FTM Tutorial ver. Alpha 0.1

Foreword

The present tutorial attempts to teach how to use the FTM library for any purposes that the user might deem useful. It is not a tutorial on FFT signal processing, music programming, matrix maths nor any of the potential uses you can give to it. Think of it as a simple “How to” if you want to start using FTM for your own needs.

Traditionally, a tutorial is written by someone who masters the subject that the tutorial refers to. This is not the case for the document you are reading. This document started as an disorganised set of notes about what I thought the different FTM modules were doing, when I started to write, I had an extremely limited knowledge of FTM and still to this day I would never consider myself an expert. However, I do find that I can learn faster and things make more sense when I try to explan them to someone else. Thus, you can think of this work not as a philanthropic activity but my extremely selfish method of learning more about FTM myself.

You might find that at some of the chapters on this document there seems to be some missing information, generally this is intended, as I try to move in a pedagogical way and introduce new knowledge in an ordered fashion even if I have to go back to some subject later down the line. This is the result of how I've managed to understand the workings of FTM and my belief that someone else might find it useful. Again, the way this document is organised makes sense to me and anyone might think otherwise.

If you think that I am missing something important and wish to contribute, do no hesitate to contact me with explanations, sample patches, figures or anything that you might seem fit.

This document does not attempt to replace or compete with the official documentation available on the FTM website. Also, some parts have been copy+pasted directly from the website where the explanations given seem appropriate, in doing so I do not pretend to claim authorship of it but just present a document that can make it as easy as possible for any begginner to start patching with FTM. If you have authored any of the information I'm using and feel that I should give full and specific credit please contact me directly. I haven't done so yet as to advance faster in writing this tutorial.

I have no links or formal relationship with either IRCAM or the IMTR group. I do not know all the ins and outs of this library as the programmers do and you might find omissions or even mistakes in this document. If you find any such cases please do not hesitate to contact me directly on or off list and I will do my best to correct any issues.

Introduction

In my brief experience you could think of FTM more like a full system within Max, (think a bit like jitter). It allows for several things that you can't accomplish with vanilla Max objects and also allows you to accomplish your everyday patching needs in faster and more efficient ways. Of course all of this advantages come at a price and you need to learn some new things, syntax and behaviours. But trust me, in the end it is all worth it!

So let's start at the beginning. What is exactly FTM? How do I use it? What's in the box?

Basically, FTM is a library and set of Max modules that implements a whole new set of data structures to be used in Max. The basic idea of FTM is to extend the data types exchanged between the objects in a Max/MSP patch by complex data structures such as sequences, matrices, dictionaries, break point functions, tuples and whatever might seem helpful for the processing of music, sound and motion capture data.

In the IMTR group's paper “FTM – Complex Data Structures for Max”, we can read the following list of advantages.

- Static and dynamic creation of data structures (FTM objects) of predefined classes
- Editors and visualization tools
- Expression evaluation including functions, method calls, and arithmetic operators
- Graphical editors (written in Java) and visualization tools.
- Import/export of text, standard MIDI, SDIF and the usual sound file formats
- object serialization and persistence.

This clearly summarises the main features offered by FTM. However, for many Max users with little or no programming background, this list of features doesn't say much or is not clear enough. This could make it look like FTM is only useful for hard-core programming gurus and the like. Nothing further than the truth!
If we dig around the FTM website we can find Chris Dobrian's excellent set of tutorials. From his introduction, we can find a similar list of advantages provided by the use of FTM that read as follows:

- FTM has a consistent syntax for accessing and modifying a variety of data structures, thus operating on sizeable chunks of data all at once.
- FTM enables one to modify data in-place (without passing it from one object to another) and often remotely (without patch cords).
- Whereas Max's message timing and MSP's audio timing are related-but-separate, and the two only can interact once per sample vector (an interval of time determined by how many samples of audio MSP is set to process at one time), FTM enables one to enact processing of audio data with sub-sample accuracy, which is crucial for high quality accurate signal processing and granular synthesis.
- FTM allows one to relate various kinds of time-based and time-tagged data such as audio samples, breakpoint functions and other arbitrary curves, MIDI events, and descriptive musical "score" objects, with a set of objects that are designed to work together.
- FTM presents some new and useful ways of visualizing and editing that data.
- While FTM establishes its own object set and syntax for managing data, it also provides ways to convert that data for, or share it with, standard Max, MSP, and Jitter objects.

Now this list looks more understandable for the average Max user, and after reading that, some might y start to imagine what can be done with such a system. To all this goodness I would add the following:

- Faster and more efficient visualization of data. Say goodbye to [multislider] graphs!
- Quick implementation of algorithms on a pseudo-text-code-like environment within max.
- Interoperability of signal and message domain data in a more elegant, flexible and efficient way than using [snapshot~] and [sig~].

If we allow ourselves to speculate a little, we could also think of FTM as an interoperability layer between Max and PD (with the upcoming FTM port for PD). This might in the future to easily patch in both environments while keeping the same syntax and a similar workflow.

It might be confusing at the start to understand the new features provided by FTM and how they relate to the specific external objects that implement them. For this reason I will first explain the former and then dedicate sets of tutorials for the later.

Data structures.

Traditionally, Max deals with a limited number of data structures. Most of the time, one works with discrete messages and numeric values being passed around Max objects to form complex programs. The [pack], [pak], [unpack] and [zl] objects, provide means to group discrete values (or streams of values) into vectors, or in Max jargon, “lists”. In Max 4.6, lists are limited to 250 elements, which is often insufficient for some applications. Also, the number of processes and manipulations we can perform on list is limited.

The [table] and [coll] objects allow for storing and manipulating arrays of data with some limitations. Additionally, Jitter extends the functionality of Max in order to provide means to deal with matrix data, with big emphasis on video, 3D geometry and still images processing. This objects already offer some of the functionalities provided by FTM, but these official objects have their own purposes for which they were implemented in the first place, as well as a limited set of methods. Often, one needs to prepare the data in a certain way before storing it into any of this objects and do the same when retrieving data from them. This requires additional patching just to get data in and out and can become a burden for big projects and complex processes when you use these objects for things for which they were not created. FTM objects will often require a similar preparation of your data to get in and out, but when working in FTM land, you can exchange data structures between objects in a consistent and easier way. You can exchange data between FTM objects a lot easier than between say [coll] [table] [text] and [jit.matrix] objects, and back to Max standard objects.

Data structures are implemented in FTM as classes. Each class has it's own set of methods and can communicate with other classes. You can create recursive structures that contain classes within classes. It is important to understand classes independently from the objects that implement them. In the same way that a max list, can live in a message box, in the [zl] object, in a table or in the [jit.cellblock] object, FTM classes are not tied to individual objects.

Currently, the following FTM classes are implemented:

bpf	- break point function
dict	- dictionary of arbitrary key/value pairs
expr	- expression

fmat	- matrix of floats
fvec	- float vector or reference to an fmat row, column, diagonal or all (unwrapped) elements
llist	- linked list of any values
marker	- arbitrary marker
mat	- matrix of arbitrary values
metric	- metric marker
midi	- MIDI message
schema	- enumeration of names or pairs of names and default values (as tuples)
scoob	- score object
sequence	- sequence of time tagged events
track	- sequence with metrics and tempo
tuple	- immutable array of any values

As stated before, each class has it's own set of methods (messages it can understand), and we use the [ftm.mess] object to do so. Each class has an editor, that allows to view and edit the data contained in the class and the [ftm.vecdisplay] and [ftm.editor] objects offer advanced visualization and interaction with the contents of the different classes.
Only basic methods and operations are accessible from the [ftm.mess] object, and more complex operations are implemented as standard Max objects.

The [ftm.mess] Object

Basics

The first object we will get to know is the [ftm.mess] object, which is perhaps the most flexible and confusing to understand. For this chapter, we will only deal with what the [ftm.mess] object can do on it's own, that is, without calling other FTM classes.

You can think of the [ftm.mess] object as an message box with a multiple personality dissorder. It is (almost) equivalent to the standard Max [message box] object to display and send (almost) any message[1]. In most cases, you can simply use it as a substitute. It can be used to replace many of the standard Max objects and it can also be regarded as an extended and improved [expr] object.
So let's start!!

You can find the [ftm.mess] object in your toolbar as shown in Illustration 1.

Illustration 1: The [ftm.mess] object.

1 The difference between this two objects will be described in the following pages.

Tutorial 1a, shows an example of the [ftm.mess] object being used in the same way as the standard Max [message] object. On this example, we can see that in principle we can deal with all data structures common to Max: integers, floats, lists and symbols. Just like [message], it when clicked or when it receives a bang, it will send its contents through the outlet. These types of data (integer, float, list, symbol) are known if FTM as *primitive values.*

Just like [message], [ftm.mess] has some characters which you can't use to compose your messages as they are reserved and have a special meaning. The first of these characters is the "," (comma). When you separate elements by a "," in an [ftm.mess] object, the elements are outputted sequentially and not in a list.

Quotation marks <"> can be used in [ftm.message] and are sent out unaltered through it's outlet as opposed to [message] in which, quotation marks are used to create an exception and print out reserved characters instead of trying to evaluate them. Single quotation marks (or apostrophes) and parenthesis "(" ")" are reserved by [ftm.mess]. ***Tutorial 1b*** shows a detailed comparison between [ftm.mess] and [message].

Please notice that when we want to change the contents of [ftm.mess] we use a "#set" message. The hash symbol means that the "set" is meant for the object itself to change it's contents and not to the FTM objects that might be referenced within. All this will be clearer as we continue the tutorials.

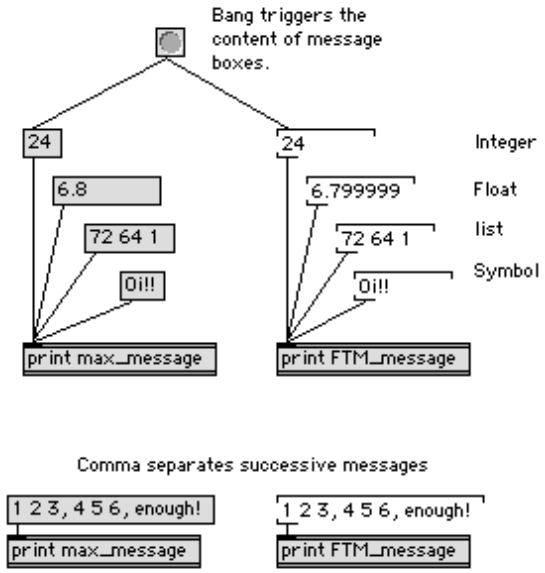


Illustration 2: Primitive values

Variables

We can use the "$" (dollar) sign to define variables in the [ftm.mess] object. Just like any other object that uses variables, the variable will be replaced with whatever it receives in it's inlet. However, unlike other Max objects, we don't need to specify the data type of the variable and we don't care if it is an integer, float, or symbol. You might have noticed by now that [ftm.mess] has two inputs by default, these corresponds to the $1 and $2 variables. If we type in more than 2 variables (i.e. $3, $4, $5, etc.) more inlets appear in [ftm.mess][2].

2 Currently there seems to be a bug in [ftm.mess] in which additional inlets are not created after typing. The eassiest workaround for this issue is to just copy or alt-drag the object. The newly created object will have the correct number of inlets.

The pseudo variable “$*” represents a list of all elements sent to the message box. The list represented by “$*” has as least as many element as the highest $-prefixed index.

The pseudo variables '$*1', '$*2', '$*3' represent the input of the individual inlets that can be single values or multiple values (lists and messages). Similar to the $1, $2, $3 one can use them to compose messages and lists in the message box. For example the expression '$*1 $*2' represents the concatenation of the inputs of the first and second inlet. By default the input lists are initialised to empty lists.

Please notice that the number after the “$” sign, always refer to the inlet register counted from left to right. So if we are using single values and lists on the same [ftm.mess] object, the number after the “$” or “$*” always refers to the corresponding inlet.

Tutorial 2. shows various examples of working with variables.

We can also define private variables and constants. To do so, we use the syntax “$X” where “X” is any non numerical character or string of characters. The value of these variables is set in the inspector or through initialisation expressions. We will look at the later method further down the tutorial.

The Inspector

I have mentioned the inspector a couple of times so far but have not yet fully explained it. Although the options are quite straight forward there are some small details that need to be explained.

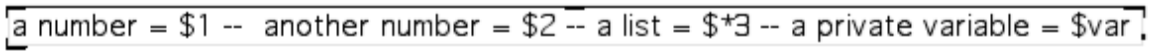
Illustration 3: ftm.mess inspector

Illustration 3 shows the inspector. The first section needs no explanation, on it, you can define the number of inlets and wheter or not all inlets trigger output. The third section sets if the contents of the [ftm.mess] object should be banged at start up, this saves you from using the standard Max [loadbang] object (note that you should always use this method when initialising other ftm objects).

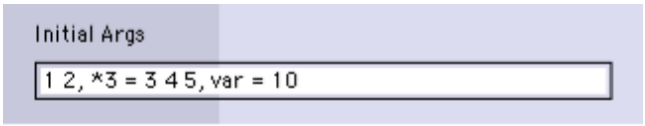
The fourth section is used to set the text and background colors.

The second section is used to initialise the object's arguments, or in other words, to set initial values for variables. For simple variables (ints, floats and symbols) we simply use a list of values with no additional syntax formatting. Values from left to right will become variables “$1”, “$2”, “$3”, etc.

When using lists or private variables, the following syntax is needed. <list of simple values>< “,” (comma)>< assignment>. So if we have an object like:



and we want the initial values to be “ 1 2 3 4 5 10”,where number 1 = 1, another number = 2, list = [3 4 5] and private value “var” = 10, then we declare it on the inspector as



For any additional list or private value that we want to initialise, we just need to separate each declaration with a “,” (comma). Additionally, if we type ** = [list] it will set all lists declared in the object to the defined values.
Do notice that when we are assigning initial values there is no need to type the “$” sign and we only use the register number for lists and the name for private variables.

Expressions

So far so good, we've seen some features of the [ftm.mess] object as a drop down replacement for Max's [message] object, we've seen some features that go far beyond what [message] has to offer and in general learned a lot about composing, manipulating and formatting messages, but there's more...

Remember how we could not use parenthesis on our messages? Well, as said before parenthesis can't be used to compose messages with [ftm.mess] because they are reserved. And they are reserved to evaluate expressions. Any time we have something enclosed in parenthesis in an [ftm.mess] object, the contents will be evaluated (or at least try to).
This opens a whole new set of possibilities and in taking advantage of them, we can no longer compare [ftm.mess] to Max's [message] object, instead, we can start thinking of it as any of Max's mathematical operator objects or the [expr] object. In this context, variables become something on which we can operate and not only display.

Tutorial 3a. Shows some comparisons between [ftm.mess] expressions and standard Max mathematical operation objects and the [expr] object. From this examples we can infer some important things. First of all is the syntax, there is some basic information that we need to give [ftm.mess] so it can evaluate our expressions correctly. We need to enclose the expression in parenthesis, this is in order to differentiate between what we want evaluated and what we want just “sent” out of [ftm.mess]. Second, we must declare our inlet registers using “$” variables. This might seem an extra effort since [ftm.mess] always have two inlets but that's just the way it is.

We can also see the advantages we have over [expr] regarding more than 10 variables and how the *"all inputs trigger"* option make [ftm.mess] more flexible.
So we can summarise expressions syntax as:

<parenthesis><inlet register> <operator> <inlet register> <parenthesis>
[($1 + $2)]

Please notice that inlet registers ("$" defined variables) and operators need to be separated by a space, this differs from the [expr] object which is space agnostic. Parenthesis don't need to be space separated from the contents within.

Tutorial 3b. Shows a full list of available operators with their corresponding standard Max objects. It is not the aim of this tutorial to go through basic mathematical concepts so if you need any detailed information on any of these operators you can unlock ***tutorial 3b*** and alt click on the Max object for each operator, the description will translate to the FTM operator transparently.

So before we move to more advanced things, we should summarise what we've covered so far.
Tutorial 3c. Shows an slightly advanced use of [ftm.mess]. On it, we can see some message formatting components, some simple variable substitutions and evaluation of expressions, all separated by the "," (comma) character which makes [ftm.mess] output each line sequentially.

At this point I would advice you to close this document and play around with [ftm.mess] trying out what we've covered so far and trying to familiarise yourself with it.