CREATOR TOOLS

A suite of tools developed to support
the instrument creation process

Contents

Application Manual

Scripting Reference

Contents

Application Manual

Creator Tools

A suite of tools developed to support the instrument creation process. It consists of the Debugger, the Instrument Editor and the GUI Designer. Switching between the tools is possible from the top tabs, or by using the shortcuts F1 (Debugger), F2 (Instrument Editor) and F3 (GUI Designer). Creator Tools actions can also be triggered via their dedicated shortcuts. Shortcuts act according to which panel the user is focused on.

Debugger

The Debugger connects to all running instances of Kontakt, both plug-in and standalone. It logs messages, warnings and errors coming for KSP, supports inspecting script variables, provides timestamps per notification and some basic filtering options.

VARIABLE WATCHING (⌘/CTRL-E)

Clicking on the eye icon reveals the Variable Watching area. This is where the current values of all watched variables and arrays are displayed, in order of appearance. For every variable or array that is inspected, an entry is created upon initialization and updated every time a value change occurs. All value changes appear also in the Log above, in chronological order.

Inspecting a variable or array is possible via the dedicated KSP commands `watch_var` and `watch_array_idx`.

For example `watch_var($count)` inspects the value changes of the variable `count` and `watch_array_idx(%volume,5)` inspects the value changes of index 5 of the array `volume`.

Please also refer to the KSP Reference Manual for more details.

FILTER (⌘/CTRL-F)

When active, it reveals the filtering options and applies them.

- Filter by type (Variable Watching, Message, Warning, Error)
- Filter by text (characters in the Message column)
- Filter by Instrument
- Filter by Script slot

PAUSE (⌘/CTRL-P) ||

Suspends the debugging session. While active, the Pause button blinks. Once the session is resumed, all messages that were received during pause will appear.

CLEAR (⌘/CTRL-BACKSPACE)

Clears all content of the Debugger log.

SETTINGS

Defines the behaviour of the Log.

LOG

This is where all notifications from Kontakt appear. The Log contains 7 columns:
- Type
- System Time
- Engine Time
- Message
- Instrument
- Script
- Line

Type and Message are set, but all other columns can be hidden. Right-clicking on the column header reveals the column menu.

Instrument Editor

The Instrument Editor connects to a running instance of Kontakt, either plug-in or standalone, and offers programmatic access to parts of a Kontakt instrument's structure through Lua-based scripting.

It loads and runs Lua scripts that have been created in a text editor and saved to disk. In this way an instrument structure can be modified. One can now easily rearrange, add or remove groups and zones, edit their names and some of their properties, like tune, volume, and mapping. Limited file system access also allows the creation of new instruments based on samples on the disk. The added MIR functions (like pitch and RMS detection) assist or automate parts of the instrument creation process.

Some Lua example and tutorial scripts are provided for the above in the application folder, to help you get started if needed. Ideally, the content of the scripts' folder can be copied to "*user*/Documents/Native Instruments/Creator Tools".

MULTI RACK MENU

Sets the focus of the tool on the multi rack of one of the connected Kontakt instances.

INSTRUMENT MENU

Sets the focus of the tool to a specific instrument that is loaded in one of the connected Kontakt instances. Note that instruments with locked edit views, cannot be selected.

PUSH (⌘/CTRL-Shift-↑) ↑

Applies all changes from the Tools' side to Kontakt. If changes are not pushed, an indication on the button appears to notify for the pending changes.

PULL (⌘/CTRL-Shift-↓) ↓

Overwrites the current Kontakt state to the tools. Whenever a change takes place on the Kontakt side, Pull needs to be manually pressed in order to apply the changes in the Tools. If changes are not pulled, an indication on the button appears to notify for the pending changes.

CONNECTION INDICATOR 🔗

The connection indicator on the top right corner, indicates whether a successful connection between the tools and the Kontakt instances is established.

INSTRUMENT TREE VIEW

The instrument structure is displayed in the form of a nested tree. The tree view shows the basic instrument structure and instrument properties that can be modified.

SCRIPT PANEL

Changes within the Tools happen exclusively via running a Lua script. A script can see and modify the instrument copy in the Tools. Scripts can be created and modified with an external editor. The script output will appear in the console output. All console output can be copied to system clipboard via the command ⌘/Ctrl-Alt-C.

LOAD (⌘/CTRL-L)

Opens the file explorer in order to locate a .lua file in disk and load it. The filename of the loaded file will then appear in the filename area.

[Currently the Creator Tools Lua runtime on Windows does not support filepaths that contain Unicode characters. Please rename the script's filepath accordingly to successfully load it.]

OPEN IN TEXT EDITOR (⌘/CTRL-O)

Opens the loaded script file in the system's default editor.

RUN (⌘/CTRL-R)

Executes the loaded .lua script. Changes are immediately reflected in the Instrument Tree View.

STOP (⌘/CTRL-I)

Stops the execution of the running script. The Instrument Editor state is reverted, as if the script never run.

CLEAR

Clears all content of the Script Output Panel.

GUI Designer

The GUI Designer allows one to assemble, customize and reuse Kontakt performance views and controls without the need to write code. It can generate two types of files, the performance view files (.nckp) and the control files (.nckc).

The performance view files (.nckp) contain all the information about an instrument's graphical interface. These files are created when a GUI Designer project is saved and can then be loaded in a KSP script (see also *Loading in KSP*).

The control files (.nckc) are files that are created by exporting a single control or a container of controls (see also *Panels*). These files can then be imported in a later GUI Designer project, shared with collaborators or set the foundation for building custom UI libraries. Control files cannot be loaded in KSP.

The two main areas of the tool are the *Tree View* and the *Properties*.

Tree View

The structure of a Kontakt performance view is displayed here in the form of a tree. A new performance view has one hierarchy level; the root level. More levels can be created when controls are added in panels (see *Panels*).

Actions on one or more selected controls can be performed from the *Tree View's* context menu. The context menu actions are:

- Cut (⌘/CTRL-X)
  Copies selection to the clipboard and deletes it from the tree
- Copy (⌘/CTRL-C)
  Copies selection to the clipboard
- Paste (⌘/CTRL-V)
  Pastes the controls from the clipboard above selection
- Duplicate (⌘/CTRL-D)
  Duplicates selection
- Rename (↵)
  Enters renaming mode for selection
- Delete (⌘/CTRL-⌫)
  Deletes selection

- Import (⌘/CTRL-I)
  Opens the system's file browser in order to locate and import a control file (.nckc) from the disk. The imported control will be placed above the currently selected control
- Export (⌘/CTRL-E)
  Opens the system's file browser in order to save the selected control's file (.nckc) in a desired location

Adding a control

INSERT CONTROL MENU +

A new control can be inserted in the performance view tree from the *Insert Control* menu. The menu lists all the known Kontakt UI controls, plus a new control called *Panel* (see *Panels*).

IMPORT

Previously exported controls can be added in the tree via the *Import* function of the context menu. Select a control and right click to reveal the context menu. Click on *Import* and locate the control's .nckc file in the system's file browser. Select *Open* and the control will be added in the tree, on top of the currently selected control.

PANELS

A panel is a control that can contain one or multiple controls. Unlike the rest of the controls, panels don't have size. They are very useful for grouping controls that are meant to be handled together. Then one can simultaneously modify the *Show*, *Position* or *zLayer* property of all the controls contained in that panel. The position of a contained control is relative to the panel's position. This means that the control's (0,0) position is the current (x,y) position of the panel.

Panels can be nested, so they can contain other panels. If panelA is contained in panelB, then panelA will appear in front of panelB. This is because children panels have a higher *zLayer* value than their parent panels. One could use this logic to easily create hierarchies in a performance view.

Panels can also be used to keep the tree view organized. They can be expanded or collapsed. When a panel is selected and expanded, new controls will be added on top of the panel's contained controls. When a panel is selected but collapsed, new controls will be added above it, on the same hierarchy level as the panel.

PANELS IN KSP

Panels, like any other control, can also be used with pure KSP outside the GUI Designer, using the following new command and control parameter:

```
declare ui_panel $<my_panel_name>
```
Creates a panel

```
set_control_par(<control-to-add-ui-ID>,$CONTROL_PAR_PARENT_PANEL,<panel-
ui-ID>)
```
Adds a UI control (or panel) in a panel

Example: Adding the volume knob in the mixer panel.

```
declare ui_panel $mixer
declare ui_knob $volume (0,300,1)

set_control_par(get_ui_id($volume),$CONTROL_PAR_PARENT_PANEL,get_ui_
id($mixer))
```

PROPERTIES

On the right side of the GUI Designer is the *Properties* area. Here, one can modify the properties of a selected control. Depending on the type of the property, editing can be done via text input, numeric input or dropdown menu selection. When an invalid value is entered, the property will be set to the last valid value. Pressing TAB takes the focus from the Tree View to the Properties area and vice versa.

IMAGE

Image fields take as input the filename of a picture (.png) that is contained in the *pictures* subfolder of the *Resources* folder.
Note: The *Resources* folder is the place to store files that an NKI can use, which are not samples. For more information please check *KSP Reference Manual - Working with the Resource Container.*

COLOR

Color fields take as input the hex code (six-digit, three-byte hexadecimal number) of a color.
A preview of the set color is displayed on the right side of the color input field.

FONTS

From the dropdown menu, one can select to set 1 of the 25 factory fonts or a custom font. If *"Custom"* is selected, then the filename of the picture font is expected. Similar to the *Image* property, the picture font should be contained in the *Pictures* subfolder of the *Resources* folder.

UNDO/REDO

Undo and Redo actions are available via the usual shortcuts (⌘/CTRL-Z) for Undo and (Mac: ⌘-Shift-Z, Windows: CTRL-Y) for Redo.

FILE MENU

From the File menu on the top right of the tool, one can *Save*, *Save As*, *Open* or create a *New* performance view. The GUI Designer saves the last 10 files that have been opened and displays them in the Recent Files list of the File menu. Existing files can also be opened by dragging and dropping the file from the OS to the GUI Designer.

LOADING IN KSP

The *Resource Container* is a dedicated location to store scripts, graphics, .nka files and impulse response files that can be referenced by any NKI or group of NKIs that are linked to the container.

When creating the *Resource Container*, Kontakt versions that are compatible with the GUI Designer will create a new subfolder named *performance_view*.

In order for a performance view to be displayed in an NKI, the performance view file (.nckp) must be stored in the *performance_view* subfolder of the NKI. It can then be loaded in the NKI via the KSP command:

```
load_performance_view(“filename”)
```

where `filename` is the filename of the .nckp file without the extension. Performance view filenames can only contain letters, numbers or underscores.

When saving a .nckp file in the GUI Designer, any changes will be automatically applied to the Kontakt side. This means when editing the performance view of an NKI, if that NKI is loaded in a running Kontakt instance, all changes can be previewed in real time upon Save.

To avoid conflicts, only one performance view file can be loaded per script slot. If needed, more controls can be additionally declared in the KSP script.

Connecting UI controls to engine parameters still happens via KSP. The variable name of a control contained in a performance view is auto generated based on its hierarchy, with underscore characters as concatenation.

Example:
`Control knobVolume` is contained within panel eqTab, which is also contained within panel `mixerTab`. The KSP variable name of the control will be: `mixerTab_eqTab_knobVolume`.

The KSP variable name of a selected control is displayed at the bottom of the properties area. Click on the Copy button next to it (or use the shortcut ⌘/CTRL-Alt-C) to copy the variable name to clipboard.

CREATING AND PREVIEWING YOUR FIRST PERFORMANCE VIEW

- Create a new NKI in Kontakt
- Create the *Resource Container* for the NKI
- Open a new performance view project in GUI Designer and start adding controls
- Make sure all the referenced data (images, picture fonts) are stored in the *images* subfolder of the NKI's *Resource Container*
- Save the performance view file in the *performance_view* subfolder of the NKI's *Resource Container*
- In the NKI script editor, write and apply the following script:

```
on init
load_performance_view (“filename”)
end on
```

- You can now continue to edit the performance view in the GUI Designer. Each time you want to preview any changes in the performance view, save the .nckp file in the GUI Designer.

Scripting Reference

Lua scripts can be loaded and run in the Instrument Editor tool to assist or automate tasks in the instrument creation process. This section of the documentation contains the scripting basics of the Lua language as well as extension bindings to a Kontakt instrument’s structure.

Instrument Structure

An instrument is shown as a nested tree with properties and values. Containers like groups and zones are represented as vectors (lists with indices). Property values are typed and value-checked so that changes are verified and ignored if the data is invalid.
The current structure looks like this:

```
Instrument                      Struct
     name                            String
     groups                          Vector of Group
          name                       String
          volume                     Real, -inf..12
          tune                       Real, -36..36
          zones                      Vector of Zone
               uniqueID              Int
               file                  String
               volume                Real, -inf..12
               tune                  Real, -36..36
               rootKey               Int, 0..127
               keyRange              Struct
                    low              Int, 0..127
                    high             Int, 0..127
               velocityRange         Struct
                    low              Int, 0..127
                    high             Int, 0..127
               sampleStart           Int, 0..inf
               sampleStartModRange   Int, 0..inf
               sampleEnd             Int, 4..inf
               loops                 Vector of Loop
                    mode             Int, 0..4 (see below)
                    start            Int, 0..inf
                    length           Int, 4..inf
                    xfade            Int, 0..1000000
                    count            Int, 0..1000000
                    tune             Real, -12..12
```

Loop modes:
0: Oneshot i.e. off
1: Until end
2: Until end alternating
3: Until release
4: Until release alternating

Future updates will allow more properties to be edited.

Scripting Basics

Scripting is based on the Lua language. Resources are available online e.g. www.lua.org.
The core language has been extended by bindings to the instrument structure. Whenever an instrument is connected and the tree view is displayed, a script can access it via the variable `instrument`.

SCRIPT PATH

The global variable `scriptPath` points to the directory of the executed script.
This is useful for file I/O related workflows.

READ PROPERTIES AND PRINT THEM

A script can print to the console e.g.

```
print(instrument)
```
Prints "Instrument" if an instrument is connected, otherwise "nil" i.e. nothing

```
print(scriptPath)
```
Prints the directory of the running script

All properties can be referenced using dots e.g.
`print(instrument.groups[0].name)`
Prints the name of the first group - or an error message if no instrument is connected

`print(instrument.groups[0].zones[0].keyRange.high)`
Prints the highest key range value of the first zone of the first group

ITERATE OVER CONTAINERS

The brackets [ ] refer to the nth element of the group or zone vector.
The number of elements can be read with Lua's length operator:

```
print(#instrument.groups)
```
Prints the number of groups of the instrument

```
for n=0,#instrument.groups-1 do
   print(instrument.groups[n].name)
end
```
Iterates over the groups of the instrument and prints their names

Note that vectors are zero-indexed! There are ways to iterate containers without using indices.

WORKING WITH CONTAINERS

```
Group()
```
This creates a new object of type Group

```
print(scriptPath)
```
Prints the directory of the running script

Structural changes like add, remove, insert are possible:

```
instrument.groups:add(Group())
```
This adds a new empty group at the end

The next example inserts a deep copy of the 4th group at index 0 i.e. at the beginning:
```
instrument.groups:insert(0, instrument.groups[3])
```

Binding Reference

Type

Base type of all object types. The following accessors are defined for objects off all types:

OPERATORS

`tostring`
Returns a string representation describing the object

FUNCTIONS

`object:equals(other)`
Returns true if object value is equal to the value of other

`object:instanceOf(type)`
Returns true if object is an instance of type
i.e. *object.type == type*

`object:instanceOf(name)`
Returns true if object is an instance of a type named name
i.e. *object.type.name = name*

`object:childOf(other)`
Returns true if object is a direct child of other
i.e. *object.parent == other*

`object:childOf(type)`
Returns true if object is a direct child of an object of type
i.e. *object.parent and object.parent.type == type*

`object:childOf(name)`
Returns true if object is a direct child of an object of a type named name
i.e. *object.parent and object.parent.type.name == name*

PROPERTIES

`object.typeinfo`
Returns type information as a string in the form Type‘Tag

`object.parent`
Returns the parent object or nil

Scalars

Basic types which contain a single value:

`Bool`
Boolean, true or false

`Int`
64 bit integer (signed i.e. can be negative)

`Real`
64 bit floating point number

`String`
Text

PROPERTIES

`scalar.type`
Returns the value type

`scalar.initial`
Returns true if the value is in the initial state

`scalar.value`
Returns the value

FUNCTIONS

`scalar:reset()`
Resets the value to its initial state

`scalar:assign(other)`
Assigns a copy of the other value object

Vector

Type-safe, dynamically sized, zero-indexed, random access container
For a `vector` object the following accessors are defined:

CONSTRUCTORS

`vector()`
Returns new vector

`vector(other)`
Returns a copy of the other vector

`vector(size)`
Returns a new vector with size elements

`vector(args)`
Returns a new vector initialized with variadic args

OPERATORS

`#`
Returns the number of elements i.e. *the size of the vector*

`pairs`
Returns an iterator function for iterating over all elements

`value/object = vector[index]`
Returns the value if vector type is scalar, otherwise returns the object

`vector[index] = value`
Sets value at index

`vector[index] = object`
Assigns object to index

PROPERTIES

`vector.type`
Returns the vector type

`vector.empty`
Returns true if the vector has no elements

`vector.initial`
Returns true if the vector is initial

FUNCTIONS

`vector:set(index, object)`
Set element at index to object

`vector:get(index)`
Returns object at index

`vector:reset()`
Resets the vector to initial

`vector:resize(size)`
Resizes the vector to size elements

`vector:resolve(path)`
Returns the object at path or nil

`vector:assign(other)`
Inserts object at the end

`vector:add(object)`
Inserts object at the end

`vector:add(value)`
Inserts value at the end

`vector:insert(index, object)`
Inserts object or value before index

`vector:insert(index, value)`
Inserts value before index

`vector:remove(index)`
Removes element at index

Struct

Type-safe, record with named fields.
For a `struct` object the following accessors are defined:

CONSTRUCTORS

`struct()`
Returns new struct

`struct(other)`
Returns a copy of the other struct

OPERATORS

`#`
Returns the number of used fields

`pairs`
Returns an iterator function for iterating over used fields

`value/object = struct.field`
Returns the value if field contains a scalar, otherwise returns the object

`struct.field = value`
Sets element value of field

`struct.field = object`
Assigns object to field

PROPERTIES

`struct.type`
Returns the struct type

`struct.empty`
Returns true if the struct has no used fields

`struct.initial`
Returns true if the struct is initial

FUNCTIONS

`vector:set(index, object)`
Assigns object at index

`vector:get(index)`
Returns object at index

`vector:reset()`
Resets the struct to initial

`struct:reset(index)`
Resets the field at index

`struct:reset(field)`
Resets the field

`struct:used(field)`
Returns true if the field is used

`struct:resolve(path)`
Returns the object at path

`struct:assign(other)`
Assigns a copy of the other struct

Algorithms

FREE FUNCTIONS

`path(object)`
Returns the path to object

`resolve(path)`
Returns the object at path or nil

`traverse(object, function(key, object, [level]))`
Recursively traverses object and calls function where key is the index or field name of the object in parent

`string = json(object, [indent])`
Converts objects to a json string and returns it

`object = json(type, string)`
Converts the string to an object of type and returns it

File system

The Lua binding is based on the C++ library boost filesystem. The reference documentation describes each function in detail. In contrary to the original C++ design the Lua binding does not define an abstraction for path. Instead path always refers to a Lua string.

EXAMPLES

```
for _,p in filesystem.directory(path) do
   print(p)
end
```
Lists paths in directory

```
for _,p in filesystem.directoryRecursive(path) do
   print(p)
end
```
Lists paths in directory and all sub-directories

FUNCTIONS

Note that all functions live in the global table filesystem.

Iterators:

```
iterator filesystem.directory(path)
iterator filesystem.directoryRecursive(path)
```

Path:

The functions return a string which contains the modified path.

```
string filesystem.native(path)
string filesystem.rootName(path)
string filesystem.rootDirectory(path)
string filesystem.rootPath(path)
string filesystem.relativePath(path)
string filesystem.parentPath(path)
string filesystem.filename(path)
string filesystem.stem(path)
string filesystem.replaceExtension(path, newExtension)
string filesystem.extension(path)
```

Query:

The functions query a given path.

```
bool filesystem.empty(path)
bool filesystem.isDot(path)
bool filesystem.isDotDot(path)
bool filesystem.hasRootPath(path)
bool filesystem.hasRootName(path)
bool filesystem.hasRootDirectory(path)
bool filesystem.hasRelativePath(path)
bool filesystem.hasParentPath(path)
bool filesystem.hasFilename(path)
bool filesystem.hasStem(path)
bool filesystem.hasExtension(path)
bool filesystem.isAbsolute(path)
bool filesystem.isRelative(path)
```

OPERATIONAL

These functions allow queries on the underlying filesystem.

Path:

```
bool filesystem.exists(path)
bool filesystem.equivalent(path1, path2)
int filesystem.fileSize(path)
string filesystem.currentPath()
string filesystem.initialPath()
string filesystem.absolute(path, [base])
string filesystem.canonical(path, [base])
string filesystem.systemComplete(path)
```

Test:

```
bool filesystem.isDirectory(path)
bool filesystem.isEmpty(path)
bool filesystem.isRegularFile(path)
bool filesystem.isSymLink(path)
bool filesystem.isOther(path)
```

Last Write Time:

```
int filesystem.lastWriteTime(path)
```

Links:

```
string filesystem.readSymLink(path)
int filesystem.hardLinkCount(path)
```

Note:
For convenience user can declare fs=filesystem and call all filesystem functions using the fs prefix:

Example:

```
fs=filesystem
iterator fs.directory(path)
iterator fs.directoryRecursive(path)
```

PosixTime

Date and time related utility functions.

EXAMPLES

```
print(posixTimetoString(filesystemlastWriteTime(...)))
```
Converts filesystem lastWriteTime to a string.

FUNCTIONS

Note that all functions live in the global table posixTime.

CONVERSION

Note that all functions live in the global table posixTime.

```
string posixTime.toString(int)
```
Converts the posix-time to an ISO string.

MIR functions

Music Information Retrieval (MIR) is the science of retrieving information from music. Among others, it allows the extraction of meaningful features from audio files, such as the pitch or the velocity of a sample. Creator Tools come with a collection of MIR functions, to assist or automate parts of the instrument creation process.

Single functions retrieve information from single files and take as argument an absolute filename (the full path to the sample file). Batch processing functions retrieve information from folders and take as argument an absolute folder name (the full path to the sample folder).

Note that all functions live in the global table mir.

PITCH DETECTION

The pitch detection tries to detect the fundamental frequency of a monophonic/single note sample. It ranges from semitone 15 (~20Hz) to semitone 120 (~8.4 kHz).

Pitch functions return a floating point value corresponding to the MIDI scale (69 = 440Hz). In case the pitch analysis fails, it will return a value of 0.

```
pitchVal = mir.detectPitch(‘fullPathToSample’)
```
Single function call

```
pitchBatch = mir.detectPitchBatch(‘fullPathToFolder’)
```
Batch processing

Batch processing will return a Lua table with samplePath as the table key and pitch as the value. It can be accessed in the following way:

```
pitchBatchData = mir.detectPitchBatch(‘fullPathToFolder’)
pitchValue = pitchBatchData[‘fullPathToSample’]
```

PEAK, RMS & LOUDNESS DETECTION

Peak, RMS and Loudness functions return a value in dB, with a maximum at 0dB.

The RMS and Loudness functions are calculated over small blocks of audio. The duration of those blocks is called frame size and is expressed in seconds. The process is repeated in intervals equal to the hop size (also expressed in seconds), until it reaches the end of the sample. The functions return the overall loudest/highest value of the different blocks.

If frame size and hop size are not indicated, the default values 0.4 (frame size in seconds) and 0.1 (hop size in seconds) are applied respectively.

Single Functions

```
peakVal = mir.detectPeak(‘fullPathToSample’)
```
Peak detection

```
rmsVal = mir.detectRMS(‘fullPathToSample’)

rmsVal = mir.detectRMS(‘fullPathToSample’, frameSizeInSeconds,
hopSizeInSeconds)
```

RMS detection

```
loudnessVal = mir.detectLoudness(‘fullPathToSample’)

loudnessVal = mir.detectLoudness(‘fullPathToSample’,
frameSizeInSeconds, hopSizeInSeconds)
```

Loudness detection

Batch Processing

Batch processing will return a Lua table with samplePath as the table key and peak (RMS, Loudness) as the value. It can be accessed in the following way:

```
peakValue = peakBatchData['fullPathToSample']
```

```
peakBatchData = mir.detectPeakBatch('fullPathToFolder')
```
Peak detection

```
rmsBatchData = mir.detectRMSBatch('fullPathToFolder')

rmsBatchData = mir.detectRMSBatch('fullPathToFolder',
frameSizeInSeconds, hopSizeInSeconds)
```

RMS detection

```
loudnessBatchData = mir.detectLoudnessBatch('fullPathToFolder')

loudnessBatchData = mir.detectLoudnessBatch('fullPathToFolder',
frameSizeInSeconds, hopSizeInSeconds)
```

Loudness detection