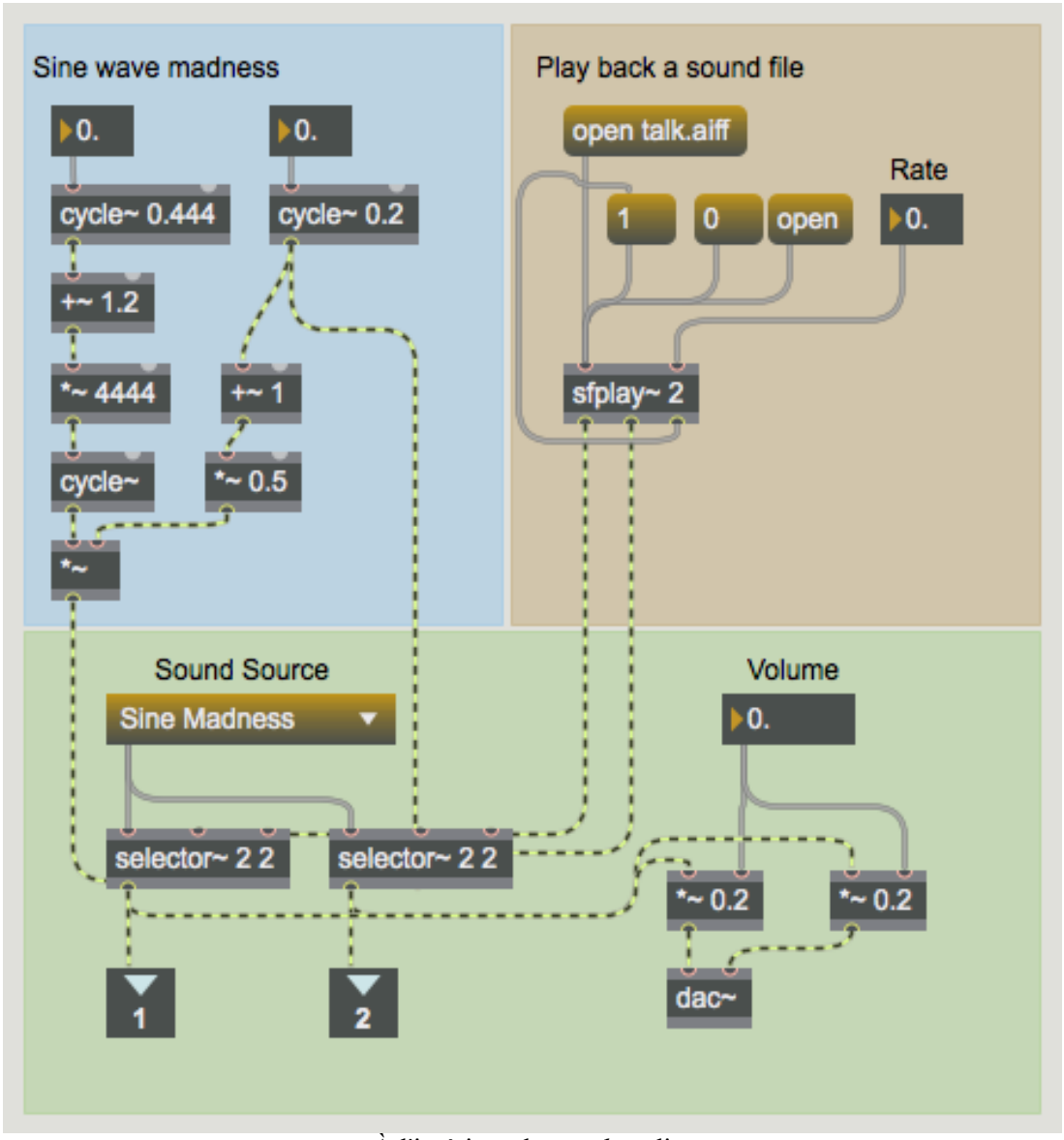
## 48-Trames de signaux MSP

Dans le *didacticiel 27*: *Utilisation de l'audio MSP dans une matrice Jitter*, nous avons appris à utiliser l'objet *jit.poke~* pour copier un signal audio MSP échantillon par échantillon dans une matrice Jitter Ce tutoriel présente *jit.catch~*, un autre objet qui déplace les données du signal vers le domaine matriciel. Nous verrons comment utiliser les objets *jit.graph* pour visualiser des données audio et d'autres données unidimensionnelles, et comment *jit.catch~* peut être utilisé dans un contexte d'analyse basé sur les trames. Nous rencontrerons également l'objet *jit.release~*, qui déplace les données des matrices vers le domaine du signal, et nous verrons comment nous pouvons utiliser les objets Jitter pour traiter et synthétiser le son.

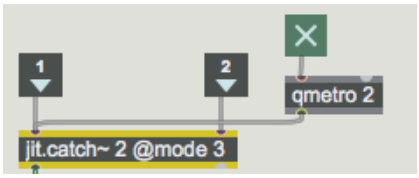
Le patch est divisé en plusieurs sub-patchs plus petits. En haut à gauche, le patch **audio** contient un réseau appelé ***Sine wave Madnesss*** où certains objets *cycle~* se modulent les uns les autres. La lecture d'un fichier audio est également possible via l'objet *sfplay~* dans le coin supérieur droit du sub-patch.




À l'intérieur du *patch* audio.

## Visualisation de base

• Cliquez sur la boîte de *message* intitulée **dsp start** pour commencer à traiter les vecteurs audio.
• Ouvrez le patch nommé **Basic visualization**. Cliquez sur le *toggle* connecté à l'objet *qmetro*.



*Capture et sortie des signaux audio sous forme de matrices Jitter.*

Dans le patch **Basic visualization**, les deux signaux du patch audio sont introduits dans un objet *jit.catch~*. La fonction de base de l'objet *jit.catch~* est de déplacer les données de signal dans le domaine des matrices Jitter. L'objet supporte la capture synchrone de plusieurs signaux; le premier argument de l'objet définit le nombre d'entrées de signaux; une entrée distincte est créée pour chacune. Les données de différents signaux sont multiplexées dans différents plans d'une matrice Jitter **float32**. Dans l'exemple de ce sub-patch, deux signaux sont capturés. En conséquence, nos matrices de sortie auront deux plans.

L'objet *jit.catch~* peut fonctionner de plusieurs façons différentes, selon l'attribut **mode**. Lorsque l'attribut **mode** est défini à **0**, l'objet émet simplement toutes les données du signal qui ont été collectées depuis la réception du dernier **bang**. Par exemple, si 1024 échantillons ont été reçus depuis la dernière fois que l'objet a reçu un **bang**, une matrice **float32** unidimensionnelle de **1024** cellules sera produite.

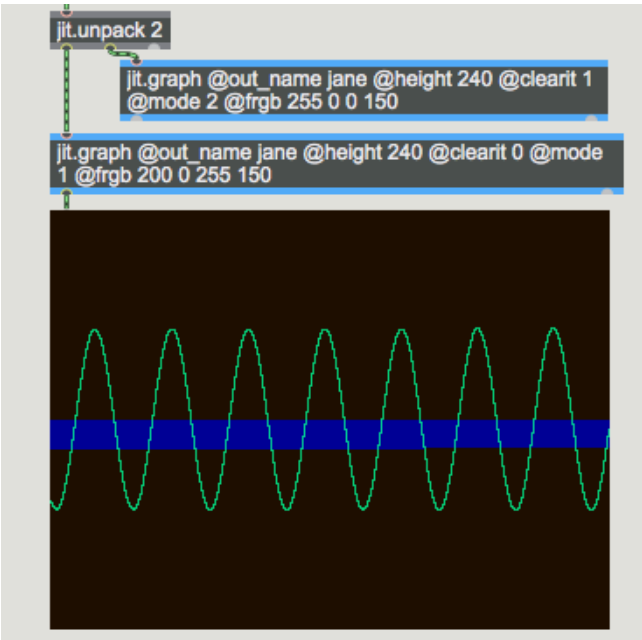
Le **mode 1** de l'objet amène *jit.catch~* à produire tout ce qui rentre dans un multiple d'une taille de cadre fixe, dont la longueur peut être définie avec l'attribut **framesize**. Les données sont disposées dans une matrice bidimensionnelle dont la largeur est égale à la taille de l'image. Par exemple, si la taille de l'image était de 100 et que les 1024 mêmes échantillons étaient mis en cache interne en attendant d'être sortis, en **mode 1**, un **bang** entraînerait l'émission d'une matrice **100x10**, et les 24 échantillons restants resteraient dans le cache jusqu'au **bang** suivant. Ces 24 échantillons se trouveraient au début de la trame de sortie suivante, à mesure que d'autres données de signal sont reçues.

Lorsqu'il travaille en **mode 2**, l'objet génère les données les plus récentes de **framesize**. En utilisant l'exemple ci-dessus (avec une **framesize** de 100 et 1024 échantillons capturés depuis notre dernière sortie), en **mode 2**, notre objet *jit.catch~* sortirait les 100 derniers échantillons reçus.

Dans notre sub-patch **Basic visualization**, notre objet *jit.catch~* est configuré pour utiliser le **mode 3**, ce qui le fait agir de manière similaire à un oscilloscope en mode trigger. L'objet surveille les données d'entrée pour les valeurs qui franchissent un seuil défini par l'attribut **trigthresh**. Ce mode est particulièrement utile pour l'observation de formes d'onde périodiques, comme c'est le cas dans cet exemple.

> Notez que le **mode 0** et le *mode 1* de l'objet *jit.catch~* tentent de sortir chaque valeur de signal reçue exactement une fois, alors que le **mode 2** et le **mode 3** ne le font pas. De plus, les matrices sorties en **mode 0** et **mode 1** seront de taille variable, alors que celles sorties en **mode 2** et **mode 3** auront toujours la même dimension. Pour tous les modes, l'objet *jit.catch~* doit maintenir un pool de mémoire pour contenir les valeurs des signaux reçus pendant qu'ils attendent d'être émis. Par défaut, l'objet mettra en cache 100 millisecondes de données, mais cette taille peut être augmentée en envoyant à l'objet un message **bufsize** avec un argument du nombre de millisecondes souhaité. En **mode 0** et en **mode 1**, lorsque plus de

données sont entrées que ce qui peut être stocké en interne, le prochain **bang** ne sortira pas de matrice mais provoquera un message de dépassement de tampon à partir de la sortie dump (à droite) de l'objet.




*Notre visualisation de base des données audio*.

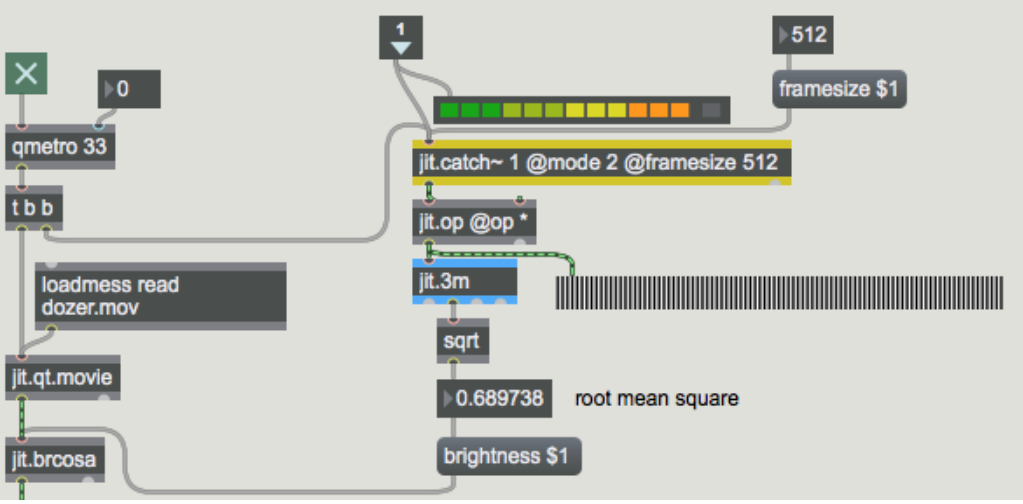
Comme les deux signaux sont multiplexés sur différents plans des matrices de sortie, nous utilisons l'objet *jit.unpack* pour accéder aux signaux indépendamment dans le domaine Jitter. Dans cet exemple, chacune des matrices à plan unique résultantes est envoyée à un objet *jit.graph*. L'objet *jit.graph* prend des données unidimensionnelles et les étend en un tracé bidimensionnel adapté à la visualisation. La plage basse et haute du graphique peut être définie avec les attributs **rangehi** et **rangelo**, respectivement. Par défaut, la plage du graphique est comprise entre **-1,0** et **1,0**.

Ces deux objets *jit.graph* ont été instanciés avec un certain nombre d'arguments d'attributs. Tout d'abord, l'attribut **out_name** a été spécifié afin que les deux objets soient rendus dans une matrice nommée **jane**. Deuxièmement, l'objet *jit.graph* droit (qui s'exécutera en premier) a son attribut **clearit** défini à **1**, tandis que l'objet *jit.graph* de gauche a son attribut **clearit** défini à **0**. Comme vous pouvez vous y attendre, si l'attribut **clearit** est définit à **1**, la matrice sera effacée avant d'y rendre le graphique; sinon, l'objet *jit.graph* rend simplement son graphique par dessus de tout ce qui se trouve déjà dans la matrice. Pour visualiser deux canaux ou plus dans une seule matrice, il faut donc que le premier objet *jit.graph* efface la matrice (**clearit 1**); les autres objets *jit.graph* doivent avoir leur attribut **clearit** réglé sur **0**.

L'attribut **height** de l'objet *jit.graph* spécifie la hauteur en pixels de la matrice rendue. Un attribut **width** n'est pas nécessaire car la matrice de sortie a la même largeur que la matrice d'entrée. L'attribut **frgb** contrôle la couleur de la ligne rendue sous la forme de quatre valeurs entières représentant les valeurs alpha, rouge, verte et bleue souhaitées de la ligne. Enfin, l'attribut **mode** de *jit.graph* permet quatre systèmes de rendu différents: le **mode 0** rend chaque cellule comme un point; le **mode** 1 relie les points en une ligne; le **mode 2** ombrage la zone entre chaque point et l'axe zéro; et le **mode bipolar 3** reflète la zone ombrée autour de l'axe des zéros.

## Trames

• Désactivez la *qmetro* (en décochant le *toggle*) et fermez le sub-patch **Basic visualization**. Ouvrez le patch nommé **frame based analysis** et cliquez sur le *toggle* à l'intérieur pour activer l'objet *qmetro*.

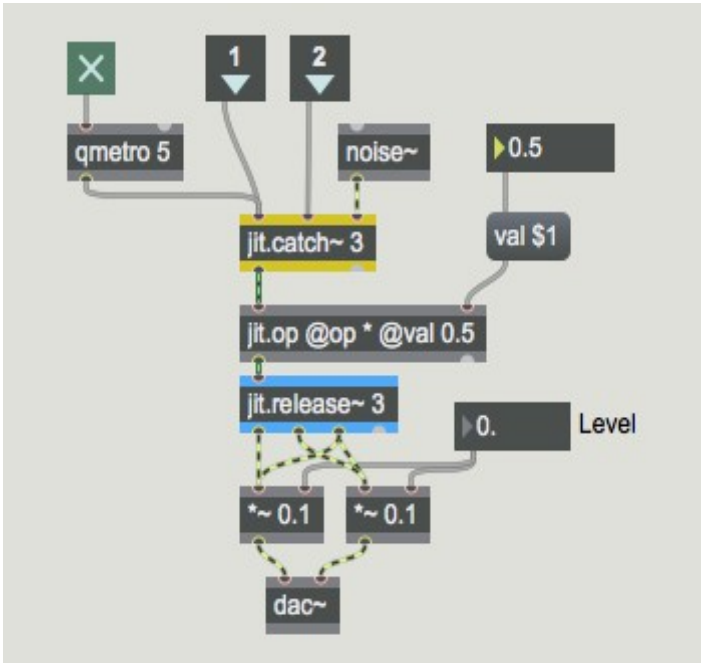



*Effectuer une analyse basée sur des trames de notre audio.*

Le sub-patch **frame based analysis** montre un exemple de la façon dont on pourrait utiliser la capacité de l'objet *jit.catch~* à rejeter toutes les données sauf les plus récentes. La sortie mono-plan de *jit.catch~* est envoyée à un objet *jit.op* qui la multiplie par elle-même, en élevant au carré chaque élément de la matrice d'entrée. Ceci est envoyé à un objet *jit.3m*, et la valeur moyenne de la matrice est ensuite envoyée par la sortie centrale de l'objet vers un objet *sqrt*. Le résultat est que nous calculons la ***root main square*** (écart quadratique principal) (RMS) du signal - une façon standard de mesurer la puissance du signal. En utilisant cette valeur comme argument de l'attribut **brightness** d'un objet *jit.brcosa*, notre sub-patch fait correspondre l'amplitude du signal audio à la luminosité d'une image vidéo.

Notre technique d'analyse basée sur les images donne une bonne estimation de l'amplitude moyenne du signal audio sur la période de temps immédiatement avant que l'objet *jit.catch~* ne reçoive son **bang** le plus récent. L'objet *peakamp~*, qui, lorsqu'il reçoit un **bang**, émet la valeur la plus élevée du signal atteinte depuis le dernier **bang**, peut également être utilisé pour estimer l'amplitude d'un signal audio, mais il présente quelques inconvénients par rapport à la technique *jit.catch~*, qui examine que les 512 derniers échantillons, trouvant ainsi un équilibre entre précision et efficacité. L'effet de cette économie est amplifié dans les situations où l'analyse elle-même est plus coûteuse, par exemple lorsqu'on effectue une analyse FFT sur la trame.

• Désactivez la *qmetro* et fermez le sub-patch **frame based analysis**. Ouvrez à nouveau le patch nommé **audio** et entrez un **0** dans la boîte de *nombre* connectée aux objets ***~** au bas du patch; les signaux générés ne seront plus envoyés à vos enceintes. Fermez le sub-patch audio et ouvrez le patch nommé **processing audio** avec des objets Jitter.




*Utiliser des objets Jitter pour traiter les données audio, transformer les résultats en signaux.*

L'objet *jit.release~* est l'inverse de *jit.catch~*: les matrices multi-plans sont entrées dans l'objet qui émet ensuite plusieurs signaux, un pour chaque plan de la matrice entrante. L'attribut **latency** de l'objet *jit.release~* contrôle la quantité de données de matrice qui doit être mise en mémoire tampon avant que les données ne soient émises sous forme de signaux. Dans le cas d'un réseau Jitter qui fournit des données à un objet *jit.release~*, plus la latence est longue, plus la probabilité que la mémoire tampon soit insuffisante est faible - c'est-à-dire que le réseau Jitter piloté par les évènements ne sera pas en mesure de fournir suffisamment de données pour les vecteurs de signaux que l'objet *jit.release~* doit créer à un rythme constant (***piloté par les signaux***).

• Cliquez sur le *toggle* dans le sub-patch pour lancer l'objet *qmetro*. Vous devriez commencer à entendre nos ondes sinusoïdales mélangées à du bruit blanc (fourni par l'objet *noise~*). Modifiez la valeur dans la boîte de *nombre* attachée à la boîte de *message* intitulée **val $1**. Notez l'effet sur l'amplitude des sons provenant du patch.

La combinaison de *jit.catch~* et *jit.release~* permet d'effectuer le traitement audio en utilisant des objets Jitter. Dans cet exemple simple, un objet *jit.op* se trouve entre les objets *jit.catch~* et *jit.release~*, multipliant efficacement les trois canaux audio par le même gain. Pour un exemple plus complexe de l'utilisation de *jit.catch~* et *jit.release~* pour le traitement de l'audio, regardez l'exemple *jit.forbidden-planet*, qui effectue un traitement FFT basé sur les trames dans Jitter.

## Varispeed

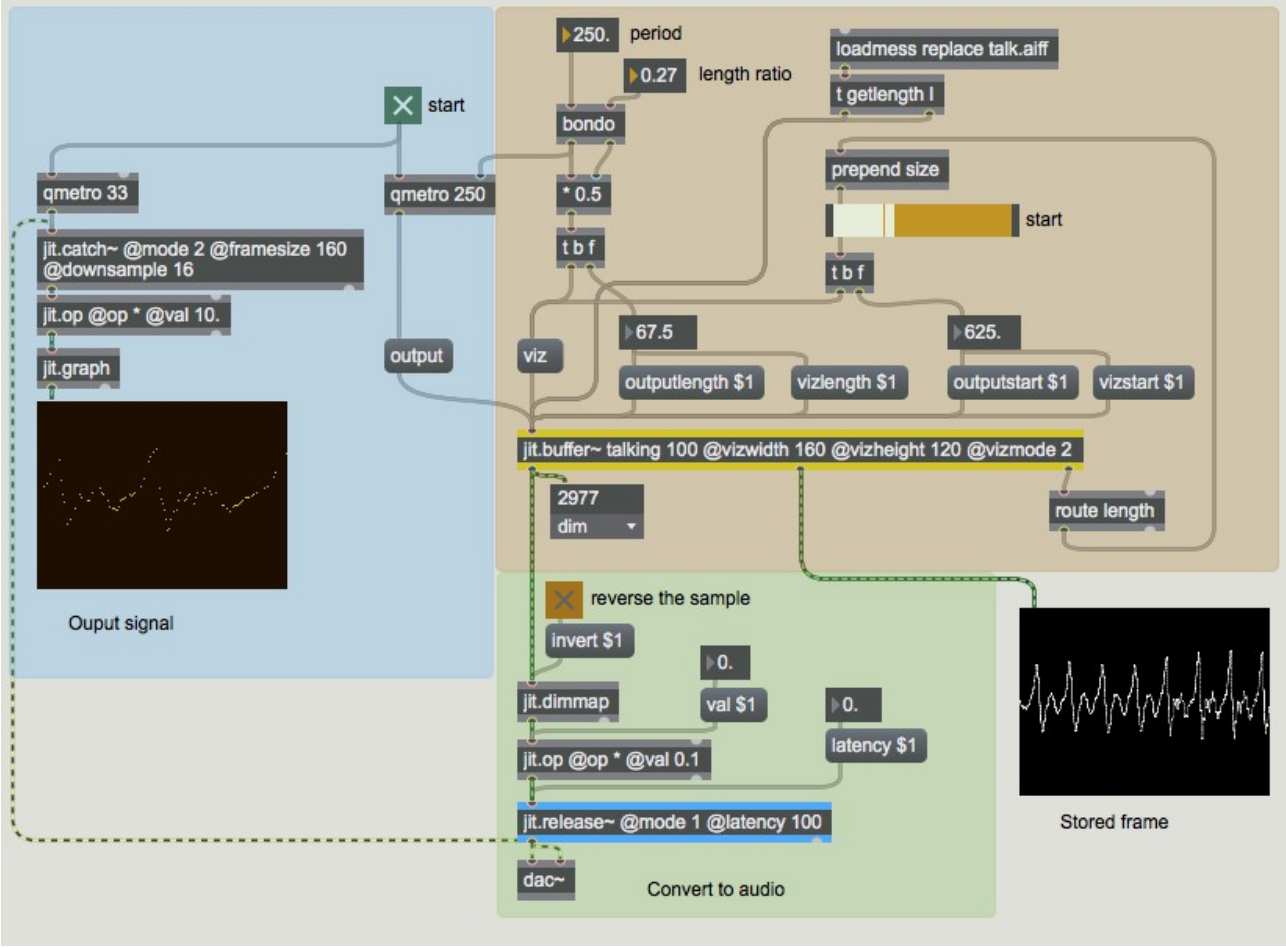
• Arrêtez le *qmetro* et fermez le sub-patch. Ouvrez le patch nommé **interpolating**.

L'objet jit.release~ peut fonctionner dans l'un des deux modes suivants: dans le **mode** standard **0**, il s'attend à être alimenté directement avec des échantillons au taux audio - c'est-à-dire que pour un son de qualité CD, l'objet recevra en moyenne 44100 éléments **float32** chaque seconde et il placera ces échantillons directement dans les signaux. En **mode 1**, cependant, l'objet va interpoler dans son tampon interne en fonction du nombre d'échantillons qu'il a stockés. Si le temps de lecture des échantillons stockés est inférieur à la longueur de l'attribut de latence, *jit.release~* jouera les échantillons plus lentement. Si l'objet cesse complètement de recevoir des données, la lecture s'arrêtera d'une manière analogue à la pression sur le bouton d'arrêt d'un tourne-disque. D'un autre

côté, s'il y a plus d'échantillons que nécessaire dans le tampon interne, *jit.release~* jouera les échantillons plus rapidement.

On peut utiliser cette fonctionnalité pour générer du son directement à partir d'un réseau événementiel. Le sub-patch **interpolating release~** nous montre un moyen de le faire. Les données qui pilotent l'objet **jit.release~** dans ce sub-patch reçoivent des données d'un objet *jit.buffer~*, ce qui nous permet d'extraire des données d'échantillons audio sous forme matricielle en utilisant des messages vers l'objet MSP *buffer~*. L'objet *jit.buffer~* est essentiellement un enveloppeur Jitter autour d'un objet *buffer~* ordinaire; *jit.buffer ~* acceptera tous les messages que *buffer~* accepte, permet l'obtention et le paramétrage de données *buffer~* sous forme de matrice, et fournit également quelques fonctions efficaces pour visualiser les données de forme d'onde en deux dimensions, que nous utilisons dans ce patch dans l' objet *jit. pwindow* à droite. Au moment du chargement, cet objet *jit.buffer~* a chargé les données dans le fichier audio **talk.aiff**.

• Activez le *toggle* connecté à l'objet *qmetro* dans le sub-patch. Jouez avec les boîtes de *nombres* intitulées ***période*** et ***length ratio*** et l'objet *slider* intitulé ***start***.




*Lecture de données audio interpolées à partir d'un fichier son dans Jitter.*

La période de *qmetro* détermine la fréquence à laquelle les données de l'objet *jit.buffer~* sont envoyées vers l'objet *jit.release~*. La construction en haut maintient un ratio entre la période et le nombre d'échantillons qui sont sortis de *jit.buffer~*. En expérimentant avec le point de départ, le ratio de longueur et la période de sortie, vous aurez une idée des types de sons qui sont possibles dans ce mode de *jit.release~*.

## Résumé

Dans ce didacticiel, nous avons présenté *jit.catch~*, *jit.graph*, *jit.release~* et *jit.buffer~* comme des objets permettant de stocker, de visualiser, sortir et lire des données de signaux MSP sous forme de matrices Jitter. Les objets *jit.catch~* et *jit.release~* nous permettent de transformer les signaux MSP en matrices Jitter à un rythme événementiel, et vice versa. L'objet *jit.graph* fournit un certain nombre de moyens de visualiser des données matricielles unidimensionnelles dans un graphique à deux dimensions, ce qui le rend idéal pour la visualisation de données audio. L'objet *jit.buffer~* agit de manière similaire à l'objet MSP *buffer~*, nous permettant de charger des données audio directement dans une matrice Jitter à partir d'un fichier son.