in MaxMSP .....	31
2.4 Practical 4 – More numerical operations in MaxMSP .....	41
2.5 Practical 5 – Sequences and External Control .....	51
2.6 Practical 6 – Tables .....	59
2.7 Practical 7 - Storing Presets and Customised GUIs .....	65
2.8 Practical 8 – Subpatchers and Abstractions .....	71
2.9 Practical 9 – Subpatchers and Abstractions II .....	79
2.10 Practical 10 - More Lists .....	85



# Chapter 1: Introduction

Welcome! This is the course handbook for the Max programming component of Introductory Programming for Audio I (IPA I). Here you will find all the material that you need for the Max programming sessions of IPA I.

## 1.1 About IPA I

Programmed electronic devices are abundant in every sector of the music industry. In whichever area you choose to focus your studies - in the creation or use of audio devices - an intimate understanding of their inner-workings will naturally enhance your abilities.

Throughout this module you will learn important fundamental programming skills and concepts which will enable you to build basic computer programs that perform audio and music related tasks.

## 1.2 Weekly tasks

Each week there will be two practical sessions, a lecture and a Peer Assisted Learning (PAL) session. Attendance is compulsory at every timetabled session. A new programming concept will be explored each week and, where possible, exercised in the programming languages C and Max/MSP.

The following section 1.3 sets out the schedule for the module where you can see the order in which topics will be covered. Each practical session has a specific set of sheets with instructions, information and exercises which must be completed before the following week. It is very important that all exercises are completed in the intended order as the material is incremental; that is, each practical builds upon the last.

**The secret to passing this module (and ultimately happiness) is to follow the schedule week-by- week!**

## 1.3 Module Schedule

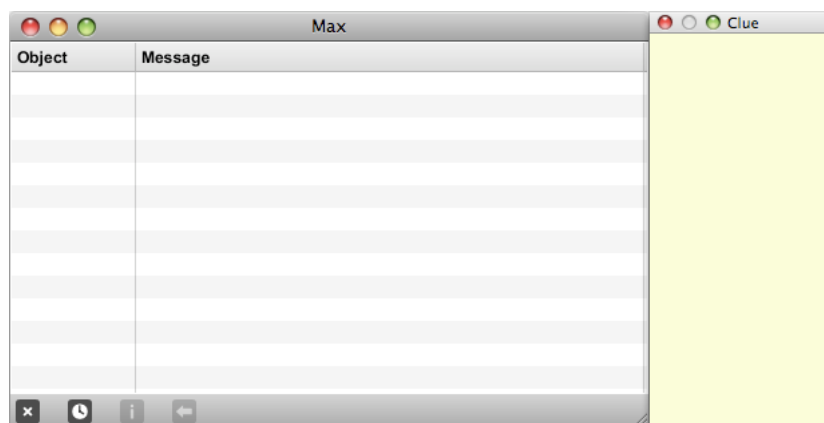
Practical Number	Week Beginning	Week Num	Lecture and Practical Topics	Notes
1	27-Sep	9	Welcome, intro, familiarisation	
2	04-Oct	10	Variables Program I/O, MIDI	
3	11-Oct	11	Arithmetic Processing	
4	18-Oct	12	Conditional processing further MIDI	
5	25-Oct	13	Further conditional processing, sequencing	Assignment Set
6	01-Oct	14	Loops, further sequencing	
7	08-Nov	15	Further arithmetic, sample playback & matrix	
8	15-Nov	16	Abstraction - functions and subpatchers	
9	22-Nov	17	Further functions, scope, bpatchers, presets	
10	29-Nov	18	Simple data structures, arrays, lists	
11	06-Dec	19	Free sessions for assignment work	Assignment Deadline
12	13-Dec	20	Exam preparation and mock	

## 1.4 Getting Started

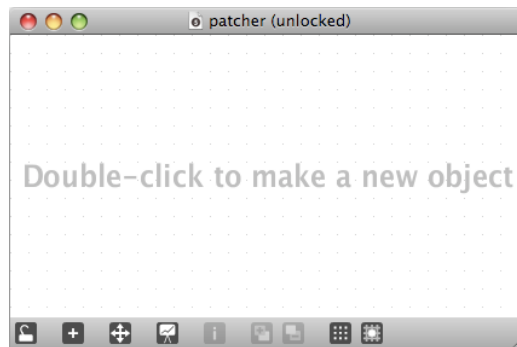
If this is your first IPA1 Max session follow these instructions:

1. Log in to the Macintosh
2. Navigate to the file from /Applications/Max5/MaxMSP and drag the file to your dock so that it is easy to locate in future.
3. Launch MaxMSP by clicking on the new Max icon on your dock.

There are two windows in MaxMSP which are very useful for debugging, especially when in the early stages of learning the system. These are the Max Window and the Clue Window. You should ensure that both of these are visible by selecting them from the Window menu (i.e., menu items **Window→Max Window** and **Window→Clue Window**). The Clue Window is a floating window so should remain visible at all times. The Max Window is **not** a floating window so you should try not to obscure it with too many other windows as it provides helpful feedback (especially errors).



The window that you will use most frequently in MaxMSP is the **patcher** window: this is the document window where you will operate and build programs. A program built in a patcher is often referred to as a **patch**.



## 1.5 Working at Home

To ensure that you keep up with the schedule you may wish to complete certain exercises at home which can be done on OSX or Windows. All you need to do is copy your ipa1/ folder to your computer at home. Alternatively, all of the files can be downloaded from here:

<http://www.cems.uwe.ac.uk/~tjmitche/aserve/ipa1.zip>

You will also need the Max 5 application which can be downloaded from here:

<http://cycling74.com/downloads/>

Max/MSP is proprietary software; as such you will need a licence to use it. The licence options and prices are shown in the table below:

Licence	Authorisation Period	Price
Demo licence	30 days	Free
9-month temporary student licence (+\$39 discount on full licence)	9-months	\$59
Full student licence	No limit	\$250
Full non-student licence	No limit	\$699

Licences can be purchased from:

<http://cycling74.com/shop/>





## Chapter 2: Practical Sheets

In this chapter you will find the practical sheets for the Max programming sessions. Each practical includes instructions, exercises and homework which must be completed within the week that practical is scheduled. Below is a table of the practicals and the week in which they are scheduled.

Practical Number	Title	Page	Week Beginning
1	Familiarisation	11	27-Sep
2	Message Ordering and MIDI	23	04-Oct
3	Numerical Operations in MaxMSP	31	11-Oct
4	More numerical operations in MaxMSP	41	18-Oct
5	Sequences and External Control	51	25-Oct
6	Tables	59	01-Nov
7	Storing Presets and Customised GUIs	65	08-Nov
8	Subpatchers and Abstractions	71	15-Nov
9	Subpatchers and Abstractions II	79	22-Nov
10	More Lists	85	29-Nov



---

## 2.1 Practical 1 – Familiarisation

The purpose of this session is to introduce the fundamentals of the MaxMSP programming environment, which you will be using for the MaxMSP component of the Introductory Programming for Audio module (in addition to C Programming).

By the end of this session you should be familiar with:

- the **MaxMSP** graphical development environment;
- the **Max** and **Clue** windows;
- Max **objects**, **messages** and **patchcords**; and
- building simple Max **patches**.

### 2.1.1 Getting started

If you have not already done so, follow section 1.4 ‘Getting Started’ of this handbook (p6). Then navigate to your desktop ipa1/Max/practical01 folder.

### 2.1.2 Introductory Tutorial Zero

Choose the menu item **Help→Max Help**, this should bring up MaxMSP’s built-in documentation.

In the right-hand panel of the documentation window click the “Tutorial Zero Video” link under the “Max” subheading for a brief introduction to MaxMSP (this video is just over four minutes long).

### 2.1.3 Adapted Tutorials

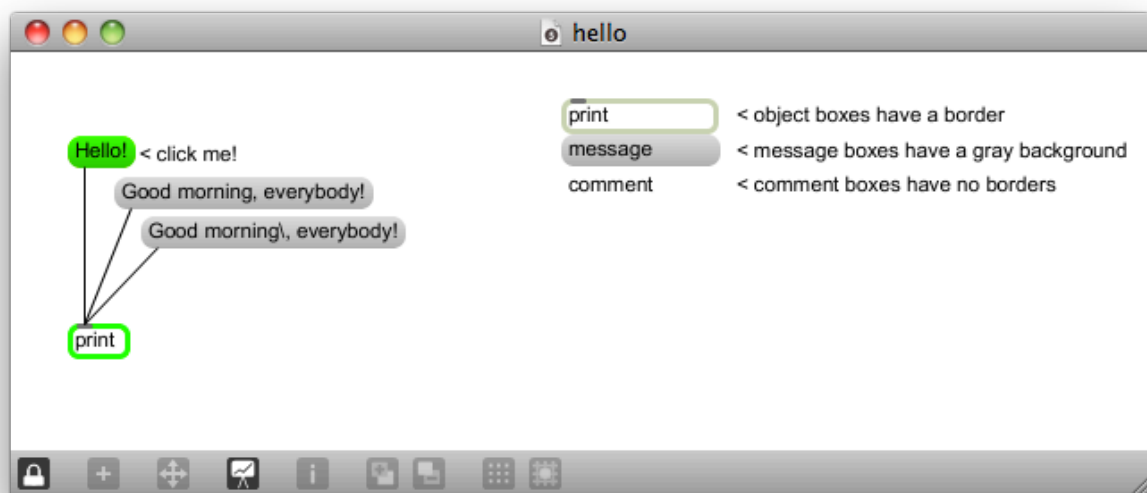
The remainder of this section is adapted from the Max tutorials 1-4.

#### Tutorial 1: Hello World – Creating objects and connections

In this tutorial, we will examine the building blocks of a Max patcher: the **object** box, the **message** box and the **comment** box. We will also delve into some of the basic editing functions provided by the Max environment.

This brief tour of a simple patcher will show the fundamental units of programs written in Max: **object** boxes that perform specific actions within a patch, **message** boxes that store messages which can be sent to objects and **comment** boxes that allow us to document our patchers to make them easier to understand. The following patch will introduce these three fundamental box types.

Navigate to your desktop ipa1/Max/practical01/ folder and open the patch `hello.maxpat`. You should see the patch shown overleaf:



### Looking Around

At the top right-hand side of the patch window, we can see three different box types.

The first element is the box labelled **print**. This is an **object** box, and can be recognised by its rounded outline and (in this case) a small "port" at the top left that we call an **inlet**. Object boxes are the basic logic element of Max – they contain functions that perform some sort of task, and operate like miniature programs within the larger environment.

Just below the object box is a **message** box (labelled with the word "message"). These are visually different from the object box – they have a gray background, and do not have an outline. Message boxes contain some information (called a message) that can be sent to objects, and can operate as either commands or control data.

The third box doesn't look like a box at all – it appears to be text on the background of the patcher window. There is actually a box around this text, but it is not visible until we decide to edit our patch. This **comment** box is used to add text for labelling controls (e.g. "click me"), or for adding comments to your patch to make its operation more apparent to you or anyone you might share your patch with.

All Max programs function by passing messages between objects. Object boxes can be connected to one another, and message boxes can be used to originate or, in some cases, process messages that are used to control other objects.

### Unlocking the patcher

When you first opened this patcher, it was in a "locked" state – meaning that we cannot edit any of the objects or text in the window. When a patcher is locked it can be used as a program. However, we can edit and manipulate these programs to do whatever we like, let's "unlock" this patch and do some editing.

Choose Edit from the View menu. You will see several changes in the patcher. First, the outlines surrounding the **comment** boxes are revealed (we told you they were there). Secondly, the **message** and **comment** boxes show inlets that were hidden in the locked state. In addition, ports on the bottom of these boxes are revealed; these are outlets.

In addition to the visible changes, we are now free to edit the contents of the window. Double-click inside the **object** box in the upper-right that says **print**; the text is selected, and you can type the name of another object name into that box. Type "metro" (without the quotes), and click outside the **object** box to complete the edit. We have now changed the function of that object (it is now a **metro** object). You will see the **object** box change size, and the number of inlets and outlets will change as the function of the object has changed. We learn Max by building up a vocabulary of objects and how they connect to other objects (i.e. what messages they understand and transmit).

Double-click inside the **message** box right below that. Change the message text to anything (for example, type in the word "anything"), then click outside the box to accept the change. Note that changing the message text does not change the inlets or outlets, since only the contents (but not the function) of the message box has changed.

Finally, double-click inside the comment box and change its text. Since the comment box has no function other than adding text to the patch, you are free to put in any text that pleases you.

### ***Interacting with a locked patcher***

Lock the patch by unchecking **Edit** in the **View** menu. Let's see what happens when we click on each of the box types. In the upper right-hand corner of the patch, click on the **object** box (which we changed from **print** to **metro**) – nothing happens. Clicking on the **message** box directly below it, however, changes the background color – obviously, something is happening when we click on that box.

On the left side of our patcher, there are three **message** boxes connected to a **print** object with lines (called patchcords). Click on each of these **message** boxes in turn, then look at the Max window; you will see that the contents of the **message** boxes we clicked are displayed there. This was a result of clicking on the **message** boxes: they generated a message (the content of the message box) and sent it down the patchcord to the **print** object. The **print** object takes any message it receives in its inlet and "prints" it out in the Max window.

### ***Getting help***

The **print** object is obviously doing something to send text to the Max window; in fact, every **object** box performs some sort of task. To determine what function an **object** box performs, you can view several types of document to get help.

**Unlock** the patcher and select the **print** object box by either clicking into it or clicking and dragging the mouse over it to highlight it. **Select Open print Help** from the **Help** menu. A new patcher window (called a Help patcher) opens with an example that shows the **print** object in action. Note that this is an actual Max patch, and can be unlocked, edited, or copied elsewhere, retaining any of the displayed functionality.

Close the **print** object's Help patcher and reselect the **print** object. **Select Open print Reference** from the **Help** menu. A browser window is displayed with the *Reference Manual* page for the **print** object. This gives detailed information about the

object, all of its inlets and outlets, and even provides an example patch for your reference. This is perhaps the most detailed information available for any object. In both the Help patcher and the Reference page, objects and topics relevant or similar to the object you're viewing can be viewed by clicking on links in the See Also section at the bottom.

### **Create some new messages**

Let's add some objects to our patch, and create our own message passing logic. First, with the patcher unlocked, click on the palette icon (the "+" symbol at the bottom of the window) and select the first object – it is the **object** box. This will place an empty **object** box on your patcher window, ready for you to enter its name. Enter "print", then click outside the box to accept this change. Move the object somewhere in the patcher with some clear space above it by clicking and dragging it around.

Next, click on the palette icon and select the second object, a **message** box. When it appears in your patcher window, type in the message "Goodbye!". Place this **message** box above the **print** object. Finally, **connect** the two boxes by clicking on the outlet of the **message** box, dragging the mouse to the inlet of the print object, then releasing the mouse button. This will create a patchcord between the two objects, allowing messages to pass from the **message** box to the **print** object.

**Lock** the patch and click on the new "Goodbye!" **message**. You will see the text "Goodbye!" appear on the Max Window. Congratulations – you've just done your first Max patching!

### **Tutorial 1: Conclusion**

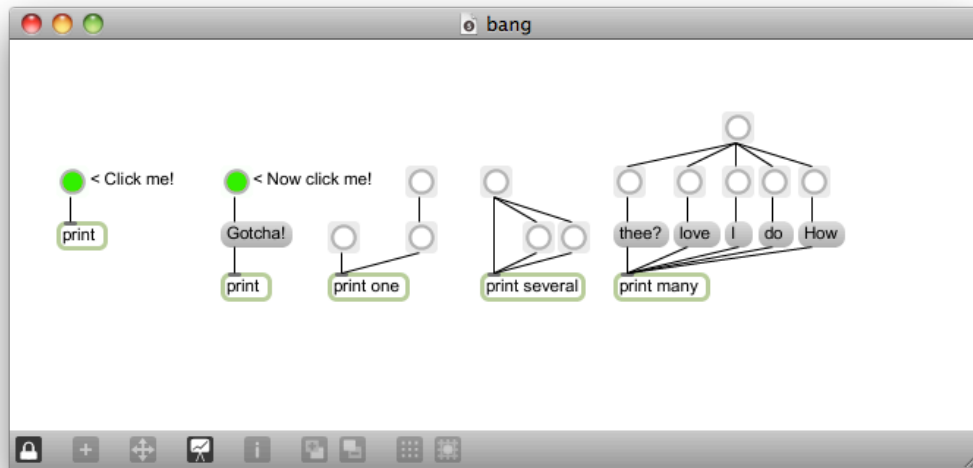
The three main elements of a Max patch (**object** boxes, **message** boxes and **comment** boxes), along with patchcords, are the core of all Max programs. Each type of element responds to a variety of messages and editing functions; these can be explored by viewing the help files and Reference Manual documents. Each of these aspects will be expanded upon in later tutorials.

## **Tutorial 2: The Bang Message**

In this tutorial, we will work with the **bang** message as a way to follow messages through the connections in a Max patcher.

This simple patcher will let us look at how to generate messages in Max, and how these messages execute through a patcher in a pre-defined order.

Navigate back to your desktop ipa1/Max/practical01/ folder and open the patch bang.maxpat. You should see the patch shown overleaf.



### Looking around

At the left side of the patcher window is a very simple patch, with a **button** user interface (UI) object connected to a **print** object. If you click on the **button** (marked by the comment, "Click me!"), you will see that the Max window displays a **bang** message – the output from the **button**.

The **bang** message has a specific use within Max – it's the message that tells many objects to "do that thing you do". As a result, sending the **bang** message to other objects will normally cause them to send messages from their outlets. In the second part of the patcher (to the right), there is a **button** (labeled "Now click me!") connected to a message box (with the message Gotcha!), which is subsequently connect to a **print** object. If we click on the message box, we can see the Gotcha! message sent to the Max window – just like in the previous Tutorial. Now, click on the **button** object. We see the same Gotcha! message appear in the Max window. How did that happen?

Let's follow the messages. First, clicking on the **button** object sends a **bang** message from its outlet. This message follows the patch cord to the inlet of the message box. When the message box receives the **bang** message, it performs its task – it sends out its message. Hence, the Gotcha! message is sent from the message box outlet to the **print** object, where it is printed to the Max window.

Most Max objects (the message box and **button** included) will interpret a **bang** as an instruction to perform their main task using whatever information they have available. Since the primary function of the message box is to construct and pass messages, sending a **bang** to any message box will cause that object to output whatever message is stored in it.

### Following messages through connections

The **bang** message is not only useful for triggering message boxes – as we mentioned above, it can act as a trigger for many other objects as well. Click on the **button** at the top-right of the third patch – it is connected to a second **button** that is triggered by a **bang** message, which then sends its message (**bang**) to the **print** object.

Note that this **print** object has an argument after it ("one"). If you look at the Max window, you will see that instead of the word "print" appearing in the object column, the argument to the **print** object appears - "one". This is useful for debugging Max

patchers, as it allows you to create multiple print objects with custom labels to differentiate them from one another in the Max window.

Now unlock the patcher, and connect the outlet of the top **button** to the lone **button** to its left. Notice that this doesn't break the connection that already exists; outlets can be connected to more than one object, and the same message will be sent down each of the patch cords. Lock the patch and click on the top-most **button**. The **bang** gets sent to both **button** objects, and each of them produces a **bang** message of their own.

By a similar token, multiple patch cords can also be connected to a single inlet of an object. In this case, both of our **button** objects are connected to a single print object. When you click on the top **button**, the output of both **buttons** that receive **bang** messages are routed to the print object, and two **bang** messages are displayed in the Max window.

The next patch over demonstrates both of these principles. Clicking the **button** at the top of the patch sends three **bang** messages to the Max window (through a print object labelled "several"). The print object receives a **bang** from both of the "second-tier" **button** objects (triggered by our topmost button) as well as a single **bang** from the top **button** itself, connected directly to the print object.

We can take this idea further with the right-most patch. It has a number of message boxes, all connected to a single print object named "many". Each message box has a **button** attached to it, and a higher-level **button** controlling them all. You can make individual message box objects print their message by either clicking on the message box itself, or by clicking on their attached **button** objects. If you want to trigger all of the message boxes at once, you can click on the top-most **button**.

When you hit the top **button**, notice the order in which the messages are displayed in the Max window. The messages are not in an arbitrary order – they are generated based on the spatial organization of the objects in the patcher, executing from right-to-left.

Unlock the patch, and add several more **button** objects. You can do this by bringing up the palette and selecting the **button** object's icon, or by simply clicking the 'b' (for **button**) key on the keyboard in an unlocked patcher. Connect them to other **button** objects on that right-hand patch. Try different combinations of multiple outlet connections and multiple inlet connection. As you connect each **button**, try to guess which messages will display and what order they will be sent to the Max window.

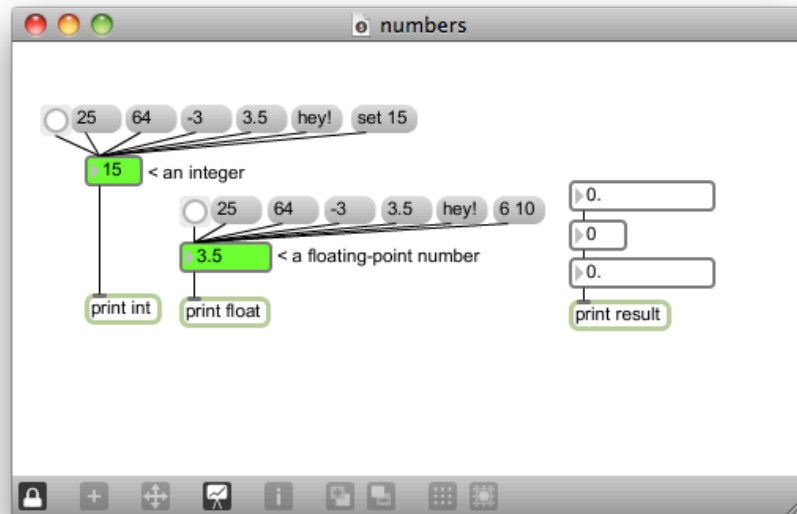
## ***Tutorial 2: Conclusion***

The **bang** message is among the most powerful messages in the Max environment; it causes other objects to trigger their output, and can be used to create matrices of connections that produce the desired output. We've also seen that both inlets and outlets can have multiple connections, and that these connections output their messages in a predictable order. We've also seen that some objects, such as the print object, can be assigned an argument, which causes the object to change its behavior. An argument to the print object allows you to identify the source of messages printed to the Max window.



## Tutorial 3: Numbers – Numeric Data in Max

In this tutorial, we'll look at how max deals with numbers. Navigate back to your desktop ipa1/Max/practical01/ folder and open the patch numbers.maxpat. You should see the patch shown below:



### Number boxes

The patch window contains several small patches that feature **number** boxes and various other objects we have already met. The left-most patch looks at a new object (in green) called the **number** box. It deals with **integer** (often called whole) numbers, and accepts many different messages. Click on some of the connected message boxes, and notice their effect on the **number** box (as well as any messages that might appear in the Max window via the print object).

The most noticeable effect comes when numeric data is sent to the **number** box: it responds by displaying the data as well as sending the number from its outlet. When a floating-point number (such as 3.5) is sent to the object, it takes the integer portion of the number and truncates the decimal portion. If a non-numeric message is sent (such as the Hey! message), the Max window displays a complaint by the object, stating that it doesn't understand that type of message.

Two other special messages are accepted by the **number** box. First, sending a **number** box a bang forces the current value to be output without any change to the data. Secondly, preceding a number with the set message will change the data without any output – in essence, providing a way to "silently" change the contents of a number box. The set method is fairly common in Max objects as a way to manipulate the state of the object without triggering any additional messages.

The patch just to the right is similar, but uses a floating-point **number** box (**flonum**) that allows for decimal values to be displayed and output. Many of the messages produce similar results to the integer **number** box. However, when a floating-point value is received, the entire value is saved (and displayed) in this **number** box. The bang and set messages behave as we'd expect, and the Hey! message is not valid.

### ***The user interface of the number box***

In addition to sending them messages, we can also directly enter and manipulate the values stored in **number** boxes. For example, we can click-drag within the box to manually change the **number** box contents. This is useful for performance situations where we may want to change a number in real-time while the patch is running. Rather than having to set up message boxes for every possible value we might need, we can manipulate the **number** box to change a setting. If we click-drag on the integer **number** box, we see that the Max window displays a stream of data – the number box generates output for each of the values that we scroll through.

In the case of the floating-point **number** box, the change in value is based on where we place the mouse before we click-drag. If the mouse is on the integer portion of the data (to the left of the decimal point), click-dragging will produce changes similar to the integer number box. However, if we position the mouse of the fractional portion of the data, we can change any of the displayed digits.

If you would like to enter a specific value into a **number** box, click on it (when the patch is locked), type in the value, and then either press the return key or click elsewhere in the patcher. This is a convenient way to enter numbers accurately without having to resort to a message box (or having to scroll for days). Notice that the new data is displayed in the **number** box, and the object outputs its value as well.

The third patch has three **number** boxes hooked up to each other. Here, you can see the visible difference between integer and floating-point **number** boxes – the floating-point versions have a displayed decimal point. If you change the top-most (floating-point) **number** box, you will see its data being used to update an integer **number** box, which subsequently updates the next (floating-point) **number** box. You can also directly manipulate the integer one to see how its messages affect the floating-point variant.

Enter a large number into the top-most **number** box – something like 60000. You will see that the integer **number** boxes can't display this number, since it is too large for the provided display area. Unlock the patch and move your mouse to the right-hand side of the **number** box object. You will see a small "handle" appear, and the cursor will change to a left/right arrow; this is called a grow box, and can be used to re-size most user interface objects in Max. Click-drag on the handle to change the width of the number box, then click on a blank area of the patch to deselect the object. You will see an immediate change in the integer display.

Finally, enter a ridiculously high number, like 7245569558. The integer **number** box changed to -2147483648. How did that happen? While floating-point numbers have a very high threshold of recognized value, the 32-bit integers have a limit – in this case, the maximum value of an integer is 2147483520. Any numbers greater than that cause the integer value to "overflow", and become the greatest negative number. Other very high values might provide other values, depending on how far over this limit the number is.

### ***Tutorial 3: Conclusion***

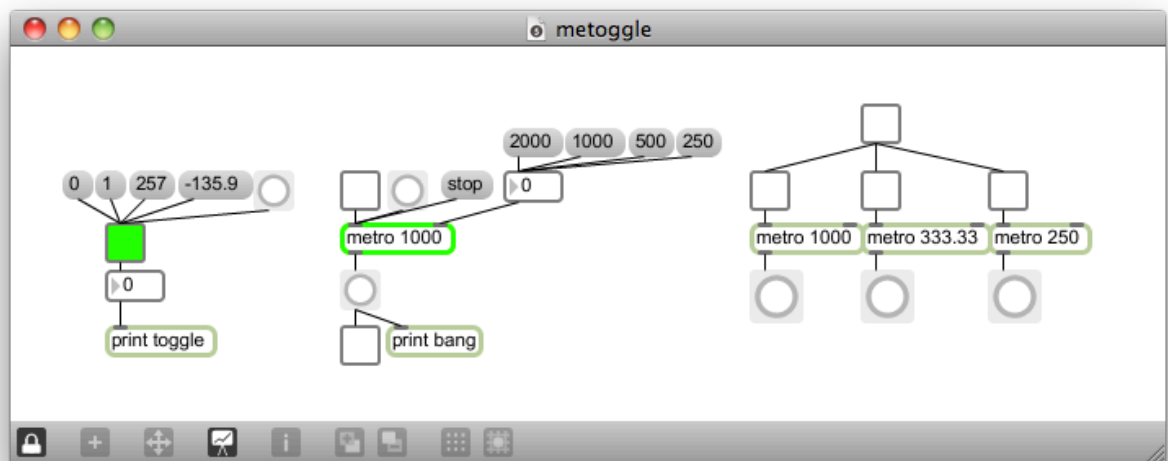
Max can send numeric data as messages, just like text. There are a few user-interface objects that help us construct and process numeric data, such as the integer and floating-point **number** boxes.

## Tutorial 4: Metro and Toggle – Automated Actions

This tutorial focuses on automated actions and the ability to control them. We will work with the **toggle** user interface object to control the **metro** timing object. We will also look further into object arguments and overriding those arguments.

One of the most used objects in your home is the light switch – it provides a simple user interface to a complex system (the lights in your home). Likewise, the **toggle** object, combined with the automated timing function of **metro**, offer a simple interface into an otherwise complicated world of automated and scheduled actions.

Navigate back to your desktop ipa1/Max/practical01/ folder and open the patch metoggle.maxpat. You should see the patch shown below:



### Fun with the toggle object

The left-most patch features a series of message boxes (and a button) connected to a square box. Start by clicking in that box – we see its interior fill with a checkbox. A 1 message was also sent from the object downstream to the number box connected below it. Clicking in the object again will remove the checkbox and send out a 0. That's all this object does - it outputs either a zero or non-zero value, depending on its state (off or on). This object's formal name is **toggle**, and it is one of the most-used objects in Max programming.

Unlike most of the checkboxes that we run into when using our computers, this checkbox can be automated by sending it messages. Click on the message boxes that are connected to the **toggle** to see their effect on the object's state. By looking at the connected number box, we see that any number other than 0 turns it to "on" (the check mark is visually set); the **toggle** also "echoes" the value of the number that turned it on out its outlet. Clicking on the 0 message turns the **toggle** off, and outputs a 0.

The action of the button object is even more interesting. Clicking on the button switches the **toggle** object on and off, with the output (seen in the Max window and in the number box) alternating between 0 and 1. This is the root of the **toggle** object's name – its ability to "toggle" states based on incoming bang messages.

## ***Introduction to metro: the metronome object***

The patch in the center introduces a new object: **metro**. The metro produces bang messages, much like the **button** object. However, unlike **button**, **metro** objects send these messages automatically on a scheduled basis. Once started, **metro** will continue to send bang messages until stopped. Click on the toggle that is connected to the **metro**, and watch the result – it sends a bang message (in this case, to a **button** object) once every second. We can turn it off by un-checking (turning off) the **toggle**.

If we click on the button that is connected to the **metro** we see that it, too, starts the **metro** running. If we click on it again, we see that it immediately triggers a bang out of the **metro** immediately; rapidly click on the button, and the output continues being generated. It is only when we pause that we see the scheduled event generation. When a **metro** receives a bang, the **metro** will "re-start" itself and begin scheduling subsequent bang messages from the moment we triggered it. When we want to turn it off, we can click on the stop message to turn off event generation; telling a **metro** to stop with a message is equivalent to sending it a 0.

The rate at which bang messages are produced by a **metro** object is controlled by its period. The **metro** object takes an argument (the number 1000, in this case) that sets the time interval (in milliseconds) between messages. 1000 milliseconds equals 1 second, which is why we see the bang messages coming out at that rate. How do we change this rate? That is the function of **metro** object's right inlet – it allows you to send a new time interval setting without having to type in a new argument value.

In our patch, there is a number box with several message boxes connected to it. Click on the 2000 message, and (after the next scheduled period) the output will slow to one message every two seconds. Click on the 500 and the **metro** speeds up, sending bang messages twice a second. The message 250 will cause four bang messages per second. Changing the interval setting doesn't change any other aspect of **metro** (e.g. whether it is "on" or "off").

Changing the interval with a message in the right inlet also does not change the argument of the **metro**. Why not? Wouldn't it be convenient to see the new value? Perhaps, but it would mean that any changes to the object would change the value saved with the patch – and, most times, you want to maintain the typed-in argument since it functions as the default starting value of the **metro**. Many Max objects behave in this way: they have one or more arguments that can be overridden by messages sent into inlets.

## ***Running in parallel***

The third patcher in our tutorial features a single **toggle** attached to three additional **toggle** objects, each of which is connected to a **metro** with a different argument. Turn on the top-level **toggle**, and the **metro** objects generate a "galloping" light show. This is because the three time interval arguments of the **metro** objects are related: the first metro fires every second (1000 milliseconds), the second triggers three times per second, and the third fires every quarter-second. There is nothing managing this pattern – it just occurs because the scheduled output of the **metro** objects are constant, so the pattern keeps appearing. Note that the time interval of a **metro** can be both an integer (e.g. 1000) or a floating-point (e.g. 333.33) number.

Let's expand on the idea of creating rhythmic patterns by using some of the other ideas we just learned. We can change the pattern by changing the time values of each

metro; let's start by adding three message boxes, of values 500, 200 and 750, and connecting one to the right inlet of each metro. When we click them to change the time intervals of our metro objects, no apparent pattern exists. This is because we need to synchronize the start of the metro objects to see the pattern. This can be accomplished by turning the top-most **toggle** off and on again. The result is a quickly flashing center light, with a syncopated outside pattern.

Let's make another pattern, using message boxes for the values 200, 400, and 600, respectively. However, instead of using the **toggle** to switch the metro objects off and on, connect a single button object to the left inlet of all three **metro** objects. Click on the three message boxes, and then click on the button to restart all of the metro objects. The button forces the metro objects to restart in sync, giving us the expected "rolling" light show.

#### ***Tutorial 4: Conclusion***

Automated actions are at the heart of many Max performance patches, and **metro** objects are often used as the starting point for these actions. Once started, the scheduling system of Max provides a stable timing system for Max output, and we can allow the events to run without further interaction.

Controlling patches using the checkbox-like **toggle** object is also a key programming tool, since a single **toggle** can provide a simple on/off interface to much more complicated processes.

## **2.1.4 Additional Objects**

### **The counter object**

Familiarise yourself with the **counter** object. (You may need this for the exercise in section 2.1.5 below.) To do this, open a new empty patcher (choose menu item **File→New Patcher**) and insert a **counter** object in the patcher. Then explore **counter**'s "Help" and "Reference" files. (You should have learned how to do all of this by following the tutorials above.)

### **The select object**

Familiarise yourself with the **select** object using its "Help" and "Reference" as you have just done with the counter object. (You may need the **select** object for the exercise in section 2.1.5 below).

### 2.1.5 Stopwatch exercise

Using what you have learned in the previous sections, you are to build a stopwatch in MaxMSP. The stopwatch should have:

- **start** and **stop** buttons;
- a **reset** button; and
- a numerical display for **minutes**, **seconds** and **tenths of seconds**.

If you complete this with spare time you should add a “lap” (or “split”) button. (See <http://en.wikipedia.org/wiki/Stopwatch> for a definition of this common stopwatch function if you're unsure.)

### 2.1.6 Homework

If you do not finish the exercise in section 2.1.5 above before the end of the practical session you should complete it during your spare study time before the next MaxMSP practical session.

When the exercise is complete you should test the stopwatch. Are there any problems? Are there any better solutions? In what ways might these revised solutions be better than your original solution?

### 2.1.7 Finishing up

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

### 2.1.8 Important material

Before the next MaxMSP practical session you should be familiar with the following MaxMSP elements and concepts:

- the Max, Clue, and Patcher windows;
- connecting objects and messages using patchcords;
- the Max Help system;
- the basic structure of Max Reference files in Max Help;
- printing data into the Max window using the print object;
- the **bang** message;
- number messages;
- the toggle and button user interface objects; and
- the basic operation of the metro, select and counter objects.

---

## 2.2 Practical 2 – Message Ordering and MIDI

This session is intended to introduce message ordering and MIDI input/output in MaxMSP.

### 2.2.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop ipa1/Max/practical02 folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.2.2 Stopwatch exercise

You should have at least attempted the stopwatch exercise from last week (or have understood one or more of the solutions to it) before proceeding.

### 2.2.3 Adapted Tutorials

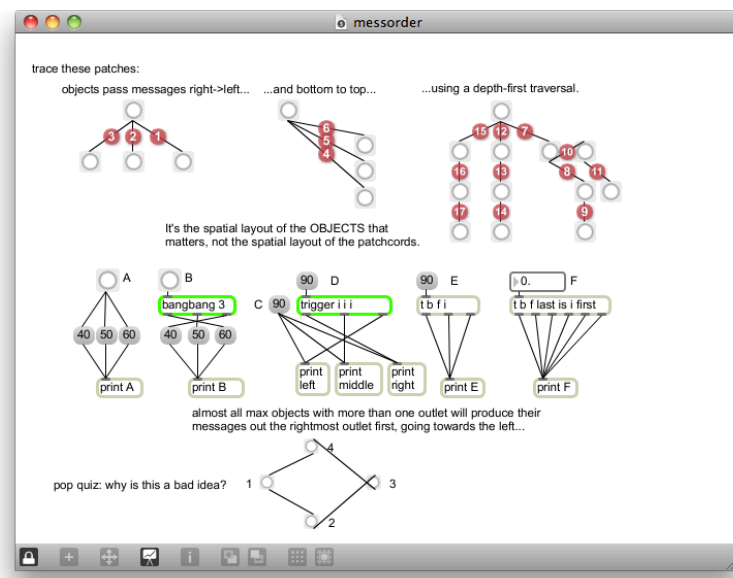
The remainder of this section is adapted from MaxMSP's Basic Tutorial 5 and MIDI Tutorial 1.

#### Tutorial 1: Message Ordering and Debugging

This tutorial is focused on message ordering – the sequence in which messages are passed from object-to-object and how objects generate them. We will also use some of the debugging tools in Max to determine how a patch is running.

Max patches often seem to have everything happening at once; in reality, messages are produced and acted upon in a specific order

Navigate to your desktop ipa1/Max/practical02/ folder and open the patch `messorder.maxpat`. You should see the patch shown below:



### ***Right-to-left, bottom-to-top***

This file contains a number of small patches that we will use to learn about the rules that messages follow. Click on top-most button in the top-left patch; it seems that all three of the connected button objects fire simultaneously. However, in reality messages are sent down the patch cables in a sequential order.

The easiest way to see this in action is to use some of the debugging tools in Max. You'll notice that the top row of patches in the tutorial have small red circles with numbers in them covering the patchcords. These are called watchpoints (specifically - breakpoint watchpoints). When we debug our patcher, we can use these watchpoints to stop the execution of the patch at a specific point. This is done by enabling the Max Debugger. Select **Enable Debugging** from the **Debug** menu. Once debugging is enabled, these watchpoints will help us see the way messages are passed in our patcher.

Click on the topmost button in the left-hand patch. Instead of the immediate "flash" of all the button objects lighting up, the rightmost patchcord begins blinking red. In addition, a window opens called the **Debug Window**. Looking at the window, you can see what kind of information it gives us. The window tells us that a bang message has been intercepted by a watchpoint; moreover it tells us which watchpoint was tripped (in this case, watchpoint #1), the name of the patcher and what class of object the sending and receiving object was (in this case, both are button objects). Select **Step** from the **Debug** menu (or press Shift+Command+T). You will see that the middle cord flashes; select **Step** again and you will see that the left-most cord flashes. Select **Step** once more to finish our patch trace. When the outlet of an object is connected to more than one inlet, the messages are sent in right-to-left order.

But what happens when the receiving objects are stacked vertically (and therefore at the same horizontal position)? With debugging still enabled, click on the topmost button in the middle patch on the top row. As the watchpoints fire, select Step from the Debug menu. When we Step through the patch, we see that the bottom-most button receives its message first, followed by the middle, then the top. So, we can see that Max has two levels of ordering: right-to-left, then bottom-to-top.

There is one more twist to message ordering, and that is the effect of messages that cause more messages to be generated. The third patch illustrates the issue. Click on the top-most button and Step through the order of the messages. By tracing through this matrix of button objects we see that messages travel through the entire depth of one message chain before sending a message to the next "branch" of objects. Notice that the Debug Window shows you the length of each message chain in a branch; it only clears when a branch has been completed.

Therefore, the message-ordering rules in Max are:

1. Right-to-left, then bottom-to-top.
2. Depth-first traversal of connected objects.

Note that, for determining the right-to-left or bottom-to-top ordering, it is the location of the connected object, not the location of the patch cords, that determines the message route.



As an exercise, create a new matrix of button objects, with both vertical and horizontal arrays of button objects (perhaps using the Align option from the Arrange menu), and multiple depths of objects. Use **comment** boxes to number the boxes in the order you think they will fire then trace the patch. Note that to add a breakpoint to a patch cable select the cable by clicking on it and select **Add Watchpoint - Break** from the **Debug** menu. If you were incorrect in your ordering repeat the exercise until you get it right.

Select Disable Debugging from the Debug menu, and let's look at the other patches in the tutorial patcher.

### **More message ordering**

Sometimes it is inconvenient to match the spatial order of the patch with our desired result. In the bottom row of patches, the left-most patch is well structured: it is cleanly laid out, and has the number boxes in ascending numerical order from left to right. If we wanted the messages to go from low to high, it is also wrong; when we click the top button, we see the numbers come out in reverse order in the Max window. This is because the message boxes are receiving bang messages in right-to-left order.

The next patch shows a corrected version of this patch, using a new object: **bangbang**. This object takes an incoming message and produces bang messages from its outlets in right-to-left order. The number of outlets is determined by the argument to **bangbang** (in this case, the 3 argument produces three outlets). The outlets are connected to the message boxes in the appropriate order; when we click on the top button, the Max window displays the messages in the preferred order.

Notice that the outlets of bangbang fire in right-to-left order, mimicking the message ordering of the patch itself. Most objects with multiple outlets will follow this rule: output is from the right-most outlet to the left-most outlet.

The **bangbang** object makes the message order explicit; that is, it forces the messages to follow a specific path regardless of spatial orientation, and lets us place objects anywhere we want in the patch, knowing that they will be triggered in deterministic order based on which outlets we trigger them from. Another message that provides explicit ordering is the **trigger** object. The **trigger** object accepts input, and outputs messages based on its arguments. The arguments determine the type of output, with options of l (for list), b (for bang), i (for integer), s (for symbol – a text message) and f (for floating-point number). You can also use specific integers, floating point numbers or symbols as constant outputs.


The next patch contains two message boxes with the integer 90. If we click on the one connected directly to the print objects, the Max window displays the typical right-to-left ordering of messages. However, if we click on the message box connected to the **trigger** object, we see that we can control the output to print in left-to-right (reversed) order. The sent messages are all printed as the integer 90, since our arguments for **trigger** were of type i (integer).

The next patch to the right also uses an incoming integer message (90), and is connected to a **trigger** object (using the object abbreviation: "t"). However, in this case, three different arguments are used. When we click on the message box, we see that the output of the three outlets are all different, matching the difference in

arguments. The right-most outlet, with argument of type **i** (integer), produces an unchanged 90 output. The middle outlet, set to type **f** (for floating-point), *casts* the incoming message to its floating-point equivalent (90.000000) and sends it to the print object. Finally, the left-most outlet, of type **b** (bang), accepts the incoming message but turns it into a bang. So, in addition to making the message order explicit, the **trigger** object can do type conversion of messages.

The right-most patch shows the use of constant values within the **trigger** object. When we change the floating-point number box to any number, the **trigger** object again converts the incoming number, but also includes a number of symbols (text messages) that are used as constant values. We also see that the incoming message did not change the output of the bang outlet – a bang is a bang, regardless of incoming message type.

### ***The trouble with loops***

The final patch in our tutorial file is a set of four **button** objects all interconnected in a loop. When we click on any of the **button** objects, the system comes to a halt with a stack overflow error. Why did this happen? Remember that Max does a depth-first prioritization of messages. However, a looped patch construction has an infinite depth, and therefore cannot be properly parsed. One **button** sends to the next, which sends to the next, which sends to the next, which sends to the first, triggering the cycle all over again. When a stack overflow occurs, Max shuts off its scheduler to interrupt the loop and displays an error bar at the top of the patch. Clicking on the bar will highlight the problematic object so the patch can be corrected (e.g. by disconnecting one of the **button** objects in the loop). The patch can then be switched "on" again by clicking the  button in the error bar.

### ***Tutorial 1: Conclusion***

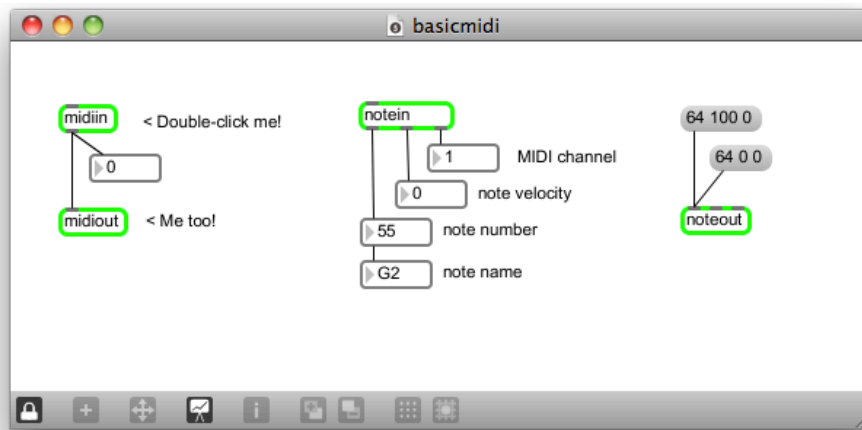
A deep understanding of message ordering rules is necessary to create properly functioning patches. The right-to-left, bottom-to-top, depth-first order is the implicit rule for message passing, but you can use objects like **bangbang** and **trigger** to make the ordering explicit. In all cases, however, you need to watch out for structural loops – they are literal patch-killers!

## **Tutorial 2: Basic MIDI**

This tutorial will cover some of the basics of using the MIDI (Musical Instrument Digital Interface) communications protocol. We will see how to receive MIDI information from input devices, send it to other hardware or software and select the device that will get the sent information.

While the prevalence of software-only music systems have reduced the need for MIDI devices, the MIDI protocol is still necessary for working with both hardware and software synthesizers and samplers. MIDI is also becoming more widely used by artists working with physical interfaces, since it provides a compact and easy-to-use communications protocol for receiving sensor information and producing control messages for motors and other controllable devices.

Navigate to your desktop `ipa1/Max/practical02/` folder and open the patch `basicmidi.maxpat`. You should see the patch shown overleaf:



The **basicmidi** patcher contains a number of small MIDI-specific patches. The left-most patch changes a number box anytime information is received from a MIDI input device (via the **midin** object), then sends that information out to another MIDI device (through the **midout** object). Double click on the **midin** object and ensure that the USB **Axiom 49 Port 1** device is selected now double click on the **midout** object and ensure that the **AU DLS Synth 1** is selected. You should now be able to play a piano sound on the Axiom via this part of the patch.

The **midin** and **midout** objects pass out and expect unformatted raw MIDI messages – if you route the output of the **midin** object to a print object, you will see that you have a serial stream of numbers that can be difficult to interpret. Max contains a number of objects that give us a little more control how we use MIDI data inside our program by selecting which types of MIDI events (notes, continuous controllers, etc.) we want to work with.

### ***All about notes: the notein and noteout objects***

The **notein** and **noteout** objects are an example of message-specific MIDI objects. They accept the input of a MIDI stream, and ignore all message types other than note messages - those traditionally caused by playing keys on a MIDI keyboard or pads on a drum trigger. As with the **midin** and **midout** objects, we can select the MIDI port by double-clicking on the object.

The second patch shows a basic note display; we can select a MIDI port, then view the notes that are received on that port. The **notein** object displays three pieces of information for each incoming note message: the note number (or pitch), the note velocity and the MIDI channel it was transmitted on. Play a MIDI keyboard or other controller that creates MIDI note messages, and see how the note information is displayed by the number boxes.

You will notice that there is no “on or off” display to differentiate keys being played (a “note-on”) and keys being released (a “note-off”) – rather, a note-off message is displayed as a note with a velocity of zero (0). This is a common MIDI convention, and it is used by many Max objects as the preferred way of displaying a note-off message. We have also used a number box with the **Display Format** attribute set to MIDI. This provides us with an easily-read cue as to the pitch of the note that is being played.

The next patch is a simple example of the reverse process: generating note messages to be played by a MIDI device. The **noteout** object expects pitch, velocity, and channel

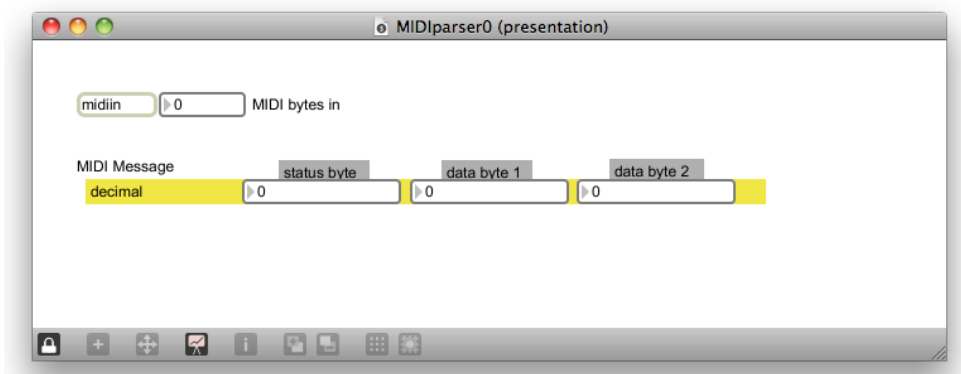
numbers to be received in its left, center, and right inlets, respectively. The left inlet (pitch) is hot, while the others are cold. In this case, we use Max's ability to decode a list of messages as the input to all of the inlets of an object. Thus, the message box labeled 64 100 0 is treated like three separate messages to each inlet: they are "channel 0", "velocity 100" and "note 64". This will send a note-on message for pitch 64 (E3) at a velocity of 100 to channel 0. The second message box is almost the same, but uses a velocity of 0 – which is the equivalent of a note-off message. Double-click on the noteout object, and ensure the **AU DLS Synth 1** device is selected. Then click on the first message box. You should hear the device sound the note. It should sustain until you hit the second message box, which will turn it off. This is a basic MIDI note event structure, and can be used as the basis for generating MIDI messages that play music.

### **Tutorial 2: Conclusion**

The basic contents of an incoming MIDI stream can be retrieved using the **midin** object, and a MIDI stream can be sent using the **midout** object. However, in most cases, it is easier to deal with individual message types using their message-specific objects. The **notein**/**noteout** objects provide an easy way to deal with messages without having to decode the raw MIDI input or create raw MIDI output.

## **2.2.4 MIDI Monitor**

Navigate to your desktop ipa1/Max/practical02/ folder and open the patch **MIDIParser0.maxpat**. You should see the patch shown below:



When you play keys on the MIDI keyboard at your workstation you should see numbers appearing in the patch. Note the result within this patch of various actions when playing and manipulating controls at the keyboard. (You may need to perform an Internet search to find out some of this information.)

What is the "status byte" when you play keys on the keyboard: \_\_\_\_\_

What information does this give you? \_\_\_\_\_

What is the 1<sup>st</sup> data byte when you move the "modulation wheel": \_\_\_\_\_

What is the status byte and 1<sup>st</sup> data byte when you move one of the sliders on the keyboard?  
Slider \_\_\_\_\_ Status byte \_\_\_\_\_ 1<sup>st</sup> Data byte \_\_\_\_\_

In this case what is the meaning/use of the 2<sup>nd</sup> Data byte?

---

## 2.2.5 Adapted Tutorials (continued)

The following is adapted from MaxMSP's MIDI Tutorial 2.

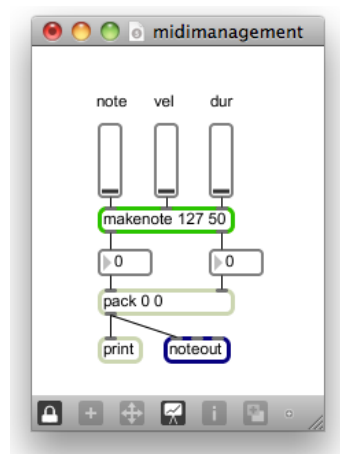
### Tutorial 3: MIDI Note Management

In this tutorial, we will look at several ways to deal with one of the more difficult issues in MIDI note generation and manipulation: dealing with pairs of note-on and note-off messages. We will use **makenote** to generate properly formed pairs of a specific duration.

When working with MIDI synthesizers and samplers, one of the most distracting situations is when a note-on message is sent to a device, but no note-off message is sent. This results in “stuck notes”, where the note is sounded but never turned off. This is never a good result, and may be seen as a “bug” by anyone that uses your patch.

In this tutorial, we will look at how we can maintain an appropriate pairing of note-on and note-off messages.

Navigate to your desktop ipa1/Max/practical02/ folder and open the patch `midimanagement.maxpat`. You should see the patch shown below:



One of the most popular uses of Max is to create generative music using a variety of Max objects. Obviously, this would include the creation of note messages that are played by connected samplers or synthesizers. If you are creating a lot of note messages, maintaining the proper pairing of note-on and note-off messages can require a lot of extra programming.

One of the objects included with Max is the **makenote** object, which will create a pitch/velocity combination of your choosing, then will generate a matching note-off message after a duration you set. The `midimanagement` patch shows the **makenote** object in action – you can select a pitch, velocity and duration using the supplied

**slider** objects; the object will generate a note-on message, followed automatically by a note-off (actually, a note-on with velocity of 0) after the selected duration. The two arguments for **makenote** are the default velocity and duration values, but these are changed when new values are received in the inlets. As with most Max objects, only the left inlet of **makenote** is hot; dragging the leftmost **slider** will cause a ripple of MIDI events, whereas the middle and right **slider** objects only set up the parameters.

Then adjust the pitch, velocity, and duration values. You should hear notes being generated, then turned off, based on the selected duration. We've also sent the note events to a **print** object; if you look in the Max window, you can see how each note-on is followed by corresponding note-off messages, and that the object will correctly queue these messages even with multiple notes sounding at the same time. Using the **makenote** object is the simplest way to generate note playback within Max without having to program the tracking and generation of note-off messages.

### 2.2.6 Exercise: Rising Scale

Using your (or another) solution to the stopwatch exercise generate three rising scales at different rates (one for each of the different time values) using the time values as MIDI note values. You will need to modify the ranges of values used to make the pitch high enough to be audible (keeping the values between 36 and 96 is a good rule of thumb).

Consider ways in which this example could be made more musical.

### 2.2.7 Homework

Make sure that all exercises are complete before next week's MaxMSP session.

### 2.2.8 Finishing up

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

---

## 2.3 Practical 3 – Numerical Operations in MaxMSP

This session is intended to introduce numerical operations in MaxMSP, this includes:

- arithmetic using standard mathematical operators;
- using arithmetic for musical purposes; and
- numerical conversions to binary and hexadecimal.

This session also introduces MaxMSP's "Presentation Mode".

### 2.3.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop `ipa1/Max/practical03` folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.3.2 Introducing numerical operations in MaxMSP

This section will introduce the fundamentals of using arithmetic and mathematical operators in MaxMSP and illustrate how these can be applied to musical situations.

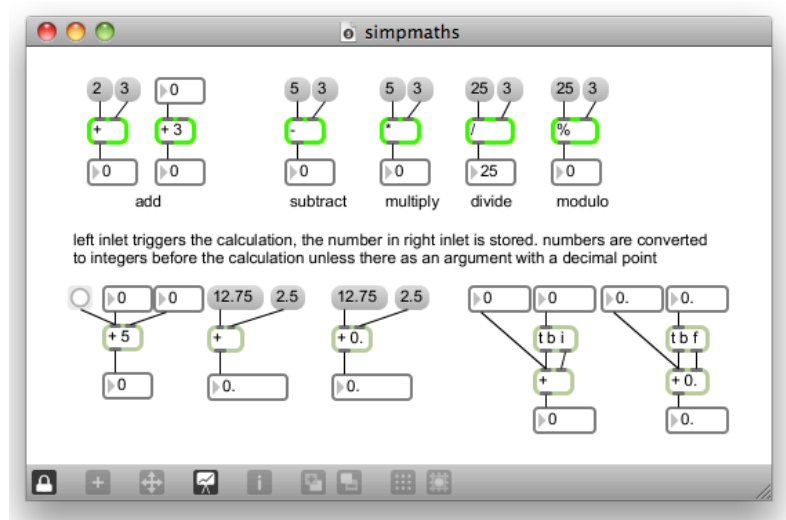
#### Adapted Tutirials

The following tutorials are adapted from the MaxMSP Basic Tutorials 6 & 7.

#### *Tutorial 1: Simple Maths – Performing Calculations*

This tutorial covers the basic mathematical operations that are available with Max objects, and the process of connecting up several objects into a calculation chain. We will also cover the common operations of objects with multiple inlets, and some issues with numeric conversion.

Navigate to your desktop `ipa1/Max/practical03/` folder and open the patch `simplemaths.maxpat`. You should see the patch shown below:



The top row of patches shows examples of the various simple mathematical operations that are available in Max. The left-most patch performs addition using the **+** (*plus*) object. The **+** object, like most maths objects in Max, has two inlets – one for each operand. Numbers coming into the right inlet are stored, and added to numbers coming into the left inlet. If no number has yet come into the right inlet, the right operand is set to the object's argument; if no argument is supplied, the right operand defaults to 0.

When we click on the message box containing the number 2 (on the left), the result (which is displayed in the connected number box) is revealed to be 2 as well; since no number has come into the right inlet, the current stored operand remains 0. Clicking on the message box with the number 3 produces no output, but stores the number 3 as the new right-hand operand. When we click on the 2 again, we see the additive result – 5 – displayed in the number box.

The next patch uses an *argument* to the **+** object that sets an initial default value for the right-hand operand. Now, when we put a number into the **+** object (using the top number box), we see a result without having to send any number into the right inlet. Using default arguments for maths operations is a common setup in many situations where you need to perform a constant mathematical operation (e.g. offsetting or scaling a number) on a stream of input values.

Next are the **-** (minus) and **\*** (multiply) objects, which perform subtraction and multiplication, respectively. Like the **+** object, they have default operands of 0, and accept stored values in their right inlet and produces results based on values coming in the left inlet. The divide (**/**) and modulo (**%**) operations are also similar, but with one difference – in order to prevent unwanted "divide by zero" errors, their default right-hand operands are one (1) rather than zero.

### ***Inlets, Hot and Cold***

As we've seen in the first set of maths objects, some inlets cause an object to output a message, while others cause the object to simply store values. In fact, messages sent to any of the inlets cause these objects to store values; the difference is that values coming in the left-most inlet have an implicit bang message attached to them - this forces the maths object to output the result of the calculation once the value is received and stored. We can see this in action by attaching a bang object to a maths object, and using it to explicitly force output.

The first patch in the second row is a simple addition patch, but with a button connected to the left inlet. Sending numbers into the **+** object performs the expected result: numbers in the right inlet are stored and used as a right-hand operand, while numbers in the left inlet are used as left-hand operands and cause the object to output results. Clicking on the button also produces a result: the addition is performed with the current set of operands, and the result is sent out of the outlet. *This is useful when you want to change the right-hand operand, but want output without having to resend the left-hand operand.* It is also a good example of the bang message performing its "do it!" function.

Inlets that cause an object to output messages are often called hot inlets while inlets that only store information are called cold inlets. It is a common practice, when a cold inlet needs to produce input, to route a bang message (via a button or trigger object) into the left-most inlet to force output.



### ***Type conversions through arguments***

The next two parts of the patcher show what appear to be equivalent operations using the `+` object. Both of them purport to add together the numbers 2.5 (in the right inlet) and 12.75 (in the left). Try them both out (remember we need to click the right-hand message first to set the right operand).

The result of adding 12.75 and 2.5 should be 15.25. But between the two `+` objects, only the right-hand one produces the correct result. The left-hand `+` outputs 14. Why?

Like many programming languages, Max has different types of numeric data, and operations on that data can be done using integer or floating-point mathematics. By default, all of the objects that perform maths operations in Max use integer-based arithmetic. When floating-point numbers are received by a maths object in "integer" mode, they are truncated to integers before being used. Thus, 12.75 becomes 12, and 2.5 is converted to 2 – which adds up to 14. In order to set a maths object to use to floating-point arithmetic, we need to provide an argument that is floating-point. The right-hand patch does just that – it uses a 0. argument to inform the `+` object that we want to perform floating-point arithmetic, and the addition results in the 15.25 value that we were expecting.

### ***Forcing a calculation***

The final two patches in the second row show a simple mechanism for triggering output from the `+` object, regardless of which inlet receives messages. The left inlet is fed directly by a number box; since left inlets are already hot, they need no further attention. However, in order for the right inlet to produce output, we have to send a bang into the left inlet after the right inlet has received input. This is a perfect application for the `trigger` object, which is used to manage this function.

In both cases, the `trigger` object (abbreviated to `t`) has two outlets (defined by its arguments): the right outlet is type-specific (`i` for integer, `f` for float), while the left (`b`) sends out a bang. When a number is entered into the object, `trigger` processes the arguments in a right-to-left order, first sending the numeric value (integer or floating-point) into the right inlet of the `+` object. Then, a bang is sent to the left inlet of `+`, forcing calculation and output to the number box below. In this way, we can force both inlets of the `+` object to behave as though they are "hot", and receive output regardless of which inlet is sent messages.

### ***Cascading maths objects***

All of the above examples have been pretty simple – they've been simple maths operations with a simple result. However, you can chain several of these operations together to create more complex equation.

For example, let's say we wanted to make a Max patcher for converting temperatures: specifically, converting Fahrenheit to Celsius. The formula for this conversion is:

$$^{\circ}\text{C} = (^{\circ}\text{F} - 32) * (5/9)$$

From an object standpoint, it might be easier to simplify this to:

$$^{\circ}\text{C} = (((^{\circ}\text{F} - 32) * 5) / 9)$$

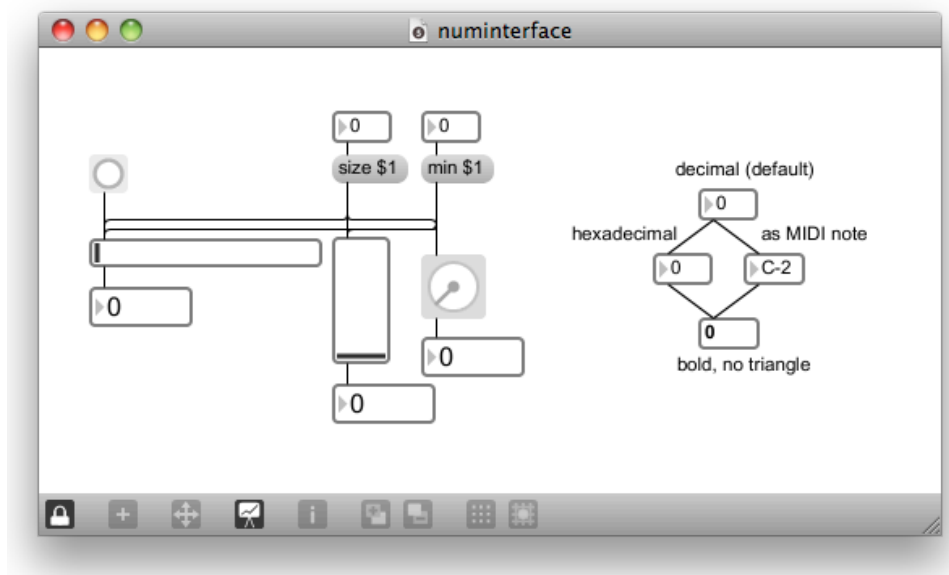
We can produce this output by using the -, \*, and / objects chained together. Unlock the patch, and begin by using a number box for the Fahrenheit (°F) input. Connect this to a - object with an argument of 32 as the default right-hand operand. Connect the output of the - object to a \* object with 5 as an argument, and connect it to a / object with 9 as the argument. Finally, connect the output of the / object to another number box to see the result of the Fahrenheit to Celsius conversion.

## ***Tutorial 2: Numerical User Interface Objects***

In this tutorial, we will work with several user interface elements that provide for numerical display and editing. We will use both messages and the object inspector to change the appearance and behavior of objects in order to make them work as we wish.

Numerical display and editing is at the heart of Max patching. Whether data comes from sensors, MIDI devices or just from moving on-screen controls, numerical data is easy to test, manipulate and use for program control.

Navigate to your desktop ipa1/Max/practical03/ folder and open the patch numinterface.maxpat. You should see the patch shown below:



## ***Numeric UI Objects***

At the left side of the patch is a set of interconnected message boxes, sliders, knobs and number boxes. The basic "slider" object in Max is called a `slider`, and can be oriented horizontally and vertically, depending on what we need for our patcher interface. The "knob" is called a `dial` object.

Click and drag on the "thumb" of the `slider` objects to see the results displayed in the connected number boxes. The output of a `slider` is an integer number based on the range of the slider. Both the vertical and horizontal slider perform identically.

Click on the button just above the horizontal slider – it doesn't seem to perform any function. However, if we change the number box connected to the slider, then click the button again, we will see that when a slider receives a bang, it responds by

outputting its most recent value. Thus, a slider acts like a number storage tool as well as a numerical editor.

Finally, let's unlock the patch. If we put the mouse pointer near the bottom-right corner of a slider or dial, we see a small resize box appear, and the cursor changes to reflect that we can resize the objects. When we click and drag the resize box, the size of the object changes, but the function of the object stays the same. This allows us to change the layout of our patch for convenience without changing the way our user interface elements operate.

### ***Changing UI object behavior***

If you haven't adjusted the size in min parameters, all three of our user interface objects will share the same output range: 0 through 127. Obviously, this is not the behavior we would always (or even normally) want. By default, these objects output numbers in the range of 0 to 127 (which mimics the MIDI number range); this range can easily be changed using messages to the object.

Above the objects are some message boxes formatting messages based on numbers coming in from attached number boxes. This is a mechanism to send commands to another object; in this case, we will use them to change the behavior of the slider and dial objects. If we change the first number box (connected to the size \$1 message) to 50 and then move the sliders or knob, we see that the range has changed: you can now produce output from 0 through 49. Why not 50? It is because the size message determines the number of discrete steps for the object. If the starting step of 0 is included, 50 steps will give you a maximum value of 49.

Next, let's adjust the number box which triggers the min message. If we change this to 30, the slider objects and the dial now output messages in the range of 30 to 79. Using the min and size messages in tandem allow us to set the objects to output whatever range of values we need for our patching.

### ***number box options and the Inspector***

On the right-hand side of the patch is a diamond array of number boxes, each one providing a different *display format*. The top-most number box is familiar to us – a decimal format number, with a triangle that helps us recognize this as an editable number box. Connected to this are two number boxes with different display formats: the left box displays information in hexadecimal (base-16), while the right box displays the information as if it were an incoming MIDI value.

The bottom number box shows a change in display characteristics: the number is in a bold font, and there is no triangle on the left of the box. This may be useful if you want to see more of the number in a limited space, or if you want to imply that the value shouldn't be edited.

All of these variations on the number box are based on changes in the attributes of the object. Attributes are changed using the object inspector, a window containing all of an object's attributes, and an interface for editing them. Unlock the patch and hover the mouse over the middle-left of one of the number boxes – a small blue circle with the letter "i" will appear. Click on this circle to invoke the object inspector.

The initial display of the inspector shows all of the available attributes, colour-coded by type. There is also a set of tabs across the top of the inspector that allow us to

limit the attributes that we see. The "Display Format" attribute has a discrete number of values; you can see the available formats by clicking on the current value. The options are "Decimal", "Hex", "Octal", "Binary", "MIDI", and "MIDI (C4)". These are all variations on decimal display that may be useful, depending on the values you are entering or displaying in a number box.

When we click on the "Appearance" tab, we only see attributes that affect the appearance of the number box. Options include "Draw Triangle", something we saw disabled in the bottom number box in our patcher. Turn this on and off in the inspector, and notice that there is an immediate change in the object; changes to inspector values cause instant changes to the object.

The names used by the inspector do not necessarily match the messages used to control the object. How do we know what message might match a particular attribute? The easiest way to create a matching message is to click on the name of an attribute and drag it into an unlocked patcher. The name will be transformed into a message box formatted with the proper message, offering a shortcut when you want programmatic control of attributes.

### ***Using the inspector***

Let's unlock the patcher and create our own small user interface. Open the object palette and select a slider, a dial and a number box. Resize them to be different sizes, and move them around to make an attractive layout. Note how the slider switches from a vertical to horizontal orientation depending on the aspect ratio (height versus width). Connect the objects to see how they interact, and use the objects as both display and editing tools. Then, for each object, open the inspector and make some changes in appearance. Change the colors, and (if you like) change their behavior. When you are done, lock the patch and make sure the object look and act as you expected.

## **2.3.3 The expr object**

In Tutorial 1 above you learned about simple numerical processes in MaxMSP.

Navigate to your desktop `ipa1/Max/practical03/` folder and open the patch `mtof-expr.maxpat`.

This patch illustrates MIDI note number to frequency conversion (which should be familiar to you from the C practicals this week). The `expr` object Help file demonstrates a slightly different, although equivalent, formula. (There is also a dedicated object for this task in MaxMSP called `mtof`.)

The `expr` object allows you to form complex expressions without using multiple `+`, `-`, `*` and `/` objects. Read the `expr` Help and Reference files and try to implement the  $F^\circ$  to  $C^\circ$  conversion (from Max Help Basic Tutorial 6) using a single `expr` object.

## **2.3.4 Making musical chord shapes**

Now open the patcher `MIDIchords.maxpat`.

This example shows how you can play back chords on a MIDI synthesiser device in response to single notes played in from a MIDI keyboard input device. When you

play on the MIDI keyboard you should be able to hear the original note, one **octave** down (**transposed** 12 **semitones** lower) from the original note and one octave up (12 semitone higher) from the original note, all played simultaneously. The values of  $\pm 12$  are used to offset the original note to achieve these octave shifts in order that you will hear the same note in different octaves. Thus if you play a C, the other two notes you hear will be C's in other octaves. If you play chords on the MIDI keyboard you will hear these three notes for each note in your chord.

Modify this example — by changing the **arguments** to the '+' objects — to produce the following **triad** chord shapes, writing the values in the spaces provided:

<b>Chord shape</b>	<b>Argument to 1<sup>st</sup> '+'</b>	<b>Argument to 2<sup>nd</sup> '+'</b>
Major triad		
Minor triad		


Now modify the example to produce the following chord shapes. (Note that you will need to add at least one object to do this.) Write the values used and which object(s) you added to achieve this:

<b>Chord shape</b>	<b>Values used</b>
Major 7th	
Minor 7th	
Objects added:	

## 2.3.5 Arpeggiator (and presentation mode)

Now open the patcher `MIDIarp.maxpat`.

This example builds on the “MIDIchords” example in section 2.3.4 above. However, in this example the notes are played back in an **arpeggio** (i.e., the notes of a chord are played in sequence rather than simultaneously).

MaxMSP enables a patcher's graphical user interface to be designed relatively independently from its functional elements. (This feature is new in MaxMSP version 5, which you are using.) The patcher mode you have been using so far is known as **patching mode**. Another mode, specifically designed for making user interfaces, is known as **presentation mode**. To switch to presentation mode click on the  button at the bottom of the patcher window. You will see that some objects disappear, some move and some even resize. Try switching back and forth between presentation and patching mode a number of times by clicking on the same button. (You can also switch modes by using the menu item **View→Presentation**.) Note that both patching and presentation mode have locked and unlocked modes (so you could think of there being four patcher modes in total).

In either locked patching or locked presentation mode you will be able to choose two note transpositions and play a single note on the MIDI keyboard. While you are holding the note, MaxMSP should generate an arpeggio using the values provided (thus playing an arpeggiated chord).

Switch to patching mode and unlock the patch. You'll notice a pink glow around certain objects, these are objects which have been added to the presentation mode of the patcher. You should recognise the `toggle`, `metro`, `counter` and `select` objects from the Stopwatch exercise from the last two weeks of practicals. Here they form the basis of a simple sequencer. You will also notice an object you may not have encountered before, `loadmess`. Read about `loadmess` in its Help and Reference documents. You can see that it is used here to initialise the various controls in the patch, including switching the `metro` on, when the patch loads.

Finally, you may notice that in this patch the '+' objects have been replaced with the `expr` object performing the same purpose (using the expression "`$i1 + $i2`". This is just to remind you of the existence of the `expr` object, '+' objects could have been used as they were in the "MIDIchords" example to make the patch simpler.

Modify the patch using the values you deduced for the triad chords in section 2.3.4 above, to produce arpeggios of either of these chords (major or minor).

### Extending the arpeggiator

You are to extend the `MIDIarp.maxpat` patch so that it plays a four-note arpeggio. Check that this works before proceeding.

Once you have completed this you should have an additional number box to set the transposition of this new note. If you switch to presentation mode your new number box may or may not be there (depending on how you performed the task). If the number box is not in presentation mode you need to switch back to patching mode, select it and choose menu item **Object→Add to Presentation**. Once you switch back again to presentation mode your number box should now remain visible. While in presentation (unlocked) mode you can move the number box into the most suitable position (and resize if you wish). You may want to add a `comment` object to label the number box, matching the others. Observe again that when you switch back and forth between presentation and patching mode the objects recall their stored positions in each mode.

### 2.3.6 Additional exercises and/or homework

Complete the tasks in the earlier sections if you have not already done so. Then open the patch `MIDIpars0.maxpat`.

In fact, this patch opens automatically in presentation mode. Switch to patching mode and enlarge the window so that you can see the whole of the inner workings of the patch. Don't worry about how the upper part of the patch works, we will be revisiting that another time. You are concerned, in this exercise, with the lower part of the patch only.

There are two comments labelled "binary" and "hex" (which have a green and blue background respectively). You should add appropriate number boxes to display the MIDI status byte and the two data bytes in *binary* and *hexadecimal* format. You should try to work out how to hide and show the number boxes for the 2<sup>nd</sup> data byte when there is only one (this is the case for MIDI channel pressure messages and MIDI program change messages). This is not critical to the operation of the patch, however.

Finally, ensure your new number boxes appear correctly in presentation mode.

### 2.3.7 Finishing up

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

You should also tidy up your desktop and organise the files from this practical in your home folder (e.g., into your documents folder: ~/Documents).

### 2.3.8 Important material

Through following this practical session you should be familiar with the following concepts and operations in MaxMSP before the next session:

- presentation mode and patching mode;
- performing numerical operations using '+', '-', '\*', '/' and expr;
- performing simple processing on MIDI data for musical purposes; and
- displaying binary and hexadecimal data.





---

## 2.4 Practical 4 – More numerical operations in MaxMSP

This session continues to explore numerical operations in MaxMSP, for technical and musical applications. This includes:

- using the ‘%’ (modulo) object;
- decoding MIDI status byte information; and
- decoding octave and pitch class information from MIDI note numbers.

This session also illustrates the use of the umenu object.

### 2.4.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop ipa1/Max/practical04 folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.4.2 The ‘%’ (modulo) object

This section illustrates some uses of the modulo operator, implemented as the ‘%’ object in MaxMSP.

#### Modulo-based stopwatch

Navigate to your desktop ipa1/Max/practical04/ folder and open the patch moduloStopwatch.maxpat.

This shows a method of solving the stopwatch exercise, from the MaxMSP Practical 1 session, using the ‘%’ and ‘/’ objects. This method uses only a single metro (which we have already discussed as being a good idea in order to ensure timing operations are kept in synchronisation). What is most important is that this method also uses only one counter (rather than three counters as we examined in earlier weeks). This single counter object simply counts the total number of 1/10<sup>th</sup> seconds since the stopwatch was started. After 1 second this will be 10, after 10 seconds it will be 100 and so on, until the counter is reset by the user hitting the ‘reset’ button. Note that there is only one counter to reset in this patch.

The total seconds  $s_{total}$  can be calculated by dividing the total 1/10<sup>th</sup> seconds  $t_{total}$  by 10, i.e.:

$$s_{total} = \frac{t_{total}}{10} \quad (1)$$

To calculate the 1/10<sup>th</sup> seconds to **display**  $t_{display}$  we need the **remainder** from this calculation. To do this we can use modulo, and in this case we need to use “modulo 10”, i.e.:

$$t_{display} = t_{total} \bmod 10 \quad (2)$$

In MaxMSP's `expr` object you could use the following expression:



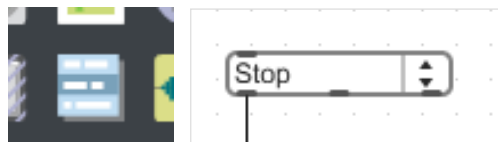
**Figure 1: mod 10 using `expr`**

The values here shown here illustrate that “23 modulo 10 is equal to 3” or “23 divided by 10 is equal to: 2 remainder 3”. The ‘%’ object is equivalent<sup>1</sup>.

Using this same technique we can obtain the seconds and minutes to display, thus deriving all of the time values from the original counter counting total number of 1/10<sup>th</sup> second.


This patch also uses the `umenu` object to start and stop the `metro` object rather than a toggle. You first encountered the `umenu` object in Max Help MIDI Tutorial 1 (“Basic MIDI”) where it was used to select from the list of available MIDI outputs on the computer you are using.

The `umenu` object is a graphical object and is available on the object palette (when you double-click in an unlocked patcher), its icon and appearance in a patcher are shown in Figure 2 below.



**Figure 2: the `umenu` object icon (left) and its appearance in a patcher (right)**

It is important to note that the `umenu` object outputs the index (numbered from zero) of the item selected, out of its left outlet when the item is selected. In this example there are two menu items, “Stop” and “Start” with indices 0 and 1 respectively. Since these values are the same as those transmitted by the `toggle` object, this `umenu` object can be used to turn the `metro` off and on.

Unlock the patch and open this `umenu`’s Inspector (by selecting it and choosing menu item **Object→Inspector**). Scroll down to the **setting** “Menu Items” (it is in the **settings group** “Items” so you can click on the “Items” tab to narrow down the number of listed settings). Now click on the small “edit” button (  ) on the “Menu Items” row. This pops up a floating text editing window. Menu items are entered into this window separated by commas (therefore items can not contain commas). Try changing the menu items to: “Off” and “On” (instead of “Stop and “Start”). These will

<sup>1</sup> This is not strictly true: there seems to be a minor difference. In the C language the modulo (%) operator is not valid for operating on floats. In MaxMSP the ‘%’ object can operate on floats giving the expected result as a “remainder” operator (e.g., 7.5 % 3.5 = 0.5). In the `expr` object the ‘%’ operator behaves more like the C language by truncating any floats involved in the calculation to ints (e.g., 7.5 % 3.5 = 1 being the same as 7 % 3 = 1).

still have indices 0 and 1 so will still control the metro in exactly the same way as before.

### Octaves and pitch classes using modulo

MIDI note numbers are commonly used to represent pitch information in computing and audio applications. These have the range 0 to 127 where note 60 is “Middle C”. Of course there is a note C in each octave, which will always be a multiple of 12 semitones away from note 60 (since there are 12 semitones in an octave). Thus note 48 and note 72 are also C’s. Note 61 is a C $\sharp$ , therefore notes 49 and 73 are also C $\sharp$ ’s. The name of a note, independent of octave information, can be referred to as its ***pitch class***. Thus middle C has the pitch class C (“middle” refers to its specific octave position). There are twelve pitch classes, which can be referred to by name (C, C $\sharp$ , D, etc.) or by number, where C is 0, C $\sharp$  is 1, D is 2, and so on. See Figure 3 below.

<i>Pitch class name</i>	<i>Alternative names<sup>2</sup></i>	<i>Pitch class number</i>
<b>C</b>	B $\sharp$	<b>0</b>
<b>C<math>\sharp</math></b>	D $\flat$	<b>1</b>
<b>D</b>	C $\times$	<b>2</b>
<b>D<math>\sharp</math></b>	E $\flat$	<b>3</b>
<b>E</b>	F $\flat$	<b>4</b>
<b>F</b>	E $\sharp$	<b>5</b>
<b>F<math>\sharp</math></b>	G $\flat$	<b>6</b>
<b>G</b>	F $\times$	<b>7</b>
<b>G<math>\sharp</math></b>	A $\flat$	<b>8</b>
<b>A</b>	G $\times$	<b>9</b>
<b>A<math>\sharp</math></b>	B $\flat$	<b>10</b>
<b>B</b>	C $\flat$	<b>11</b>

Figure 3: Pitch class names and numbers

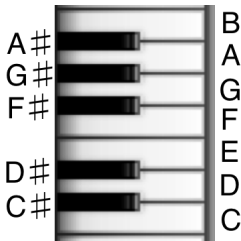
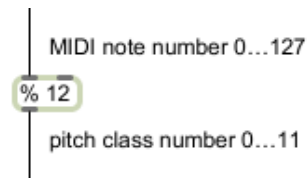


Figure 4: Musical keyboard showing pitch class

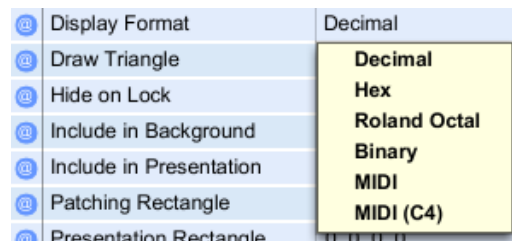
Using a technique similar to that in the ‘modulo\_stopwatch’ example, we can obtain the pitch class number from a MIDI note number using the ‘%’ (modulo) object in MaxMSP as shown in Figure 5 below.

<sup>2</sup> This is not an exhaustive list of the alternative names. The ‘ $\times$ ’ symbol means ***double sharp*** i.e., up by two semitones. The ‘ $\flat\flat$ ’ symbol means ***double flat*** i.e., down by two semitones. Although in some contexts (for example) D $\sharp$  and E $\flat$  would be considered slightly different, in terms of pitch class they are considered identical.



**Figure 5: Pitch class from MIDI note number**

As a first step in obtaining the octave number from a MIDI note number we can use the '/' (division) to divide by 12. MIDI note names give the name of middle C as "C3" (where 3 is the octave number). Although a broader, international standard of note naming gives middle C as "C4" (octave number 4). The display style for MaxMSP's number boxes differentiates between these as "MIDI" and "MIDI (C4)" respectively in its Inspector, as shown in Figure 6 below.



**Figure 6: Number box display format options**

Of course  $60 \div 12 = 5$ , so to obtain either 3 or 4 (as the octave number) to follow either one of these two standards we would need to subtract 2 or 1 respectively.

## Exercise — pitch class and octave calculation

Make a patch which takes a MIDI note in from the MIDI keyboard and displays its:

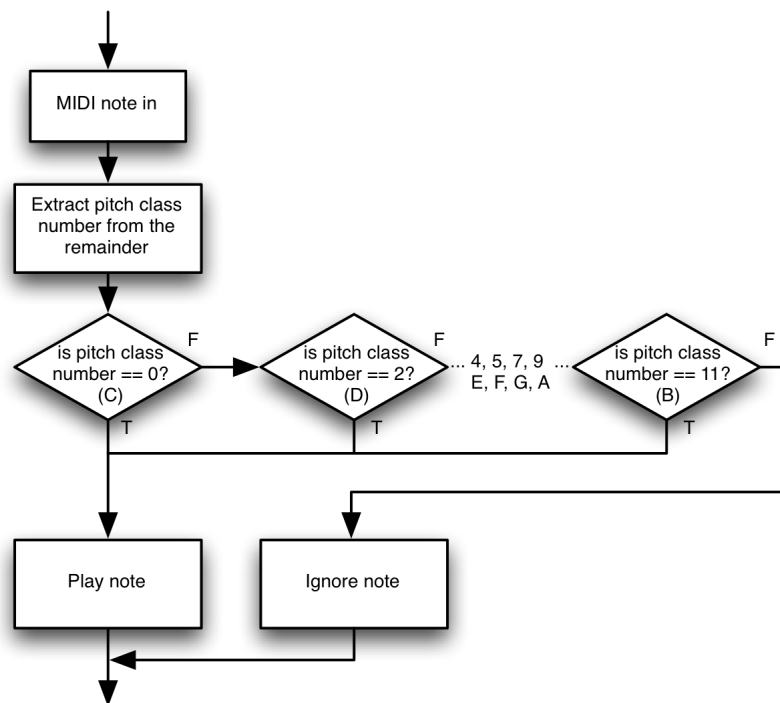
- octave (using either of the standards discussed above)
- pitch class number; and
- pitch class name.

Use a umenu object to display the pitch class name and number box objects to display the octave and pitch class numbers (in decimal format).

## Filtering notes according to pitch class

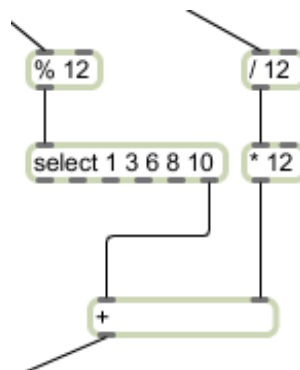
Obtaining the pitch class of a note number may be required when performing some kind of analysis on note data. This is useful when generating pitch material using randomness or some algorithmic method (since the generated data can, for example, be "rounded" to the nearest note in a desired musical scale). Figure 7 below illustrates a flow chart that could be used in designing a program/patch to filter out all the "black notes" (i.e., C#, D#, F#, G# and A#) played on a MIDI keyboard, allowing only the "white notes" to be played on a synthesiser.

Open the patcher `noteProcess.maxpat` into MaxMSP.



**Figure 7: flowchart for selecting to play only "white notes" from a MIDI keyboard input**

This patch uses the `select` object to filter out all of the unwanted notes. This is achieved by entering the unwanted values as arguments to the `select` object and using its right ("reject") outlet to allow the other values to pass through. Although this outlet is called "reject" we are in fact using the outlet to *accept* values **not** listed. See Figure 8 below.



**Figure 8: using the `select` object's reject outlet**

Switch on the toggle to the top-right of the patch. This starts the metro which is in turn connected to a random object which generates random values between 0 and 59. These values all have 36 added to them to transpose them into a useful range as MIDI note numbers. There should be no sound at this stage.

Now select the item "no processing" from the `umenu`. This opens the left outlet of the gate object allowing values from the random number generator to pass through to the `makenote` and `noteout` objects. The resulting sound will sound quite atonal although it does have a regular rhythm.

Now select the item "'white notes' only - omit others" from the `umenu`. This opens the third outlet of the gate. You will hear the rhythm change as certain notes are omitted. These are notes which have failed the test when their pitch class is checked using

one of the select objects in the patch. Notes which succeed the test and pass through the select object are recombined with the octave information (by multiplying the integer octave number by 12 and adding this to the pitch class number).

Now select item “white notes' only - repeat others” from the umenu. Use the space below to describe more accurately what happens when this menu item is selected:

### Exercise — “black notes” only

Modify the “note\_process.maxpat” patch to add two further options:

- 'black notes' only - repeat others; and
- 'black notes' only - omit others.

### Random chords

Open the patcher chord\_process.maxpat.

This example includes a new section to the patch illustrated by Figure 9 below (bounded by a grey panel object in the patcher).

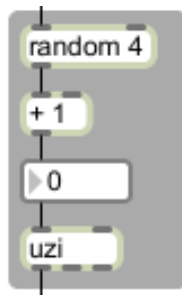


Figure 9: random chords

Turn on the toggle and listen to the result. Examine the patch and describe what is happening and how this patch works below:

## MIDI status byte information using modulo

Open the patcher from `MIDIparser1.maxpat`. Open this patcher into MaxMSP. Note that this is similar to the patch used in earlier practical's but it is a new version which includes the solution to Section 2.3.6 "Additional exercises and/or homework" from last week's practical. The MIDI status and data bytes are displayed in decimal, binary and hexadecimal format.

Note that the new item "MIDI Channel" and the large number box does not appear to do anything. It will be your task shortly to configure this to work correctly.

The type of MIDI message (e.g., whether the message is a **note on**, or a **control change** message) is encoded in the first byte of a MIDI message called the status byte. Status bytes are those where the value are 128 or greater. Values between 0 and 127 are data bytes. The type of message encoded in the status byte can be decoded by dividing its value by 16 as shown in Figure 10 below.

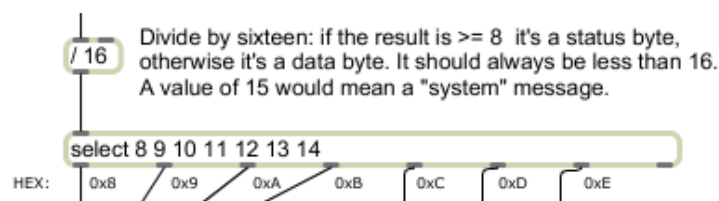


Figure 10: dividing by 16 to obtain the status byte type

One way to determine if a MIDI byte is a status byte is to test the result of this division. If it is greater than or equal to 8 it is a status byte. This value is in fact equal to the **upper nibble** (i.e., the most significant four bits) of the status byte. Figure 11 below lists these values, the corresponding MIDI message type and the number of data bytes for that type of message.

Decimal	Hexadecimal	Message type	Data bytes
8	8	Note off	2
9	9	Note on	2
10	A	Polyphonic aftertouch	2
11	B	Control change	2
12	C	Program change	1
13	D	Channel aftertouch	1
14	E	Pitch bend	2
15	F	System message	varies

Figure 11: MIDI message types — upper nibble

If the upper nibble of the status byte is between 8 and 14 (8 to E in hex) the message is a **channel voice message**. A value of 15 (or F in hex) signifies a **system message** (which includes things like transport controls and timecode messages among other things). If a message is a channel voice message, the **lower nibble** of the status byte encodes the **MIDI channel** of the message (this is not the case for system messages). Figure 12 below lists the possible values for this lower nibble and to which MIDI channel these values correspond. Note that the values are offset by 1 (e.g., a value of 7 indicates the message is on MIDI channel 8).

<i>Lower nibble (decimal)</i>	<i>Lower nibble (binary)</i>	<i>MIDI channel</i>
0	0000	1
1	0001	2
2	0010	3
3	0011	4
4	0100	5
5	0101	6
6	0110	7
7	0111	8
8	1000	9
9	1001	10
10	1010	11
11	1011	12
12	1100	13
13	1101	14
14	1110	15
15	1111	16

**Figure 12: Channel voice message status byte lower nibble**

The “MIDIparser1” patch determines how many bytes to expect for each type of message according to the value of this status byte upper nibble. The z1 object groups the bytes into **lists** according the number of bytes to expect.

### **Exercise — decoding MIDI channel from status byte**

Modify the “MIDIparser1” patch to correctly decode the MIDI channel

## **2.4.3 Additional exercises and/or homework**

Complete the tasks in section 2.4.2 if you have not already done so.

As an additional exercise combine your attempt at the ‘Exercise — “black notes” only’ (p46) with the ‘Random chords’ (p46) such that chords are played back so that the note filtering can be selected using the umenu as before.

## **2.4.4 Finishing up**

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

You should also tidy up your desktop and organise the files from this practical in your desktop ipa1/ folder.



### 2.4.5 Important material

Through following this practical session you should be familiar with the following concepts and operations in MaxMSP before the next session.

- Using the '/' and '%' objects to process:
  - timing information;
  - musical pitch information; and
  - information encoded in MIDI status bytes.
- Building menus using the umenu object.

### 2.4.6 Further reading

*Internals and Interconnections: MIDI in:*

Roads, C. 1996. *The computer music tutorial*. Cambridge MA: The MIT Press. pp969–1016. You should read pp983-93 especially.



---

## 2.5 Practical 5 – Sequences and External Control

This practical session builds upon previous sessions involving numerical operations and MIDI input/output. In particular, by the end of this session you should be able to:

- use a slider on the Axiom keyboard to control a GUI<sup>3</sup> slider on the screen;
- use a button on the Axiom keyboard to control GUI items on-screen such as buttons and toggles;
- apply this knowledge to other keyboards / devices which implement the MIDI standard;
- construct simple drum sequencers; and
- employ random number techniques to generate simple musical patterns.

### 2.5.1 Getting started

This practical is designed to work with the Axiom keyboards. Ideally you should reset the Axiom before you start: switch it off, hold down the '+' and '-' buttons on the upper left hand side and switch it back on (holding the buttons until the display appears as normal).

1. Log in to the Macintosh
2. Navigate to your desktop ipa1/Max/practical05 folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.5.2 External control

So far we have been sending and receiving MIDI note on and off messages using MaxMSP. To do this we have been using the `notein` object for receiving note on/off messages (from the MIDI keyboard) and the `noteout` object for sending note on/off messages (triggering sounds on the synthesiser built-in to the computer). If you are not familiar with `notein` and `noteout` you should review their Max Help files **now**.

#### The `midiparse` object

We have seen the `midiin` object in earlier practicals (in the `MIDIparser0` and `MIDIparser1` patches which we used to display the raw MIDI data in various formats such as binary and hexadecimal). The `midiin` object simply sends integers into our patch as they arrive at the MIDI port. (The port can be selected by double-clicking on any of the MIDI objects in a locked patch.) All messages arriving at the selected MIDI port are encoded into this stream of integers. For these numbers to be useful they need to be decoded or ***parsed***. Conversely the `notein` object sends out data only when the MIDI message received by its chosen port is a MIDI note on or off message.

---

<sup>3</sup> Graphical User Interface

Navigate to your desktop ipa1/Max/practical05/ folder and open the patch otherinput.maxpat.

This patch uses the midiparse object, which is designed to receive raw MIDI data from a midiin object and to output the various types of MIDI channel voice message from its various outlets. Play notes on the MIDI keyboard and you should see the note data arriving at the message box labelled “note on/off” as shown in Figure 13 below. (Note that you should not hear any sound, this is normal — nothing is connected to transmit these MIDI messages to a synthesiser!)

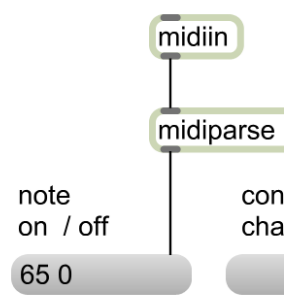


Figure 13: obtaining note data using midiparse

Note the appearance of the message box (filled grey) compared to an object box (border-only). Notice also that the patchcord from midiparse’s left outlet is connected to the message box’s **right** inlet. This changes the contents of the message box to the message received at the right inlet. In this way we can use a message box to display any data within a patch.

MaxMSP data type	Examples				
int	<table><tr><td>1</td></tr><tr><td>-2</td></tr><tr><td>0</td></tr><tr><td>999</td></tr></table>	1	-2	0	999
1					
-2					
0					
999					
float	<table><tr><td>0.0</td></tr><tr><td>3.0</td></tr><tr><td>-1.0</td></tr><tr><td>6.283186</td></tr></table>	0.0	3.0	-1.0	6.283186
0.0					
3.0					
-1.0					
6.283186					
list	<table><tr><td>65 0</td></tr><tr><td>0 1 2 3 4 5</td></tr><tr><td>0.0 3.14 6.28</td></tr><tr><td>1 0.9</td></tr></table>	65 0	0 1 2 3 4 5	0.0 3.14 6.28	1 0.9
65 0					
0 1 2 3 4 5					
0.0 3.14 6.28					
1 0.9					

Figure 14: Data types in MaxMSP

The data displayed in the message box is a pair of integers separated by a space. Grouping data into pairs or larger groupings in MaxMSP is known as a **list**. **Floats** and **ints** can be mixed in a list (although you are unlikely to need this kind of list at the

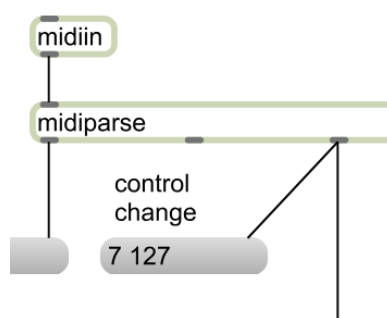
moment). Figure 14 above illustrates some of the MaxMSP data types you have encountered so far.

Thus there is a subtle difference between obtaining note data from `notein` and obtaining note data from `midiin` → `midiparse`. The `notein` object sends the pitch, velocity and MIDI channel values from separate outlets “simultaneously”. (However, recalling the right-to-left rule, MIDI channel is sent first from the right outlet, then velocity from the middle outlet and finally pitch from the left outlet.) The `midiparse` object receives raw MIDI note data from the `midiin` object and sends the MIDI channel from its right outlet and then the pitch/velocity pair as a **list** from its left outlet.

In addition to starting and stopping “notes” MIDI implements messages which are designed to control the way sounds play back on a particular MIDI channel. The simplest example is the **volume** control, which is intended to control how loud sounds play back on a particular channel. Most of these controls are known as **control change messages** (often referred to as **controllers**). These controllers are numbered according to the MIDI standard. For example, volume control is controller 7. The `midiparse` object sends any control change messages received via its third outlet. Again this is formatted as a list in the following format:

*<controller number> <controller value>*

For example, a maximum volume control message would be displayed in our patch as “7 127” and a minimum volume control message would be displayed as “7 0”. If you move slider D17 on the Axiom keyboard you should see these volume control messages appearing in the message box labelled “control change” as shown in Figure 15 below.



**Figure 15: control change messages via `midiparse`**

A simple way to decode and route control change messages in MaxMSP is to use the `route` object in conjunction with the `midiin` and `midiparse` objects. The `route` object accepts **lists** arriving at its inlet and examines the first element. This is compared to any arguments typed into `route`, if this first **list** element matches any of the `route` arguments, the **remainder** of the **list** (i.e., everything **except** the first element) is sent out of the corresponding outlet. (Such that `route`’s first argument corresponds to the first outlet, the second argument to the second outlet and so on.) The `route` object is therefore similar to the `select` object although the `select` object sends only **bangs**. Similar to `select`, `route` always has an additional outlet (i.e., one more than the number of arguments) used as a “reject” outlet. Any **lists** received where the first element **does not** match any of the arguments are sent out of here unchanged. You could test this by connecting a `print` object to the right outlet of the `route` object and

move various sliders/dials on the keyboard. Any not matching controllers 74, 71, 91, or 93 will be dumped into the Max window.

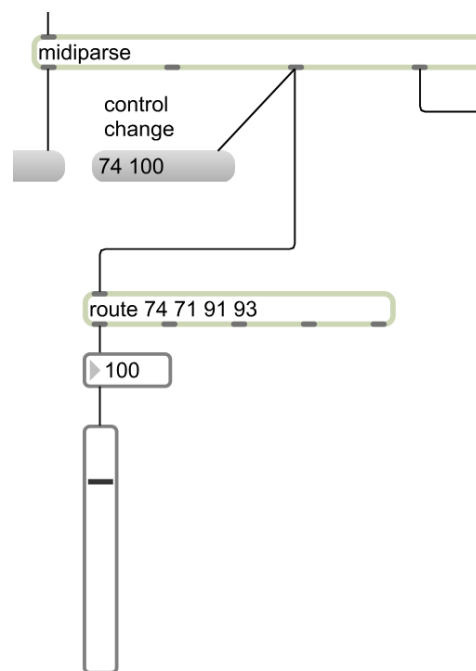


Figure 16: Using route to decode control change messages

### Exercise — adding more sliders

Extend the 'otherinput' patch adding eight more sliders such that your nine sliders in total respond to the nine sliders labelled D9–D17 on the Axiom. The controller numbers for the first four sliders are already provided for you in the route object, you should already know the controller number for slider D17 and you will need to work out the controller numbers for the other four sliders.

### Program change messages

MIDI program change messages are usually used to change the “instrument” or sound type being used by a particular MIDI channel. The buttons beneath sliders D9–D17 on the Axiom transmit program change messages. These buttons are labelled “Zone 1–4” and “Group A–D”, and transmit program change messages 0–3 and 4–7 respectively (the button under slider D17 transmits program change 8). In MaxMSP we can use these messages for anything we like, in the 'otherinput' patch the Axiom’s Zone 1 button should cause a button to flash and the toggle to turn on and off for each button press. Note that this is due to the behaviour of the toggle, it switches to the opposite state each time it receives a **bang**. This section of the patch is shown in Figure 17 below.

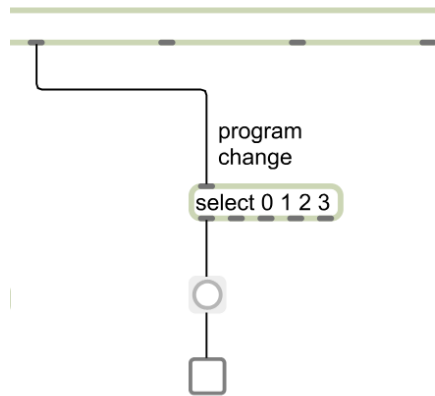


Figure 17: program change messages

### Exercise — adding more buttons

Extend the patch further adding more buttons and toggles such that each of the nine buttons on the Axiom described above can turn a toggle on and off.

### 2.5.3 Simple sequencing

Now open the patcher `simpledrum.maxpat` in MaxMSP.

Most of this patch should be familiar to you by now, if you do not recognise any of the objects you should review their Max Help files **now**.

The metro and counter cause a regular count between 0–7. This is in turn passed to a select object which decodes this number to one of its outlets. The gate and toggle objects then govern whether the **bang** messages are passed to the int object.

In particular you should be aware of the function of the int object. This is a very simple object that simply stores an integer which can be “**banged**” out at some point in the future. It is similar to declaring an int variable in C. Here the int object stores a MIDI note number which is released to the makenote and noteout objects to play a drum sound.

Thus for each step of the sequence the state of the corresponding toggle object controls whether the sound will play. If the toggle is on then the gate is open, the **int** is released and the sound plays. If the toggle is off then the gate is closed and **bang** messages from select go no further than this.

Note that there is argument “10” to the noteout object setting its MIDI channel. MIDI channel 10 is commonly used for drum or percussion sounds and this is the default on the built-in synthesiser.

If you wish, you can change the type of drum kit using program change messages using a pgmout object. You can try this as an additional exercise, especially if you don’t like the default drum sound! In this case you will find program change numbers 1, 17, 25, 26, 41 and 49 useful (the others are equivalent to program change 1 on this basic synthesiser).

You can experiment with different drum sounds by changing the number box connected to the right inlet of the int. (And to make this change permanent in the patch you can change the argument to the loadmess object above it.) Make a note of some of the drum sounds and their corresponding note numbers in the space overleaf.



## Exercise — extending the sequencer

Extend the drum sequencer such that it has 16 steps. Then add at least one additional drum sounds so that you have 16 toggles for each instrument. Add the appropriate objects to presentation mode, your final patch in presentation mode should look something like Figure 18 below. (Assuming you added two additional instruments.)

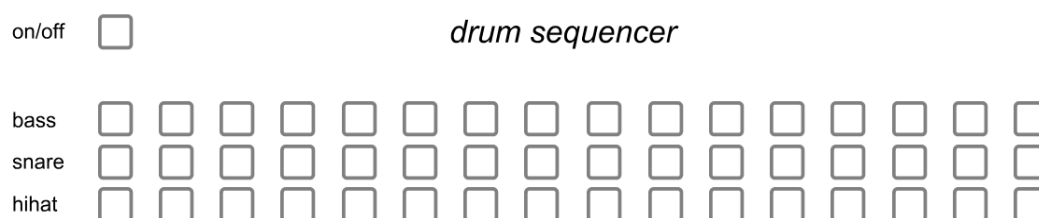


Figure 18: example drum sequencer interface

## Random drums

Open the patcher `randomdrum.maxpat` in MaxMSP. You should see the following:

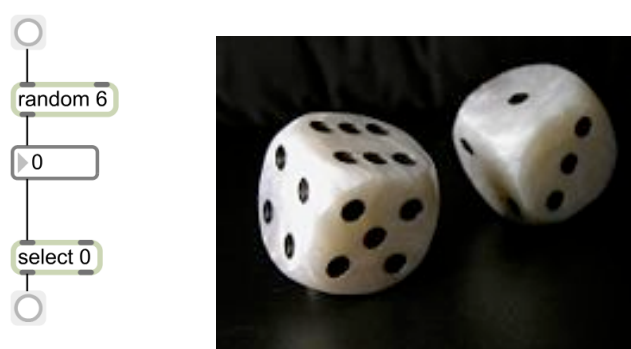


Figure 19: rolling a die<sup>4</sup>

This patch simulates rolling a die. The main difference is that it results in values 0–5 rather than 1–6. Clicking the button at the top causes random to generate a random number between 0–5 (i.e., from zero to one less than the argument provided<sup>5</sup>). If the

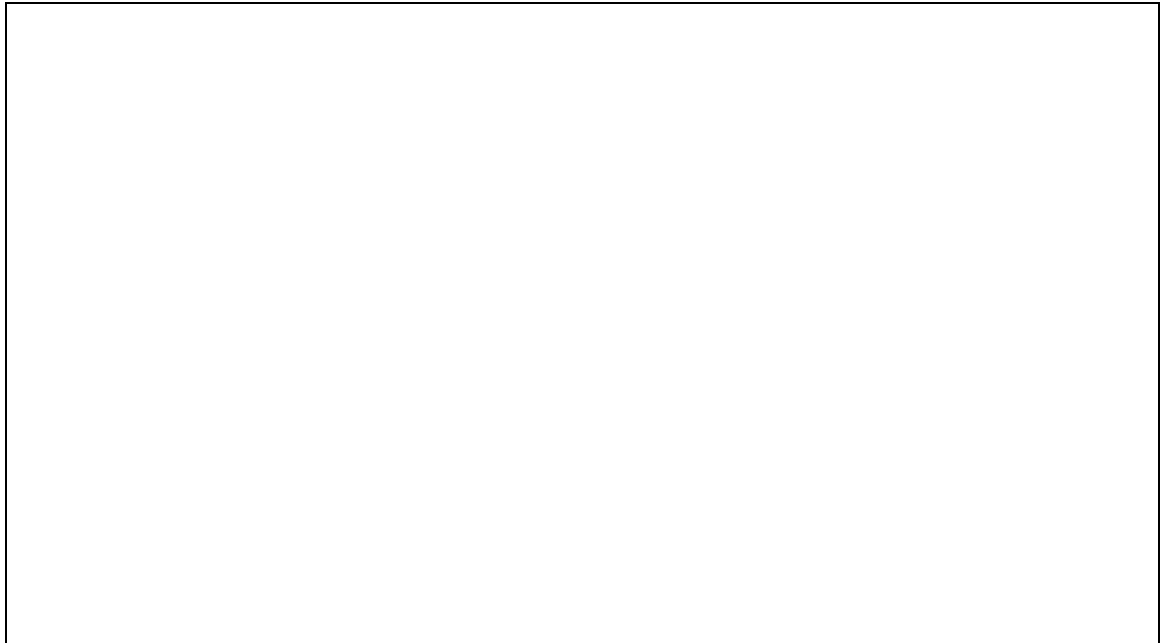
<sup>4</sup> Dice image from: <http://upload.wikimedia.org/wikipedia/commons/6/6a/Dice.jpg>

<sup>5</sup> You can think of the argument to `random` controlling the number of possible outcomes.



result of this “die roll” is a zero the select object detects this and causes the button at the bottom to flash. Thus the probability of “rolling” a zero is 1 in 6 or  $\frac{1}{6}$ .

Of course this lower button can then be used to trigger any event you like. Use the space below to draw a flowchart to demonstrate the operation of this patch.



### Exercise — random drum generator

Using the ‘randomdrum’ patch as a starting point:

- make a patch which plays a drum sound if the outcome of the “die roll” is zero;
- add a metro so this can generate a rhythmic pattern; and
- extend this to play at least six different drum sounds, each with their own individual “die roll” (i.e., random object) so they each play different patterns.

You may wish to experiment with different probabilities. For example if you set the argument to random to “2” there is a 50/50 chance that the sound will play on any particular sequence step. This may work better with instruments which tend to be played often in many musical styles (e.g., closed hi-hat or ride cymbal). If the argument to random is “40” there is only a 1 in 40 chance of that sound being played. This may be more suitable for sounds less frequently played in most many styles (e.g., a crash cymbal).

### 2.5.4 Additional exercises and/or homework

Complete the tasks in section 2.5.2 and 2.5.3 if you have not already done so.

#### Controlling the random drum generator

The right inlet of the random object overrides the argument so you can send numbers in here to dynamically change the probability of zero occurring. Use your patch from the ‘Exercise — adding more sliders’ (p54) – where you used the sliders D9–17 to

control a collection of sliders — to enable you to control these probabilities dynamically using these sliders.

### **Transport buttons**

Try to work out what kind of messages the Axiom's transport buttons transmit (i.e., play, stop etc.). Use the play and stop buttons to turn a metro on and off respectively.

### **2.5.5 Finishing up**

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

### **2.5.6 Important material**

Through following this practical session you should be familiar with:

- receiving and decoding MIDI control change and program change messages for controlling elements of MaxMSP patches;
- building simple sequencing patches; and
- using random numbers to produce simple musical results.

---

## 2.6 Practical 6 – Tables

This practical session examines some uses of the table object for storing numerical sequences.

### 2.6.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop ipa1/Max/practical06 folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.6.2 Tables

The table object is one way to store relatively large amounts of data in your patches. These can be used in various ways.

#### Tables for remapping data

Open the patcher `pitchcorrect.maxpat` in MaxMSP.

This example shows another method of **remapping** randomly generated note data onto a particular musical scale using the table object. It also shows another method of generating random values using the drunk object.

The drunk object produces a random sequence known as a **random walk**. Values generated by the random object do not take into account any previous choices of random value when producing their next value. Each random choice is completely independent. This is the same as flipping a coin or rolling a die. Compare this with the balls drawn from the National Lottery machines: the probabilities of which balls can emerge from the machine change slightly as each ball is chosen. This is because there are fewer balls in the machine after each choice. Thus some random processes rely on previous choices to influence the future choices. A random walk is similar in that its next choice can be a maximum of a certain **step** away from the previous choice. The drunk object in the patch is initialised with the arguments '24 5'. This means the range of random values is 24, meaning it can produce values between 0–23. The second argument specifies the step size, which means each time the drunk object receives a **bang** it generates a new random value which is between 0–4 away from the previous choice (if there has been no previous choice it starts around the midpoint, which is 12 in this case). In fact the step can be negative so in this case it can be in the range -4...+4. Random walks are a reasonably good starting point for generating random melodies.

In this patch we are using a technique to decode the octave and pitch class seen in a previous practical. In this case we are feeding the pitch class number into a table object. The table object can store a collection of integers in a series of numbered **slots** or **indices**. The number of slots the table object has is known as its **size**, each slot can store a value determined by its **range** and whether it can store **signed** or **unsigned** values. All of these settings can be configured in the table's inspector. Have a look at this now, you can see the most important settings by clicking the 'table' tab. (You should by now be familiar with opening an object's inspector. If this is not familiar you should put aside some time to review previous practicals.)

This table has a size of 12, meaning it has slots numbered 0–11 and a range of 12 meaning each slot can store values between 0–11. The simplest way to use a table is to store some values in it and request those values in your patch. If an integer is sent into a table’s left inlet it looks up the value stored in that numbered slot and outputs the value from the left outlet. In this patch values which represent the “white notes” on a piano keyboard output the same value (0, 2, 4, 5, 7, 9 and 11) as the input value. Values which represent the “black notes” on a piano keyboard (1, 3, 6, 8 and 10) output a different value, ‘rounding’ them to the nearest white note below. See Table 1 below.

<i>Input pitch class</i>	<i>Output pitch class</i>
0	0
1	0
2	2
3	2
4	4
5	5
6	5
7	7
8	7
9	9
10	9
11	11

**Table 1: Rounding notes to the "white notes"**

Double-clicking on the table object in a locked patch opens its editor window. You should notice that this looks similar to the object in the top right of the patch. This is an itable which enables a table and its editor to be embedded graphically in a patcher. In this patch the table and the itable share the same data store, this is achieved by naming them.

The table is given a name ‘**pitchmap**’ provided by the first argument, note that this name has no spaces (the rules for naming are similar to the variable naming rules in C). Once a table is named, this name is global across all patches in Max on your machine. (This allows data to be shared between patches but you need to be careful if you intend tables with the same name to store different data.) The itable in this patch is given the same name, ‘**pitchmap**’, in its inspector (under the ‘Table’ tab).

Finally the table is set to save the data within the patch using the appropriate option in its inspector. (Data can be saved in separate files but the method used here is simpler.)

Note the pgmout object is used here to allow you to send MIDI program change messages. This allows you to assign a different “instrument” to a MIDI channel. (There are many resources that will list the standard General MIDI program change instruments, one example is [http://www.jososoft.dk/yamaha/articles/midi\\_6.htm](http://www.jososoft.dk/yamaha/articles/midi_6.htm) which is in a useful format which we will see later.)

## Exercise—two part random music

Modify the patch to round the notes to the “black notes” on the piano keyboard. Add another musical part such that there is a bass line and a melody **using separate** **drunk objects**. (You may find using Table 1 helpful for working out the settings for the ‘table pitchmap’ object.)

## Melodic sequence

Open the patcher `simplemelody.maxpat` and the Excel file ‘GM-PC-Map.xls’.

This patch illustrates one way to generate simple minimalist patterns in MaxMSP. Before we explore this example let’s look at a means of getting useful data into MaxMSP from other sources. Open the excel file ‘GM-PC-Map.xls’. This list of program change instruments was copied and pasted from [http://www.jososoftware.dk/yamaha/articles/midi\\_6.htm](http://www.jososoftware.dk/yamaha/articles/midi_6.htm), this was done from FireFox into Excel. Then the columns were re-organised such that there is a single list of instruments (rather than four columns). You are going to paste this into a `umenu` object but before you do, the data still requires some additional formatting. Remember from previous practicals that the `umenu` object’s items are listed in the appropriate section of its inspector **separated by commas**. You can add these commas to the Excel spreadsheet before copying the data to the clipboard. Put a single comma in cell C1, hit return and then select cell C1 (using the mouse, or you should be able to press the up arrow ↑ key on the keyboard). The cell should now look like Figure 20 below.

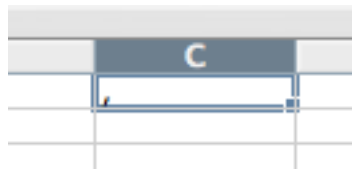


Figure 20: Comma in C1

Click and drag the handle in the bottom-right corner of this cell to copy the comma down to all the cells between C1–C127. Now select all of the cells A1–C128 and copy this to the clipboard using menu item **Edit→Copy**. Go back to the ‘`simplemelody.maxpat`’ patch and paste the clipboard (using menu item **Edit→Paste**) into the ‘Menu Items’ setting of the `umenu` inspector. You should now have a menu of all the standard GM program change instruments. Note that the menu has one added to its output since the output of the `umenu` will be from 0–127 (since its numbering starts at zero for the first item). Program change values need to be in the range 1–128.

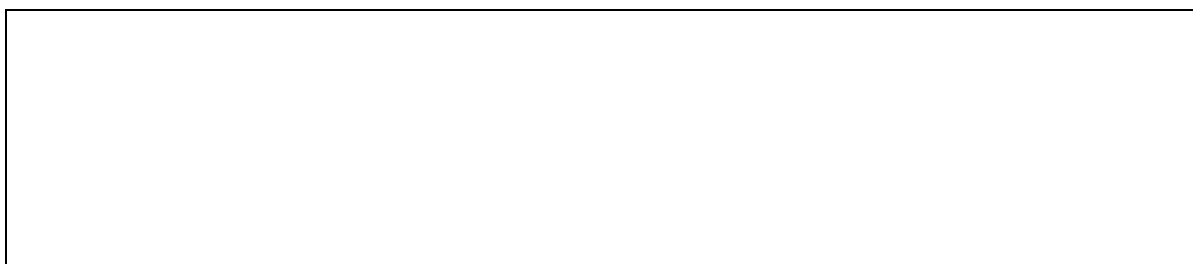
This patch uses a table to store a sequence of notes, which are looped continuously. This table has a size of 16 and a range of 60. Notice that there are two sections marked “Example 1” and “Example 2”, which are not connected to the other parts of the patch, you’ll be examining these in a moment. The part of the patch marked “Original” should play the sequence when the toggle at the top is turned on. Experiment with different sequences in the `itable` object.

Notice that this uses a counter without any arguments, which makes an unlimited counter (it should count to just over 2 billion if left for a very long time). The value is limited by the use of the ‘%’ object, here limiting the values to 0–15 since there are 16 slots in the table. The playback of the sequence can be manipulated in various

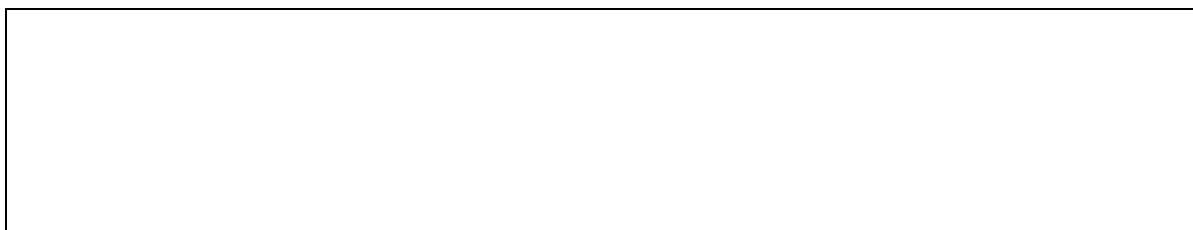
ways by modifying the counter value before this is used to lookup the stored value in the table.

### Exercise – minimalist composition

Leaving the original connected as a comparison, connect up “Example 1” such that the counter value has 15 added to it before this goes into the ‘% 16’. Listen to the result and note what effect has been introduced in the space below. (Look carefully at the other objects between the ‘+ 15’ and note out as many of these have been tweaked slightly!)



Disconnect the connection you just made for “Example 1” and now connect up “Example 2” instead. Again, note the result and how this is achieved in the space below.



Add two more modifications of your own based on these examples as ideas and a starting point. Connect up all five streams of notes and listen to your minimalist composition!

### Steve Reich (1936–), *Piano Phase* (1967)

Minimalist composer Steve Reich composed a piece called *Piano Phase* in 1967 as part of a collection of works which employ a **phasing** technique. Many of these pieces (including this one) have a simple pattern which is played in a loop such that multiple versions of the loop are heard simultaneously. These loops are then shifted out of phase, often very gradually.



Figure 21: Piano phase — score fragment<sup>6</sup>

---

<sup>6</sup> Image from: Potter, K. 2002. *Four Musical Minimalists: La Monte Young, Terry Riley, Steve Reich, Philip Glass*. Cambridge: Cambridge University Press. p185.

Open the patcher `pianophase.maxpat` in Max/MSP.

This patch implements the kind of phasing employed in *Piano Phase* and uses its melodic fragment as a starting point. Once the toggle at the top is turned on the original sequence starts. When this reaches step 96, the select object at the top starts the second sequence. Notice that there are two metro objects with a time difference of 1ms, it is this which causes the gradual phasing. When the original sequence reaches step 1000 the second sequence is stopped.

Other things to note in the patch:

- The two sequences are played on different MIDI channels (1 & 2).
- Both MIDI channels are set to play the same instrument using two pgmout objects.
- Two ct1out objects are used to pan channels 1 & 2, left and right respectively.
- The sequence is stored in a table called 'melody'.
- The 'select 1' object near the top of the patch does two things:
  - It resets the counters when the main toggle is turned on; and
  - It allows a zero to pass through to the second toggle enabling the main toggle to stop both sequences if desired.

### **Creative Exercise – “General MIDI Phase”**

Copy the umenu you created in section 'Melodic sequence' (p61) into this patch and connect it up appropriately to enable the user to select an instrument by name. Choose an instrument and experiment with the melody stored in the itable until you are happy with it.

Now add a few more “layers” to this musical composition by using the ideas from the 'Exercise – minimalist composition' (p62) to manipulate the counters, transpose the notes etc. Use the technique already employed in this patch to bring the layers in gradually over time, then gradually switch them off over time until only the original remains. Finally make the whole piece stop after playing a few final loops of this original sequence. The whole “piece” should last around 3 minutes.

## **2.6.3 Additional exercises and/or homework**

Finish section 2.6.2 if you have not already done so. You may wish to expand your solution to the 'Creative Exercise – “General MIDI Phase”' (p63) further by adding a percussive track using MIDI channel 10 (using ideas covered in an earlier practical).

## **2.6.4 Finishing up**

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

## **2.6.5 Important material**

This session introduced the use of the table object for storing and using numerical data. Ideas for generating minimalist music were explored. You should by now be familiar

with `umenu`, `'%'`, `select` and `counter`, and this session explored further applications of these objects. MIDI messages for panning (control change 10) were also introduced here.



---

## 2.7 Practical 7 - Storing Presets and Customised GUIs

This practical session introduces one of the ways in which MaxMSP patches can store “presets” (i.e., snapshots of the current state of sliders, numbers etc.) using the preset object. A method for incorporating customised graphics into a user-interface object (through the `matrixctrl` object) is also introduced. Finally, this practical reinforces material from previous weeks, including use of `umenu`, `table/itable`, `lists` and fundamental arithmetic objects.

### 2.7.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop `ipa1/Max/practical07` folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.7.2 Presets and more GUI issues

So far we have used the `loadmess` object for initialising values in our patches. (If you are not familiar with the `loadmess` object by now you should review previous practicals — it has been used a number of times. You should also review the `loadmess` Help patch, if necessary.) The `preset` object (shown in Figure 22 below) offers a means of storing “snapshots” of all the variable controls in a MaxMSP patch, including: number boxes, sliders (there are many and various types in MaxMSP), dials, tables, `itable`s and many more.

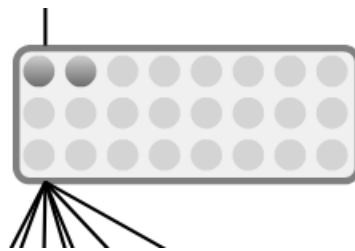


Figure 22: The preset object

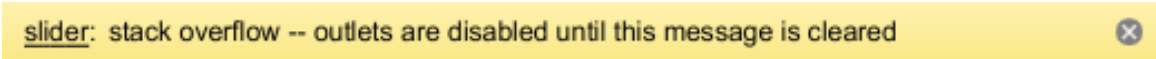
#### The preset Object

The simplest way to use `preset` is to place it in a patch: by default it can obtain and store **all** values from variable controls **in the same patcher**. It obtains these values using a hidden internal mechanism and as such, does not require patchcord connections. However, this “automatic” method can lead to problems since it stores and recalls **everything** (i.e., it may store and recall values you did not intend). The recommended approach is to specify exactly which objects `preset` should store/recall. To do this, patchcord(s) are connected between `preset`’s left outlet and the left inlet(s) of the object(s) in question.

Open the `presets.maxpat` patcher: This illustrates the recommended method of using `preset` as described above. To store a preset state into `preset` set up the parameters you wish to store and shift-click (in a locked patch) in one of the preset's **cells** (the small round buttons). If a cell has a preset state stored in it, it displays a different colour (compared to its empty state). You may notice that there are already two presets stored in positions 1 and 2 (the top-left cell and the cell to the right of this). Add a few more numbers and/or sliders (etc) to the patch and experiment with storing and recalling a few different settings into the preset object. Save and reopen the patch to observe that the data has been stored in the patcher file.

### ***Avoiding feedback loops (particularly in GUIs)***

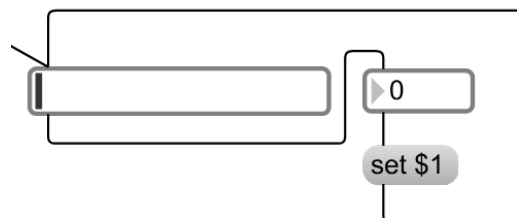
To the upper right hand side of the `presets.maxpat` patcher there is a horizontal slider and a number box side-by-side. With the patch unlocked, try connecting from the slider's outlet to the number's left inlet, then from the number's left outlet to the slider's inlet. This forms a feedback loop, which looks as if it should allow you to display a value using two different graphical objects. However, this kind of feedback loop will not work (as you may remember from previous practicals). With the patch locked, try moving either the slider or the number: you will get a **stack overflow** error. As shown in Figure 23.



**Figure 23: stack overflow error**

This error stops MaxMSP's scheduler — which controls the timing for metro objects, among many other things — so before you proceed you need to repair this error, by disconnecting the number box from the slider, and resume the scheduler. To do this click the 'x' button on the error bar.

A common method to avoid this problem is to insert a 'set \$1' message box in the feedback loop as shown in Figure 24 below.



**Figure 24: the 'set \$1' technique**

Note that this is a message box (with a solid/gradient fill) **not** an object box (with an outline only). Try inserting this 'set \$1' in the `presets.maxpat` patcher to fix the stack overflow problem. You should now be able to change either object and **both** will display the up-to-date value.

In the context of a message box, the special symbol **\$1** is used a **variable** — it is replaced by any numerical value (int or float) arriving into the message box. The entire message is

then transmitted by the message box to the next object(s) in the patch. For example, if a message box contains **'set \$1'** and it receives an integer **42**, it will transmit **'set 42'**. The slider, number and many other objects which store values, understand the **set** message to mean that they should update their internally stored (and displayed) value but **not** retransmit the value to the next object(s) in the patch. Thus breaking the any potential feedback loops at this point.

It is usually necessary only to connect one of these objects — connected in a loop — when storing these values in a preset (although in practice it is usually harmless to connect both).

### ***Exercise — controlling the preset***

The preset object will also recall presets when it receives an **int**. The presets are numbered 1 in the top-left, numbered across and then down. Try adding a number box to the patch so you can recall the stored presets. Now, add a umenu object containing the items "Preset 1", "Preset 2", "Preset 3" etc. to control the preset. Ensure this recalls the correct preset number.

## **Customising GUIs using `matrixctrl`**

Open the `matrix.maxpat` patcher. This introduces a new object, `matrixctrl`, which is a grid-based object commonly used for organising a collection of toggle-like switches in a **matrix** (although as we will see later it can be customised beyond this). In this patch, it is being used as a drum machine sequence controller; sequence step numbers are represented by the horizontal axis and the different instruments are represented by the vertical axis. Open the `matrixctrl` inspector and look under the "Behaviour" tab, notice that the "Number of Columns" is set to 16 and the "Number of Rows" is set to 4. (Also compared to default values "Autosize to Rows and Columns" is also turned on and "Scale When Resized" is turned off. Review the Help and Reference files for more information on these settings.)

Try the patch using the toggle at the top of the patch to turn it on/off.

This patch works by asking the `matrixctrl` for the contents of one of its columns (i.e., the sequence step number) using the **getcolumn** message, driven by the output of the counter. This uses the **"\$1"** technique as outlined in the 'Avoiding feedback loops (particularly in GUIs)' section above (p66).

The `matrixctrl` object responds to the **getcolumn** message by outputting a **list** out of its right outlet representing the state of the column in question i.e., which instruments are on (set to 1) and which instrument are off (set to 0). The list is displayed at each step by connecting the right outlet of the `matrixctrl` to the right inlet of a message box. The list is also connected to an `unpack` object which splits the **list** into individual elements so they can be examined separately. The `unpack` object needs to be provided with a number of arguments equal to the number and type of items in the list it should expect to be unpacking. In this case we need to unpack four **ints** so the `unpack` object is provided with four **0's**, giving it four outlets. The `select` object then decides whether to play the drum instrument or not (if the unpacked value is 1).

### Exercise — preset sequencing

Add a preset object to this patch to store the `matrixctrl` settings. Then compose four drum sequences (which should function as a four-bar sequence) and store the four sequences in cells 1–4 of the preset. Now make the patch automatically cycle the four bars one at a time (changing at the beginning of each bar, and returning to bar 1 after playing bar 4).

*Hint:* you could use a select object and another counter object, or you could use the ‘%’ and ‘/’ objects.

### Customising `matrixctrl`

Open the `matrix2/matrixCustion.maxpat` (i.e., the `matrixCustion.maxpat` patch in the `matrix2` folder). This illustrates how `matrixctrl` can be customised using specially designed image files. Experiment with the patch to observe that its behaviour is similar to the `matrix.maxpat` from earlier in this practical. One main difference is that the sequence steps have three possible values: 0, 1 or 2 (corresponding to the 0, – and + symbols). Look in the `matrixctrl` inspector under the “Value” tab, notice the “Cell Range” setting is 3 (i.e., 3 possible values 0, 1 or 2). To cycle through the possible states simply click on a particular cell multiple times.

This additional range is used to allow two possible “on” states, thus implementing a drum machine with “accents” (i.e., some steps which are louder). This enables the drum machine to be a little more expressive. This uses `table` and `itable` object for controlling the MIDI note velocity levels for the 0, 1 and 2 values.

Sophisticated images may be constructed using image editing software such as Adobe Photoshop. However, basic interfaces can be made by taking screenshots of sections of Microsoft Excel spreadsheets (which is how this interface was created).

Open the `matrix2/mymatrix.xls` file into Microsoft Excel. First ensure Excel is in “Normal” view mode (in the **View** menu) and Zoom is set to 100%, this will enable you to read the brief guidance in this file more easily. The image file needs to be in a format (as shown in Figure 25 below) such that the columns represent the possible states, and the cells within each column are used to display that state depending on certain conditions (such as mouse activity).

	A	B	C	D	
1	0	–	+		Unclicked state (i.
2					Clicked state (i.e.
3					Inactive state (ca
4					Mask for Unclicke
5					Mask for Clicked
6					Mask for Inactive
7					

Figure 25: `matrixctrl` image format in Excel

You are going to modify this file so that the interface will display **0**, **1** and **2** (instead of **0**, **-** and **+**):

1. First, change the symbols/characters in the cells as appropriate (you may also change the colour scheme if you wish).
2. Now change the view mode in the **View** menu to “Page Layout” (this makes taking an accurate screenshot easier). Ensure Zoom is still 100%.
3. Type ⌘-Shift-4 on the computer keyboard (this is the Mac shortcut for grabbing a section of the screen to a PNG image file), you should now see the mouse pointer change to a crosshair tool.
4. Align the crosshair tool such that the centre of cross is the top-left pixel of cell A1 in the spreadsheet. (This may be a little fiddly.) See Figure 26 below.



**Figure 26: alignment of the crosshair tool (top-left)**

5. Click and drag until the centre of the crosshair is the bottom-right pixel of cell C6 (see Figure 27, below) and release the mouse button. (A “hint” on the crosshair tool indicated the size of the selection, which should be 54 x 97 pixels in this case.)



**Figure 27: alignment of the crosshair tool (bottom-right)**

This will create a file ScreenshotXXXXX.png on the Desktop (where XXXXX is the date and time of the screenshot). Copy this file to the `matrix2` folder and rename it if you wish. The file is now ready to be added to the `matrixctrl` object in the `matrixCustom.maxpat` patch:

1. Open the `matrixctrl` inspector.
2. Under the “Image” tab click the **Choose...** button in the “Cell Image File” setting.
3. Select the image file you just created and click the **Open** button.
4. Ensure that all three options, “Has Clicked Image”, “Has Image Mask” and “Has Inactive Image” are turned **ON**.
5. Close the inspector.

Your new image file should work as your interface. Lock the patch to test it out now.

### ***Exercise - increasing the range of `matrixctrl`***

Modify this patch such that between 4–6 velocity levels (including zero) can be controlled from the sequence in the `matrixctrl` object. You will need to generate a new image file and modify the `table/itable` objects.

## **2.7.3 Additional exercises and/or homework**

Finish section 2.7.2 if you have not already done so. This is also a good opportunity to check you have completed ALL of the previous practicals including the additional exercises/homework.

## **2.7.4 Finishing up**

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

## **2.7.5 Important material**

By the end of this session you should be familiar with using the `preset` object for storing and recalling settings within your patches. You should also be familiar with using the `matrixctrl` object for producing simple drum machine control structures. You should also have an awareness of the means by which `matrixctrl` can be customised using imported graphics (and how these graphics may be generated).

---

## 2.8 Practical 8 – Subpatchers and Abstractions

This practical introduces the use of *subpatchers* and *abstractions* as a means of organising the functionality of your patches in MaxMSP. This is similar to the use of *functions* in C programming.

### 2.8.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop ipa1/Max/practical08 folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.8.2 Subpatchers and Abstractions

You will have almost certainly found already that it is easy to run out of screen space even with relatively simple patches. One solution (which you have probably used) is to extend your patch beyond the size of the window such that you can scroll left, right, up and down to find the various parts of the patch. This is perhaps equivalent to writing a longer and longer `main()` function in a C program. In C, programs may be divided into *functions*, as you have already seen in this week's C practical. In MaxMSP, patches may be divided into *subpatchers*, where some objects are *encapsulated* into a subpatcher which resides in the main patcher. First, we are going to examine an existing patch and divide it into subpatches.

#### Encapsulating

Open the `matrixProbable/matrixProbable.maxpat`, which is a modification of the solution to the exercise in the section titled 'Exercise - increasing the range of `matrixctrl`' of last week's practical (Section 2.7 p70). Before examining this patch in more detail you will notice the collection of objects near the top of the patch as shown in Figure 28 below.

Most of these objects have been used many times in the musical/sequencing examples. In this case the objects have a clear function as a group — the group is issued an on/off command, and it produces messages to request data from the `matrixctrl` object. As such, it makes a good candidate for subpatching.

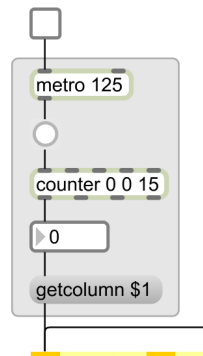
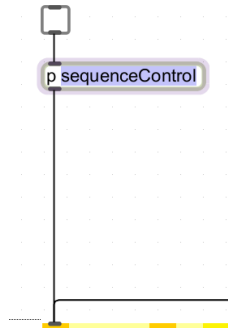


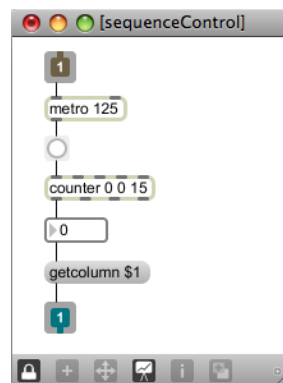
Figure 28: Candidate for subpatching

Unlock the patch and select the objects shown within the grey panel in Figure 28 (i.e., metro through to the `getcolumn` message). Now choose menu item **Edit → Encapsulate** and this should collapse these objects into a subpatch, which appears as a 'p' object. (The 'p' object is simply an abbreviated version of patcher, just as the trigger object can be abbreviated to 't'). It is a good idea to provide the patcher with a name that has no spaces and reflects the task that the subpatch performs, as shown in Figure 29.



**Figure 29: Naming the encapsulated patcher**

Now lock the main patch and double-click on this new patcher object. The patcher should now open into another window as shown in Figure 30.



**Figure 30: Inside the encapsulating patcher**

The important new objects in this subpatch are the inlet (i) and outlet (o) objects, these provide communication routes in and out of the subpatch. Note that you need not use the **Edit → Encapsulate** command to create subpatches, this is simply a convenient shortcut. You can create a subpatch directly in your patch by adding a patcher (or 'p') object. You would then need to add inlet and outlet objects manually from the object palette (see Figure 31) as appropriate.



**Figure 31: inlet and outlet objects on the object palette**



### Exercise

You are now going to examine what this patch does. Close the subpatch window, lock the main patch and turn on the main toggle. The `matrixctrl` object apparently stores a drum sequence, but matters are not quite that simple. In the space overleaf, try to explain in more detail the drum patterns this patch produces and the means by which it is achieved.



### Creating abstractions

Using the patcher (or 'p') object can be useful when the reason for subpatching is simply to save screen space in an enclosing patcher. When function of the subpatch can be reused elsewhere, creating an **abstraction** is a preferable method. An abstraction is simply a patch saved to disk like any other patch (i.e., a '.maxpat' file) which can be loaded into, and use within patches.

It is quite straightforward to convert a subpatch into an abstraction.

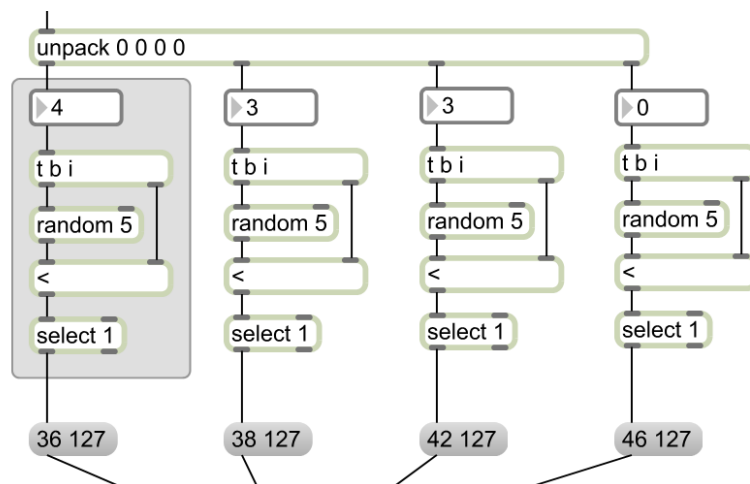


Figure 32: Candidate for an abstraction

Figure 32 above shows a candidate for an abstraction since this collection of objects is repeated several times in the same patch. To turn this into an abstraction:

1. Unlock patch and select the objects shown within the grey panel. Now use **Edit** → **Encapsulate** to encapsulate these objects into a subpatch. Do not name the 'p' object in this instance.
2. Lock the patch and double-click the 'p' object to view its contents in a separate window.
3. Choose menu item **File** → **Save As...** and save the patcher into the `matrixProbable/` folder with the name `rchoice1.maxpat`. It is **very important** that there are no spaces in the filename.
4. Now close this `rchoice1` patch and go back to the main patcher (`matrixProbable.maxpat`).
5. Edit the text in the 'p' object such that it reads `rchoice1` instead of 'p'.

This part of the patch should now look similar to Figure 33.

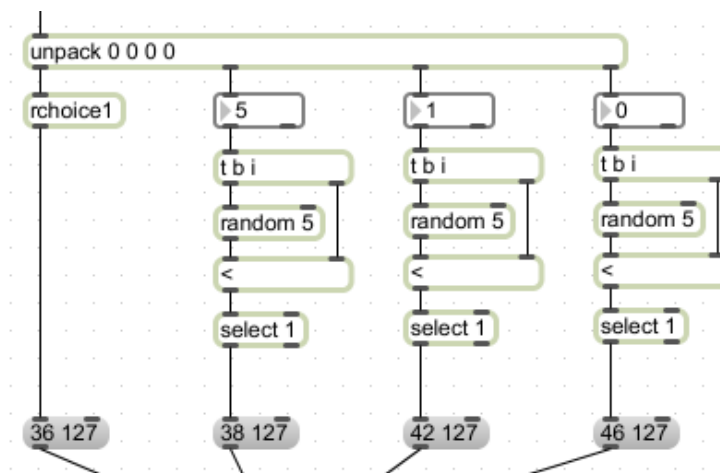


Figure 33: Using an abstraction

You may then replace the other instances of this collection of objects (and take advantage of the saving in screen space) as shown in Figure 34 below.

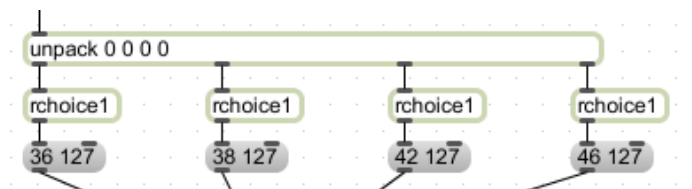


Figure 34: Repeating an abstraction

Now save the main patcher under a different name e.g., `matrixProbable2`, but ensure this is in the same enclosing folder (`matrixProbable/`) otherwise it will not find the abstraction! An important advantage to using abstractions is that there is only one copy of the abstraction on disk, this is then loaded — possibly multiple times — by the enclosing patchers.

This means two important things:

1. The abstraction cannot be edited from within the main patch. If you double click to open an abstraction loaded into an enclosing patcher it cannot be unlocked. To edit the abstraction, the file must be loaded in separately and saved.
2. When an abstraction file is edited (i.e., changed and saved) any other loaded patchers, which use this abstraction, automatically load the modified version.

You can test the latter by making a trivial change to `rchoice1`. Open `rchoice1.maxpat` separately and add a `comment` object. Once you save `rchoice1.maxpat` you can check that the main patch has loaded this new version (by double-clicking and looking inside the abstractions).

### 2.8.3 Abstractions with Arguments

We can take this abstraction further by making it more flexible. Close all patchers and open the **original** `matrixProbable/matrixProbable.maxpat` patcher (take a fresh copy from `Ranger` if you have modified the original). This time encapsulate the section of the patch shown in Figure 35 (from the `number` to the `message`) and save it as `rchoice2.maxpat` (again into the same folder).

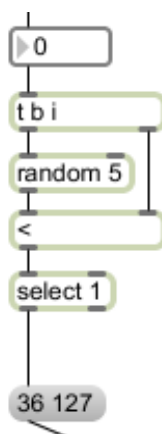


Figure 35: Flexible abstraction (part 1)

Clearly the problem with repeating ALL of these objects in this patch would mean that all the MIDI note numbers would be 36 (i.e., bass drum) rather than 36, 38, 42 and 46 respectively (for bass drum, snare drum, closed hi-hat and open hi-hat).

To remedy this, edit `rchoice2.maxpat` as shown in Figure 36, replacing the 36 with the special symbol `#1`.

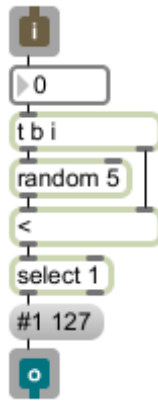


Figure 36: Flexible abstraction (part 2)

This special symbol **#1** in the message box will be replaced by the first argument to the abstraction when instantiated in an enclosing patcher. You can now make the part of the main patch (`matrixProbable.maxpat`) relevant to this discussion look like Figure 10 below.

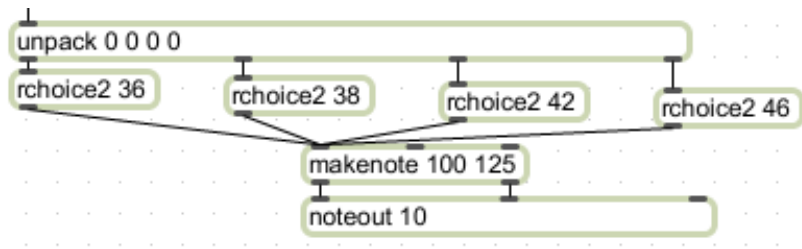


Figure 37: Flexible abstraction (part 3)

Once you have done this, save this main patch under a new name (e.g., `matrixProbable3.maxpat`) again making sure this is in the same folder as the others.

Abstractions can take up to nine arguments using special symbols **#1**, **#2** through to **#9**. These special symbols can appear anywhere in a patch, not only in message boxes (e.g., as arguments to other objects).

## 2.8.4 The MaxMSP search path

Once you begin to build your library of abstractions they can, of course, be used over and over again across many different projects. In this case you can use a central repository of abstractions rather than keep all patches in the same folder. To do this, create a folder in your homespace for keeping MaxMSP abstractions. For example, create a `maxmsp/` folder in `~/Documents`. Then create an `abstractions/` folder in the `~/Documents/maxmsp` folder.

Now add this folder (`~/Documents/maxmsp/abstractions/`) to the MaxMSP **search path** by choosing menu item **Options** → **File Preferences...** and click the **+** button (at the bottom of the window). Click the “Choose...” button and select the folder you just created.

Once you begin to use this method you must be very careful to copy **all** the abstractions used by your patches when moving your work to other computers, distributing your work or submitting coursework.

### 2.8.5 Exercise — makenote and noteout

The `makenote` and `noteout` objects are commonly used together and we often initialise these objects with a velocity value, a duration value and a MIDI channel. Make an abstraction that contains a `makenote` object and a `noteout` object. Add appropriate arguments (i.e., #1, #2 etc.) and appropriate `inlets` (this example does not need `outlets`). Use this abstraction in a main patcher (e.g., `matrixProbable`).

### 2.8.6 Additional exercises and/or homework

Finish section 3 if you have not already done so. Take at least one of the patchers from your previous work and apply the encapsulation and abstraction techniques as appropriate.

**IMPORTANT:** If you are using `preset` objects these will not be able to store data within `patcher` objects or abstractions without additional objects to help them. A solution to this will be examined next week using the `autopattr` and `pattrstorage` objects.

### 2.8.7 Finishing up

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

### 2.8.8 Important material

By the end of this session you should be familiar with using encapsulation to group functionality of MaxMSP patches in subpatches. You should also be able to convert encapsulated subpatches to abstractions by saving their contents into a separate file. You will need to practice this technique throughout your work to be able to use these tools in the most powerful way possible.



---

## 2.9 Practical 9 – Subpatchers and Abstractions II

This practical introduces the use of bpatchers for including the user interface of abstractions within other patchers. This practical will also examine the use of autopattr and pattrstorage to enable preset objects to work with subpatchers and abstractions, including bpatchers. This technique enables the entire state of a patch to be stored and recalled, no matter how many levels deep the data is stored within subpatches or abstractions.

### 2.9.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop ipa1/Max/practical09 folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.9.2 Subpatchers and Abstractions


Encapsulating via subpatching or abstractions is a way to organise MaxMSP patches more clearly. There are a number of other objects that are able to load abstractions (e.g., the poly~ object for polyphonic synthesis; and the pfft~ object for spectral — or frequency domain — processing). The bpatcher object enables abstractions to be loaded within an enclosing patcher **and** display the abstraction's interface (most usefully this is done using the abstraction in presentation mode).

We have examined the use of the preset object to store patcher data (for numerical and user interface data). The preset object does not work well when using subpatchers or abstractions which would be a particular problem when using bpatchers. Fortunately (in Max 5) this can be solved very easily with the addition of only two new objects autopattr and pattrstorage. It is suggested that you **do not** look at the Help or Reference for these two objects in the first instance. They are very powerful objects with many options and features that may seem overwhelming and confusing. Their default behaviour is sufficient for the time being.

#### More abstractions: bpatchers

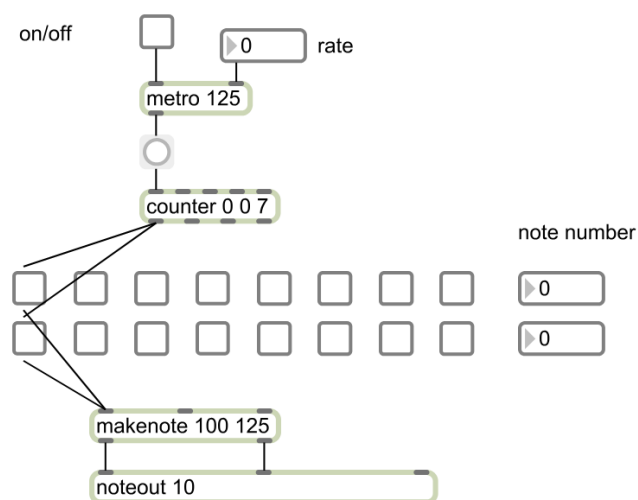
Open the subdrum/bpdrum.maxpat patcher, which is a modified version of the patch simpledrum.maxpat from Practical 5.

This patcher automatically loads in presentation mode. This option is set by turning on the “Open in Presentation” setting in the *Patcher Inspector*. You can open a Patcher Inspector by choosing menu item **View → Patcher Inspector** (this will open the Patcher Inspector for the front most patcher).

Switch to Patching Mode by clicking the  button. This abstraction takes the sequence step number (from a counter object) and decodes it via one of the outlets from the select object as a **bang** message. Each of the outlets from the select object passes via a gate object (controlled by a toggle). If the gate is open, the **bang** message passes through and triggers the output of the MIDI note number from the int object.

This then passes out of the outlet of the abstraction. (Which ultimately plays the note.) Notice that only the eight toggles and the number box are added to the patcher's Presentation Mode. In Presentation Mode these graphical objects are then placed in the upper-left corner of the patcher. The width and height of the patcher are not important.

Open the subdrum/subdrum.maxpat patcher, which uses the bpdrum.maxpat in a bpatcher (as shown in Figure 38 below). Put two valid MIDI note numbers for the drum channel (e.g., 36 and 38, for bass drum and snare drum respectively) in the two number boxes, and test the patch behaves as expected.



**Figure 38: Abstraction loaded via bpatcher**

Unlock the patch and you will notice a faint blue line around the toggles and number box, this faint blue line identifies the position of the bpatcher object(s). Select a bpatcher and open its inspector. The most important setting is the Patcher File setting (under the Patcher tab). The name of the Patcher File (or “abstraction file”) is entered here (or may be selected via the Choose... button.). Since bpdrum.maxpat is in the same folder, only its name, not its full path is shown. It is a good idea **not** to use full paths here since your patches will not work correctly on other computers or others’ logins. If you choose a Patcher File which is in the same folder as your main patch (or you choose a Patcher File which is elsewhere in the Max search path) Max will remove the full path leaving only the Patcher File’s name.

Using regular (non-bpatcher) abstractions, arguments may be typed into its object box as demonstrated in Practical 8. These arguments then replace any of the special symbols #1, #2 etc in the abstraction. Arguments may be used with bpatchers but the arguments are passed in via the Arguments setting in the bpatcher’s Inspector. (Since there is clearly nowhere else to type in the data in a graphical object!)

## Exercise — bpatcher

Extend subdrum.maxpat and bpdrum.maxpat such that there are 16 steps and 4 drum kit instruments.



## More On presets

Open the `mypreset/mypreset.maxpat` patcher. This shows four sliders in the main patch and four sliders in a subpatch (patcher object) called “asubby”. Check that the three pre-stored presets can be recalled.

Here a preset object is used, but it is merely an interface to the `pattrstorage` object, it does not store the data itself and is **not** connected to any objects. The `pattrstorage` object is given a name, here “mypreset”. This name is in turn entered into the preset object’s Inspector in the *pattrstorage* setting (which at the very end of the list under the All tab). Using this method data in the main patch **and** data in any subpatchers (including abstractions and bpatchers) may be stored in the preset/`pattrstorage`. This is achieved by adding an `autopattr` object to each patcher which needs its data stored in the preset/`pattrstorage`. The `autopattr` objects must then have their left outlets connected to the objects that need their data stored. (This is very similar to connecting the left outlet of the preset object as you have done previously.)

Click on the **clientwindow** message box in the main patch. The Client Objects window should appear, this shows all the objects which have their data stored by this `pattrstorage` object along with the current state of the objects (in the Data column) which will update if anything changes (try moving some of the sliders). These objects will have been named automatically by the `autopattr` object according to the objects’ type names and an index (e.g., “**slider[1]**”) where there is more than one object of the same type. The Client Objects window is also organised in a hierarchy that represents the structure of the entire patch, including subpatchers and abstractions (the four sliders in the “asubby” patcher are shown indented). Don’t worry about the Priority or Interp columns for now.

Click on the **storagewindow** message box in the main patch. The Storage Slots window should appear, this shows the contents of the **slots** (i.e., “presets”). Notice only slots 1, 2 and 3 have data (unless you have added some presets yourself).

The default behaviour of the `pattrstorage` object is to save its slot data into an XML file of the same as the `pattrstorage` name (in this example this is `mypreset.xml`). The XML file for this example is shown in Listing 1 below. You do not really need to understand XML in detail, especially since its implementation for the `pattrstorage` object is so simple. It is sometimes useful to edit the XML file. As further reading, in your own time, you can follow a tutorial here:

[http://www.w3schools.com/xml/xml\\_what\\_is.asp](http://www.w3schools.com/xml/xml_what_is.asp)

In this XML file, “`pattrstorage`” is the **root element** (in XML terminology) The “`pattrstorage`” element has an (XML) **attribute** “name”, which in this case is “mypreset”.

The “slot” elements are all **child elements** of the “`pattrstorage`” root and have a single attribute: its “number”. The “slot” element may have two types of child element: either a “`pattr`” element (which stores an object’s data) or a “`patcher`” element (which groups other “`pattr`” or “`patcher`” elements depending on the patch hierarchy).

The “`pattr`” element has two attributes: “name” (the name of the object as discussed above) and “value” (the data stored for this object in this slot).

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<pattrstorage name = "mypreset">
  <slot number = "1">
    <pattr name = "slider" value = "0" />
    <pattr name = "slider[1]" value = "41" />
    <pattr name = "slider[2]" value = "82" />
    <pattr name = "slider[3]" value = "127" />
    <patcher name = "asubby">
      <pattr name = "slider[4]" value = "68" />
      <pattr name = "slider[5]" value = "82" />
      <pattr name = "slider[6]" value = "58" />
      <pattr name = "slider[7]" value = "80" />
    </patcher>
  </slot>
  <slot number = "2">
    <pattr name = "slider" value = "127" />
    <pattr name = "slider[1]" value = "97" />
    <pattr name = "slider[2]" value = "52" />
    <pattr name = "slider[3]" value = "0" />
    <patcher name = "asubby">
      <pattr name = "slider[4]" value = "89" />
      <pattr name = "slider[5]" value = "60" />
      <pattr name = "slider[6]" value = "82" />
      <pattr name = "slider[7]" value = "56" />
    </patcher>
  </slot>
  <slot number = "3">
    <pattr name = "slider" value = "49" />
    <pattr name = "slider[1]" value = "97" />
    <pattr name = "slider[2]" value = "73" />
    <pattr name = "slider[3]" value = "48" />
    <patcher name = "asubby">
      <pattr name = "slider[4]" value = "43" />
      <pattr name = "slider[5]" value = "83" />
      <pattr name = "slider[6]" value = "51" />
      <pattr name = "slider[7]" value = "10" />
    </patcher>
  </slot>
</pattrstorage>

```

**Listing 1: the mypreset.xml file generated by the pattrstorage object**

### 2.9.3 Exercise—editing the XML

Close the mypreset.maxpat patch and open the mypreset.xml file into TextEdit (ctrl-click — or right-click — on the file in the Finder and choose **Open With → Text Edit**).

#### Simple editing

Modify the data such that the three slots have the contents as listed in Table 2 below.

<b>Object</b>	<b>Slot 1</b>	<b>Slot 2</b>	<b>Slot 3</b>
slider	10	20	0
slider[1]	20	40	50
slider[2]	30	60	10
slider[3]	40	80	60
slider[4]	50	10	20
slider[5]	60	30	70
slider[6]	70	50	30
slider[7]	80	70	80

**Table 2: Desired pattrstorage contents**

Save the file in TextEdit then check that this works by loading the `mypreset.maxpat` patch again into MaxMSP.

### **Adding data**

Close the patch again, now edit `mypreset.xml` adding a slot “4” with any data of your choice. Again check this works in MaxMSP.

### ***Exercise—presets and bpatcher***

Edit the `subdrum.maxpat` and `bpdrum.maxpat` patches such that their state can be stored in `pattrstorage/preset`. Include the toggles, the MIDI note number box and metro rate number box controls. Arrange the main patch using Presentation mode as appropriate and save the patch so it opens in Presentation mode automatically.

## **2.9.4 Additional exercises and/or homework**

Finish section 2.9.3 if you have not already done so. You should now add `bpatcher` and `pattrstorage/preset` functionality to your own patches where appropriate.

## **2.9.5 Finishing up**

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.

## **2.9.6 Important material**

By the end of this session you should be able to construct `bpatchers`, which enable you to enclose the interface of one patcher within another. This is another way of using abstractions. You should also be familiar with using the `pattrstorage` object in conjunction with the `preset` object to store the entire state of your patches — including subpatchers and abstractions (no matter how many levels deep the subpatches are located in your patch).



---

## 2.10 Practical 10 - More Lists

This practical illustrates various uses of Max *lists* (which are the equivalent of *arrays* in C). It is important that you complete both sections 2.10.2 and 2.10.3, whether in the practical session or in your own time.

### 2.10.1 Getting started

1. Log in to the Macintosh
2. Navigate to your desktop `ipa1/Max/practical10` folder.
3. Launch MaxMSP and ensure that the Max and Clue windows are visible.

### 2.10.2 Lists

The following section is adapted from 'Data Tutorial 5: List Processing' (available via Help→Max Help→Tutorials→Data→List Processing).

This practical covers a big topic: *list processing*. We cover several of the available modes of the `zl` object, which acts as a central container for a variety of list processing functions. We will also see how we can perform mathematical operations on lists elements with the `vexpr` object.

Within Max, the *list* is one of the most powerful data structures available. You can define any combination of values in a list, then have them sent as messages, get processed by objects or get treated as small tables. Much of this is made possible by the `zl` object, which gives you the ability to query, reorder and access any of the elements in a list. Adding `vexpr` into the list processing mix allows you to process list elements mathematically without having to break them into their individual components.

#### Example Patch

Open the example patch `listmelody.maxpat`.

This is the quintessential Max patch – the step sequencer. In addition to lists there are several other new objects introduced here. Firstly, the `multislidr` object, which is the graphical object shown in figure 1. In this patch the `multislidr` object is used to store a 32 note melody which is played back repeatedly when the start toggle is clicked, more on this later.

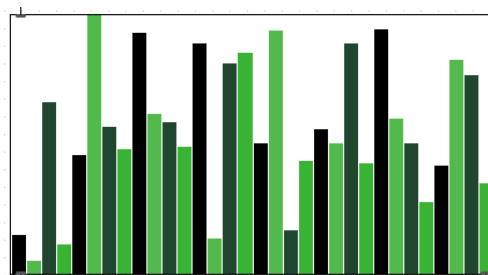
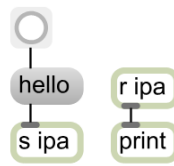


Figure 39: multislider object

Also introduced are the send and receive objects abbreviated here as *s* and *r*, respectively. These objects are quite straightforward. All messages received at the inlet of a send object are transmitted to all receive objects named with the same argument. This process is illustrated in the simple patch in figure 2.



**Figure 40: send and receive example**

When the button is clicked the message **hello** is received by the send object with the argument *ipa*. The message is then transmitted to the receive object with the matching argument (*ipa*) where the message is then printed in the max window. It is as if there were a patch cable connecting the **hello** message box and the print object.

In the `listmelody.maxpat`, the *s* and *r* objects are used to route lists to and from the `multislider` object to manipulate the melody data.

When the `listmelody.maxpat` is first opened, the `multislider` object is seeded with random values from the section of the patch labelled *generate random melody*. In this example, the `multislider` object contains a melody sequence for a MIDI playback system, which is fed to the `noteout` object connected to the right outlet. If you have not already done so, turn on the `metro` at the top-left by clicking the toggle, and the sequence will begin to play. Click on the `multislider` and drag your mouse to set values for the melody line – the result should play back using your computer's synthesizer.

At the right of the patch are a number of editing routines using the `zl` object, which we'll look at in more detail later. You can rotate, reverse, sort, transpose and randomly shift the melody. Click on the button objects that invoke the editing routines, and observe how the changes immediately affect the `multislider` contents.

The basic function of the step sequencer should be familiar: the `metro` drives a 32 position counter, whose output is turned into a **fetch** message for the `multislider` object. The corresponding output from the `multislider` is then sent to a `makenote/noteout` combination to trigger a note on MIDI channel 1.

## List Processing with `zl`

A key object when working with lists is the `zl` object: it is a single object that works in almost 20 different *modes* (set by the object's first argument), all of which perform useful list processing functions. In the case of the *generate random melody* section of the patch, the `zl` object is used in **group** mode, with an argument of 32. This means that the object will collect 32 values, group them into a single list, and then output the list from its left outlet. This is a quick and very efficient way to pack a stream of data into a list of predefined length, and turns an otherwise onerous task into a job for a single object.

The tutorial patch shows many uses of the `zl` object – particularly in the editing functions. All of these editing routines start with the `zl` object in *reg* mode. In this mode, the `zl` object will accept and store a list received on its *right* inlet. When the

object subsequently receives a **bang** in the left inlet, the list emerges from the object's left outlet. The `z1 reg` object is, in essence, the equivalent of the `int`, `float` or `value` objects, but is designed for list storage.

When you click on the button for each editing routine, it outputs the list from the `z1 reg` into another list processing object. Several of the editing routines use `z1` (in another mode) for the list manipulation; for example, the “rotate” routine uses `z1` in `rot` mode (with an argument of 1) to rotate the values by one entry – everything moves one position to the right, and the last entry becomes the first. The “reverse” routine uses `z1 rev` to *reverse* the order of an incoming list, while the “sort sections” combines three `z1` processing objects to *split* (`iter`), sort and reassemble (`group`) a list. Most `z1` modes take a second argument or allow a number in the right inlet to change a parameter - for example, the `z1 iter` object breaks a list into sub-lists of a size set by the argument (in our case, 4) that can also be changed by sending a number into the right inlet. If we change the number box connected to the `z1 iter` object to 32 (the entire length of the sequence) and then click the button, the melody stored in the `multislid` will become sorted from low values to high values.

Perhaps the best way to see all of the `z1` operating modes is to look at the `z1` help file. Unlock the patcher, select a `z1` object and choose “Open `z1` help” from the contextual menu or from the application help menu. The help file is broken into three segments, since there are so many modes – but you will be able to see all of the ways that `z1` can be used to bend, fold and mutilate Max lists.

## List Processing with `vexpr`

When working with lists, there are times when you want to perform mathematical expressions against all of the elements in a list. Using standard Max math objects (+, etc.) would require splitting, processing and reassembling a list – not the most efficient way of dealing with the problem. There is a solution: the `vexpr` object.

As you can tell by the name, the `vexpr` object is very similar to the `expr` object we learned about earlier, with the exception that it is made specifically to process lists. If we look at the use of `vexpr` in the “transpose” routine, we see that the familiar `$i1` syntax is used to reference the individual integer values of the list, and that we will add (or subtract) the number 1 from each entry. Therefore, when the `z1 reg` object outputs a 32-value list (sent originally from the `multislid`), each of the entries will be incremented or decremented, and will be output as a 32-value list.

The “randomise” routine shows the use of `vexpr` with *two* lists used for input. The right-hand side of the routine will generate a 32-value list of random numbers varying from -1 to 1 (the result of the `uzi`, `random 3` object and the `- 1` objects sent into a `z1 group`). This is input into the right inlet of `vexpr`. The left inlet receives the 32-value list from `multislid` being held by a `z1 reg`. The `vexpr` object iterates through the two lists, adding the elements from the right inlet (identified as `$i2`) to the elements from the left inlet (identified as `$i1`). This means that each of the list elements will be altered by a different random element, giving a small randomisation to the `multislid` contents. Note that this randomization can be done multiple times to gradually morph the melody into a new pattern altogether.

## List Processing Conclusion

Earlier we covered several ways that a `multislider` object's content (provided as a list) could be processed using the `zl` and `vexpr` objects. Since lists are an integral data structure in the Max environment, these objects will become an often-used part of your programming toolkit.

### 2.10.3 Additional exercises and/or homework

Open the example patch `pianoroll.maxpat`.

This patch illustrates a possible use of the `matrixctrl` object as a 'pianoroll' editor (typical in many audio/MIDI packages). It uses various list processing techniques in the subpatch `procMatrix` ('p `procMatrix`').

There are two presets, one bass line and one piano riff. Test these out then examine the patch in general. You must add comments to the subpatch `procMatrix` to describe how this patch works to the best of your understanding.

### 2.10.4 Finishing up

Make sure that you have backed up all your patches and email or copy the files to the user Drop Box of any other people you were working with, such that everyone has an electronic copy.