POLYSPRING : A PYTHON TOOLBOX TO MANIPULATE 2-D SOUND DATABASE REPRESENTATIONS

Victor Paredes, Frederic Bevilacqua
UMR STMS IRCAM/CNRS/SU
victor.paredes@ircam.fr
frederic.bevilacqua@ircam.fr

Jules Françoise
Université Paris-Saclay, CNRS, LISN, France
jules.francoise@lisn.upsaclay.fr

ABSTRACT

Corpus-based concatenative sound synthesis is typically used with a projection or reduction of the sound parameter space to a 2-dimensional map where sound segments form point clouds that can be visualized and explored with a mouse or a touch interfaces. While this is satisfying with visual feedback, where possibly sparse and heterogeneous sound spaces can be easily controlled, this remains challenging or impractical without visual feedback and using whole-body movements.

We present *polyspring*, a Python toolbox dedicated to manipulating the distribution of a set of points in a 2-Dimensional plane. This package implements an algorithm based on a spring network simulation that can redistribute points according to a density target within a given bounded region while preserving the initial order between points. We made several modifications and additions to the previously published *unispring* algorithm to allow for concurrently interacting with the dataset and manipulating the distribution in real time. The toolbox is open-source and can be used with Max/MSP. We also present different applications of this toolbox in movement-based sound interaction.

1. INTRODUCTION

Sample-based sound synthesis techniques, such as corpus-based concatenative synthesis, are particularly efficient for real-time multidimensional control of sound textures, which finds applications in music performance, gaming, or sound design. Many approaches adopt point-based representations of databases to facilitate navigation and continuous control, with examples including CataRT [1], earGram [2], AudioStellar [3], and FluCoMa [4]. Sound samples can be selected in real-time by navigating a parameter space, often represented as 2D 'sound clouds', the dimensions of which are either based on audio descriptors such as loudness, pitch, spectral centroid, or noisiness, or extracted using dimensionality reduction techniques. Such a visual representation of the sound space enables direct gestural sample selection using the mouse or touch gesture with pads. In this case, direct visual feedback is key to ensure efficient navigation in the sound space that might not be uniform [5, 6].

Copyright: © 2023 Victor Paredes et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

The control of such sound spaces can be challenging when no visual feedback is available for the performer. Movement-based sound control, with sensor-based or video-based movement tracking, represents exemplary use cases where the performer must navigate, using body movements, in the sound space without the help of any screen [7, 8]. In such scenarios, movement parameters should be mapped to the sound space parameters in a way that all the sound clouds can be reached and explored. As illustrated in Figure 1, this is generally not guaranteed using simple parameter scaling or rotation: the 2D sound map presents several clusters of different sizes and densities, as well as empty zones. As a result, it might be difficult or impossible for the performer to access and control all the sound space continuously, as certain movements might not trigger any specific sound sample.

This paper aims to propose a method that enables both the control of the points distribution, from homogenization to custom density function, and the setting of adaptable border geometries. The method presented here is based on the *unispring* algorithm introduced by Lallemand and Schwarz [9], with improvements at both theoretical and practical levels.

First, we present several theoretical contributions that simplify the use of the base physical model and facilitate the real-time manipulation of the sound map structure. In particular, we propose to estimate a key parameter from the dataset to provide users with a minimal number of control parameters, allowing them to use this tool without knowledge of the underlying physical models. Additionally, we formalized a simpler density manipulation technique using Gaussian mixtures as inputs, which allows users to manipulate the distribution of the sound clouds in real-time.

Second, we provide the community with an open-source Python package called *polyspring* that can be used in conjunction with several audio software. *Polyspring* enables the control of a sound dataset distribution starting from any type of 2D projection. It can be used to distribute the dataset uniformly while preserving neighborhood relations between points. A density function can be provided to further shape the final distribution of sounds. This distribution can also be restricted to custom boundaries (see figure 1). Additionally, a Python script and a Max patch communicating over OSC are provided to manipulate a sound dataset in the Max package *Mubu*. [1]

[1] MuBu download page: https://forum.ircam.fr/projects/detail/mubu/

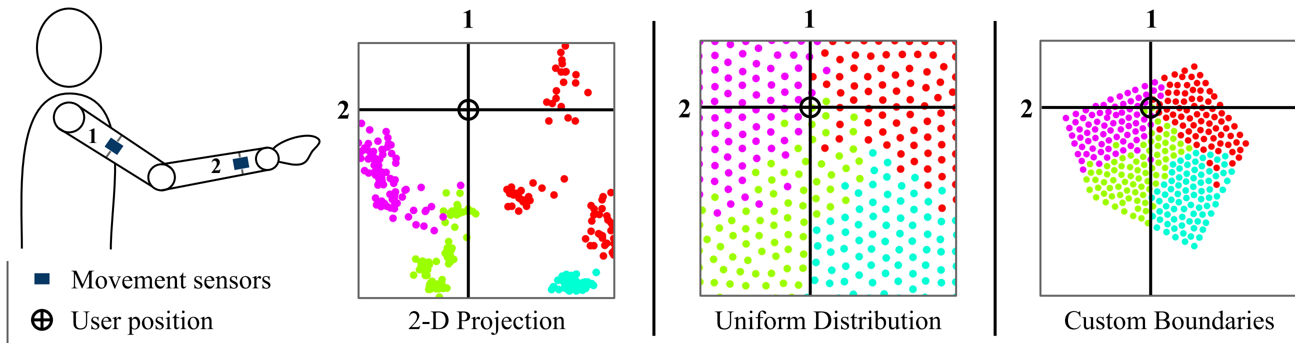

Figure 1. Exploration of a sound database using motion sensors mapped to the control space. Without visual feedback, the 2-D audio space might present empty space and tight clusters compared to the uniformly distributed space. The control space can be adapted to the user's motion range using a custom bounded region.

2. RELATED WORK

Several popular sound synthesis methods rely on analyzing and recombining databases of recorded sounds. Such methods involve the segmentation of audio recordings, their parametrization (for example, with audio descriptors), and their resynthesis using selection methods. Various implementations have been proposed, such as the real-time corpus-based concatenative synthesis system CataRT [1], used by several research groups and musicians [10–12], the tools developed in the Fluid Corpus Manipulation project (FluCoMa) [4, 13], earGram [2] or the AudioStellar system [3]. 2D representation of the sound parameterization greatly facilitates its direct manipulation by the performer through projection or dimensionality reduction from the possibly high dimension of the sound description. The relationship between the user gestural input and the selection of such 2-D reduced space can be formalized as a special case of the so-called gesture-sound mapping [14].

Over the years, many approaches have been proposed to design complex multidimensional mappings, using diverse methods such as physical models [15, 16], machine learning [17] or geometrical approaches [18]. In the latter approach, discrete points serve as control structures for the generation of continuous mappings, where several techniques can be used to define and transform the geometry of the intermediate mapping layer. Zbyszynski et al. [11] proposed a regression method (Neural Network Regression Model) to set the mapping between gestural data to the audio space. Nevertheless, with corpus-based approaches, the problem often consists in mapping continuous movements to a sound space that might be sparse and/or heterogeneous in terms of sound density, requiring novel approaches to the visualization or distribution of sound database.

Gerard Roma and colleagues made several contributions concerning the visualization and mapping of sound collections. In [19], they present a novel framework for sound space creation for interaction. Features are extracted from the sound corpus using a neural encoder that produces high-dimensionality features. A segmentation algorithm chops initial files into sound fragments based on a novelty algorithm. Eventually, each fragment is projected in a low dimensional space (2-D or 3-D) using a dimensionality reduction technique. The playability highly depends on the visual characteristics of the 2-D representation obtained from this last step. For Isomap and the Fruchterman-Reingold algorithm, the authors highlight a trade-off between perceptually meaningful clusters and good spatial occupation when dealing with the number of neighbors. They also experimented with Self-organizing Maps (SOMs), which is a mapping technique between high-dimensional data and a 2-D grid layout. Still, SOMs can assign multiple data points to the same output location, which is not a feature we want in our case.

More recently, Roma et al. produced a more general framework for visualization [20] where they evaluated several dimensionality reduction algorithms for music creation applications. They found the uniform manifold approximation and projection (UMAP) algorithm to be more suited for this use case. One of UMAP parameters is the number of neighbors for projected points. This parameter impacts the clustered aspect of the 2-D projection, therefore allowing for control on the spread of the points over the space. They presented a solution to de-clusterize points by mapping the output of a dimensionality reduction technique to a grid using the Hungarian algorithm. By oversampling the output grid, the initial distribution of points can be preserved while avoiding the overlapping of points. While useful to get a uniform distribution, this technique requires to define a grid matching the desired distribution and the right number of points, which can be difficult to generate for general cases (typically arrangements not initially following a grid).

3. THEORETICAL BACKGROUND

Polyspring builds upon several algorithms and methods to manipulate a set of points in a 2-dimensional plane. After introducing the base *unispring* algorithm initially proposed by Lallemand and Schwarz [9], we present several modifications and additions we made to this algorithm. Then, we describe a cost-effective interactive method to shape the density of the distribution.

3.1 Unispring Algorithm for Density Manipulation

The *unispring* algorithm [9] manipulates a 2-Dimensional set of points to distribute it across a given region either

uniformly or according to a user-defined distribution. The *unispring* algorithm is composed of two steps explained below: the pre-uniformization and uniformization as illustrated in Figure 2.

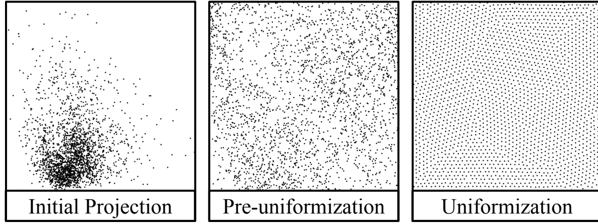

Figure 2. The *unispring* algorithm takes the 2D projection of a dataset and distributes it uniformly in the space using a mechanical analogy between a triangle mesh and a spring network. A first pre-uniformization step is used to spread the points in the space to make the simulation converge faster.

3.1.1 Pre-uniformization

The first step of the algorithm consists in scattering the initial distribution by performing a "pre-uniformization" step. It replaces the x (resp. y) coordinate of each point with the index of this coordinate in the sorted list of all x (resp. y) coordinates in the set. Note that these intermediate coordinates are scaled so that the initial range of the x-axis (resp. y-axis) is preserved. This intermediate distribution fills the space while preserving the order between points from the initial projection. This first step will help the physical algorithm to converge faster.

3.1.2 Uniformization

The second step is based on the *distmesh* algorithm [21], introduced by Persson and Strang, that is used to generate meshes using a simple mechanical analogy between a triangular mesh and a 2D spring network. The triangle mesh is created using a Delaunay triangulation: each point is linked with its neighbors, forming triangles whose circumcircles do not contain any point. The physical model is created by considering each connection between vertices as a spring. The obtained model is a network of joints linked by springs. Only the repulsive actions of the springs are considered. Allowing for attractive forces would require complex simulation schemes to ensure stability in every case. Therefore, a point is subjected to a force whose direction and amplitude depend on the distance from connected neighbors. Each connected point applies a force in the direction of the connection with an amplitude proportional to the length of the connection:

$$f = \begin{cases} k(l_0 - l) & \text{if } l < l_0 \\ 0 & \text{if } l \geq l_0 \end{cases} \tag{1}$$

with l the current length of the spring, l_0 the rest length, and k the spring stiffness.

The position at a given step is solved using a forward Euler method. To end the simulation, a stop criterion is checked at every step: if for all points the displacement at a step is under a certain threshold, then the simulation stops. Further details on how the algorithm is implemented can be found in the original *distmesh* article [21].

The region in which the points are distributed consists of boundaries that cannot be crossed. If a point moves past a boundary, it is brought back to the closest point on the boundary. The region is defined by the user as a *signed distance function* which returns the distance to the closest point on the boundary, positive if inside the region or negative if outside. We shall see in Section 4.1.1, that in our implementation we modified this definition to simplify user interaction.

The target density is defined by the *element size function h(x, y)*. For a uniform target density, all springs have the same rest length so that the final distance between points is the same everywhere. By specifying different rest lengths, we can modify the density in certain parts of the region. It is not required to specify the exact rest length of the springs, *h(x, y)* only gives the relative distribution over the domain. The actual rest length of a spring whose center is at a (x, y) coordinate is then calculated using the following equation:

$$l_0(x, y) = \sqrt{\frac{\sum l_i^2}{\sum h(x_i, y_i)^2}} * h(x, y). \tag{2}$$

3.2 Additions and Modifications to the Unispring Algorithm

The aim of the *polyspring* toolbox is to allow users to manipulate a 2-D point distribution easily, without having to tweak the parameters related to the underlying physical model. We made the following additions and modifications to the *Unispring* algorithm for this purpose.

3.2.1 Calculation of the Springs Rest Length for a Uniform Distribution

The results of the physical model used to move the points highly depend on the choice of the springs' rest length since the final distance between points will be, approximately, the rest length. So it must be chosen according to the target distribution, the number of points and the region dimensions. We deduce an estimate of the target distance l_0^{uni} between points for a uniform distribution from the region area and the number of points.

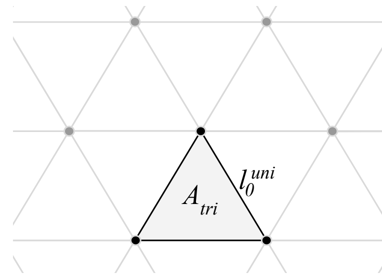

Figure 3. For a uniform distribution, points should be arranged so that they form equilateral triangles. Therefore, the spring rest length l_0^{uni} must be chosen to verify the target density.

Let a uniform target distribution of density $d = \frac{N}{A}$ with N points contained in a region of area A. Since all pairs of points linked by the Delaunay triangulation must be at the same distance, they form equilateral triangles with their neighbors, as shown in figure 3. Since each point is the vertex of 6 different triangles, we can write the density of points in a single triangle as:

$$d = \frac{3}{6A_{tri}} \tag{3}$$

with $A_{tri} = (l_0^{uni})^2\frac{\sqrt{3}}{4}$, the area of a triangle. Resulting in:

$$l_0^{uni} = \sqrt{\frac{2}{\sqrt{3}\frac{N}{A}}}. \tag{4}$$

This calculation does not take into account the points that are on the borders of the region: a point on the border is the vertex of fewer than 6 triangles. Therefore, the density in Eq. (3) is underestimated in these cases, resulting in an overestimation of the rest length. However, as stated in [21], "it is important that most of the bars give repulsive forces to help the points spread out across the whole geometry, [...] which can be achieved by choosing l_0 slightly larger than the length we actually desire".

3.2.2 User-defined Distribution: Non-uniform Rest Length

The *distmesh* algorithm introduces an *element size function* $h(x, y)$ that is used to specify the relative density in various parts of the region through a non-uniform rest length of the springs in the physical model. This method is implemented in the toolbox, albeit with some differences in the function that the user specifies. Instead of defining an *element size function*, we introduce a *density function* ρ, such that $h(x, y) = \frac{1}{\rho(x,y)}$, that specifies the relative density of points in different parts of the region. We made this change to focus the interaction on the density of points at certain coordinates rather than the distance between points. The *element size function* derives directly from the analogy of the spring network, the elements refering to the connections made by the Delauney Triangulation. Like h, this function is relative and does not need to sum to any particular value across the region. A location where $\rho = 2$ will be twice as dense as if $\rho = 1$.

The rest length of a spring whose center is at the coordinates (x, y) is calculated using the uniform rest length l_0^{uni} that we obtain using Eq. (4) and the ρ function that is provided by the user:

$$l_0(x, y) = l_0^{uni}.\sqrt{\frac{N_c}{\sum_{i=1}^{N_c}\left(\frac{1}{\rho(x_i,y_i)}\right)^2}}.\frac{1}{\rho(x, y)} \tag{5}$$

where N_c is the number of connections and (x_i, y_i) is the center of the connection i. Note that in Eq. (5), the sum of l_i^2 from Eq. (2) is replaced by the average rest length l_0^{uni} from Eq. (4) which is a better estimate of the average rest length when there are large empty spaces in the initial distribution.

The user can specify any density function $\rho(x, y)$ as long as it verifies that $\forall(x, y) \in \mathbb{R}^2, \rho(x, y) > 0$. For instance in figure 7, $\rho(x, y) = x + y + 1$. Which means that, since $\rho(0, 0) = 1$, $\rho(1, 0) = \rho(0, 1) = 2$ and $\rho(1, 1) = 3$, the distribution is three times denser at $(1, 1)$, and two times denser at $(1, 0)$ or $(0, 1)$), than at $(0, 0)$. The resulting distribution is displayed figure 7. Note that in this case coordinates have been scaled to $[0, 1]$.

3.3 Gaussian Attractors: Manipulating the Distribution during Interaction

Originally, *unispring* was designed to obtain a "static" representation of the database, as stated by the authors: "we mean that the interface is determined once by the user and does not require further adjustments in the course of the interaction process" [9]. Our goal in designing *polyspring* was to let users manipulate the distribution and interact with it concurrently. While it is possible to interact with the sound corpus during the mass-spring simulation process, the speed of the simulation highly depends on the number of points and cannot be anticipated easily.

We created a method to interact with the point distribution using a force field that can move points away from their stable position. This force field can be specified by users through a mixture of Gaussians. After uniformization using the spring network method, the points are pushed away from their initial positions. The gradient of the force field directs the displacement vector of a point, and its norm is equal to the value of the field at the point coordinates. The field is scaled between 0 and l_0^{uni} so that the points displacements are bounded. An example of a distribution obtained with a mixture of three Gaussians is given in Figure 8. While this model allows for other inputs than Gaussian mixtures, it is difficult to anticipate its behavior without understanding the process behind the creation of the displacement vector. So we restricted the input possibilities to make it more accessible while allowing for complex final distributions.

4. TOOLBOX DESCRIPTION

The *polyspring* toolbox implements a modified *unispring* algorithm with its extensions described above. This toolbox consists of a Python package that provides classes to manipulate the distribution of a set of points inside a given region. It also includes a Cycling' 74 Max package that enables its use with the MuBu [2] library. All the scripts and patches are provided in a repository [3] along with a video demonstrating several features of the toolbox.

4.1 Implementation

4.1.1 Polyspring Implementation in Python

The models described in section 3 are implemented with Python using standard packages for scientific computing, except for the region definition. All Matrix operations are made using *numpy*, and we use the Delauney triangulation from the *scipy.spatial* submodule. The gaussian attractors

[2] https://forum.ircam.fr/projects/detail/mubu
[3] https://github.com/ircam-ismm/polyspring

use *numpy.gradient* to compute the gradient of the force field over a grid and the *scipy.interpolate.griddata* to interpolate the gradient at each datapoint from the gradient grid.

In the base *distmesh* algorithm, the main difficulty for region management user-wise is to create the *signed distance function* that returns the distance from a point to the closest boundary point (positive if inside, negative if outside). While it allows for any region to be defined, it is complicated to create in most cases.

We decided to use the *shapely* package [4], a set of classes aimed at set-theoretic analyses and manipulation of planar feature that uses functions from *GEOS*, a C/C++ library for computational geometry. *Shapely* provides a polygon class with an highly efficient point-in-polygon (PIP) test and closest points calculation. Using this package, any region can be easily defined as a simple polygon or more complex regions with holes using set operations. In our implementation, the region is provided by the user as a shapely *Polygon* or *MultiPolygon* object.

4.1.2 Using Polyspring in Max

We created a Max patch and a Python script to interface *polyspring* with a MuBu container that we used to create the example described above. The Max object can be used to distribute any set of points contained in a MuBu track.

The Max patcher and the Python script communicate using OSC (Open Sound Control) over UDP (User Datagram Protocol). On the Max side, data are sent and received using the base Max UDP implementation (*udpsend* and *udpreceive* objects). On the Python side, the communication is managed by the *python-osc* package [5] that allows for an OSC address $\leftrightarrow$ callback matching system. Thus, the communication consists of packets sent over UDP from one process to another. If both processes run on the same machine, the packets are sent to the local host IP address (127.0.0.1), and each process uses different receive ports (8011 for Max and 8012 for Python by default).

4.2 Features

We now detail the main features of *polyspring*, illustrated with an example built using the Max implementation for CataRT-MuBu [6]. A screenshot of the Max patcher is shown in Figure 4. The green and yellow parts (”Initialize Corpus” and ”Choose Representation”) are dedicated to the creation of the corpus and the initial projection, while the blue part (”Shape Distribution”) uses *polyspring* to allow for manipulating the sound fragment distribution. The visualizer on the right side displays the current distribution in a square. Note that it is automatically scaled to the initial distribution range on both axes and does not change its limits unless the descriptors are modified.

4.2.1 Initialization and uniformization

In our example, the corpus is created using CataRT’s automated segmentation and analysis process. The user imports one or more sound files that are first segmented (using for example a fixed segment length or onsets), then analyzed with a series of N audio descriptors. Then, the user can choose two audio descriptors used to visualize all the sound segments as points in a 2D map, which corresponds technically to the projection of the 7-D audio descriptors space to a 2-D plane. All of this is implemented using the externals provided in the *MuBu* package.

The interest of *polyspring* lies in the possibility of modifying the distribution of points representing each sound segment. By default, the region is set as the non-oriented bounding box of the set of points, and the density function is set to 1. When pressing the ”Distribute” button, the points get uniformly distributed across this region. Since the visualizer is scaled to the initial range in both directions, it is equivalent to the default region, and the points fill the whole visualization space. The projection of a sound corpus in a descriptor space (spectral centroid and periodicity), before and after uniformization, is represented in Figure 5.

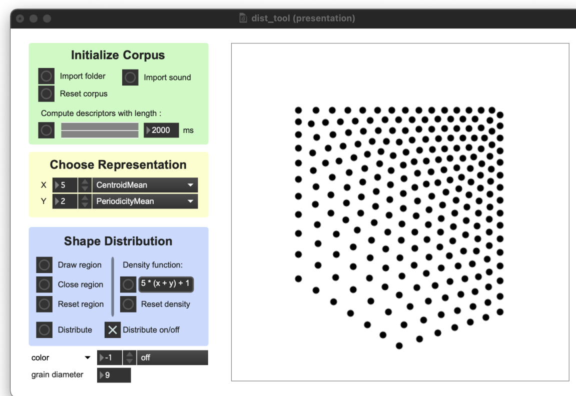

Figure 4. Screenshot of a Max patch demonstrating *polyspring* capabilities. The user can draw custom regions directly on the visualizer and type any density function of x and y such as $\rho(x, y) = 5(x + y) + 1$.

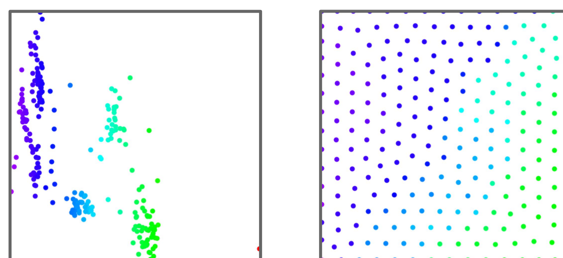

Figure 5. Example of basic uniformization of a sound corpus. The left scatterplot represents the projection of the segmented corpus over two descriptors: the spectral centroid and the periodicity. The scatterplot on the right depicts the representation resulting from a uniformization of the control space.

[4] shapely documentation
[5] python-osc reference
[6] https://forum.ircam.fr/projects/detail/catart-mubu

4.2.2 *Custom region*

Users can specify a custom region by drawing a polygon on the visualization space. After clicking the "Draw region" button, they can click on the visualizer to specify each vertex of the polygon sequentially. Clicking the "Close region" button will close the polygon by linking the last defined vertex with the first one. The uniformization algorithm then distributes points inside this region, as illustrated in Figure 6.

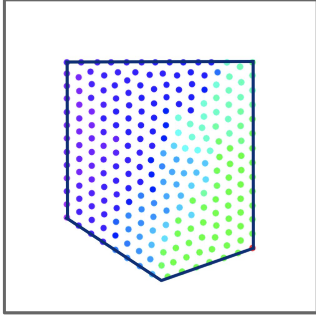

Figure 6. Example of uniformization of a sound corpus over a user-defined polygonal region.

Since the region definition relies on the package *shapely*, any polygon that can be created in shapely can be used to shape the distribution.

4.2.3 *Custom density*

The user can define the density function ρ as a function of x and y (See section 3.2.2). For instance in figure 7, $\rho(x, y) = x + y + 1$. The user can type any function existing in the *numpy* package by specifying "np." before the function name (eg. *np.exp, np.sqrt*...).

The Python class accepts any density function that takes two coordinates as arguments and returns a float indicating the relative density at this position. The example makes use of a formal expression definition but we can also input a function obtained from a Kernel Density Estimation (KDE).

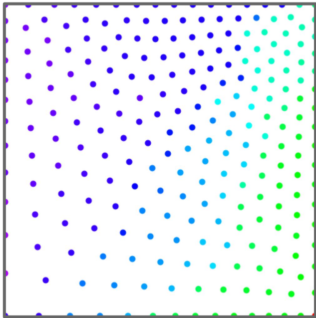

Figure 7. Example of a distribution obtained with a user-defined density function $\rho = x + y + 1$

Real-time interactions with the distribution, based on Gaussian attractors, are presented in a separate window as three configurable 2-D gaussian functions exposing the gaussian parameters: the center (μ_x, μ_y), the spread (σ_x, σ_y) and the angle θ. Figure 8 shows the Max window with the chosen parameters and the resulting distribution. Since the points are attracted from their original position by different Gaussians spread around the space, it can create empty spaces in the map. This distribution is not equivalent to what would be obtained by providing the gaussian mixture as the density function ρ in the spring network model, which is significantly slower to compute – and therefore not appropriate for real-time manipulation. For example, this method takes less than $30ms$ compared to several seconds with the spring network. However, the resulting distribution does not guarantee the complete coverage of the space and is more difficult to anticipate. This is complementary to the density definition using the ρ function, which is slower but affords more control on the final distribution.

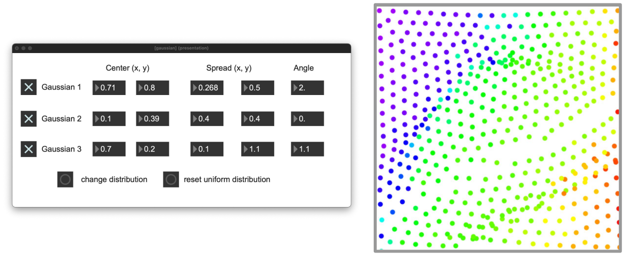

Figure 8. A distribution made using 3 gaussian attractors. The parameters for each gaussian are defined by the user.

5. EXAMPLES

In this section, we describe two case studies in which we used the Max implementation of *polyspring* to design movement-based interaction with concatenative synthesis using CataRT. Both applications, to the design of a lab experiment and for a dance performance, required sound spaces to be carefully adjusted to specific spaces.

5.1 Sound Exploration with Motion Sensors Attached to the Arm

In the context of experimental studies on sensorimotor learning, we are currently using motion sensors strapped to the arms to control a textural sound space without visual feedback, as described in Figure 1. Inspired by the work of Van Vugt and Ostry [8], we study the exploration of such continuous audiomotor space, which necessitates continuous sonic output to movements. As shown in Figure 1, the 2-D sound space is neither continuous nor homogeneous. To make the sound exploration practical, we used *polyspring* to distribute the sounds equally over the whole control space, thus suppressing silent zones and ensuring that the entire movement space is mapped to a unique sound output. Each sound segment takes the same area in the control space, and the density of points was the same everywhere, making the sound synthesis behave identically over the whole space. This enables users to explore the sound space with their arm movement without any visual feedback. Qualitative reports from the users confirmed that such uniformized textural sound space was significantly easier to explore and learn compared to the 'raw' sound cloud distributions.

5.2 Sympoiesis performance

The toolbox helped design corpora for the sound design of the dance performance *Sympoiesis* [7] by Laurane Le Goff.

5.2.1 Dance Sound interactive setup

In the *Sympoiesis* performance, four dancers were equipped with a motion sensor strapped to their right arm. Two orientation angles were mapped to the x-axis and y-axis of a CataRT sound space independently for each dancer. For each dancer, we created a custom sound space with approximately 200 sound fragments. Using *polyspring*, we distributed the sound corpus of each dancer uniformly in a different quarter of the space as represented Figure 9. In these conditions, three-quarters of the space was empty, and only a quarter contained sound. The position of the sounds changed before each performance, such that they did not know in advance where the sound segments would be located. At one point in the performance, the sensors were turned on, and the dancers started exploring a designed audio space that can be reached within a specific range of arms angles, which they must discover by improvising. For this performance, the partial coverage of the sound space was a desired and controlled property to bring an exploratory side to the dance. The fine control of the shape and the density of this audio space, as well as their real-time adaptation, was thus made possible by the toolbox.

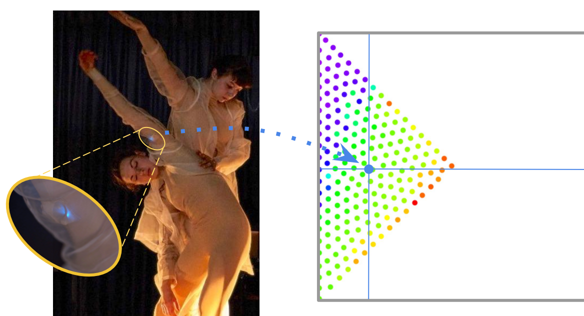

Figure 9. During the *Sympoiesis* performance, dancers were equipped with a motion sensor embedded in their costume. The sensor orientation was mapped to a 2-D CataRT space with sound points located only on a quarter of the space. *Performance picture credit: Younguk Choi*

5.2.2 Music Generation with Slime Molds

The second part of the *Sympoiesis* performance was centered around slime molds, a unicellular organism whose growing behaviors have been thoroughly studied [22]. For this performance, we wanted to use slime mold growth video recordings to control sound synthesis. A dataset of several hours of field recording in a laboratory has been scattered in a *CataRT* space. Since the videos consisted of upper views of a Petri dish, it was interesting to restrict the sound grains to a circle using *polyspring* to be able to superimpose them, as shown in Figure 10. The uniform distribution also ensured that the sound could be triggered everywhere in the circular region.

[7] Laurane Le Goff Sympoiesis project website

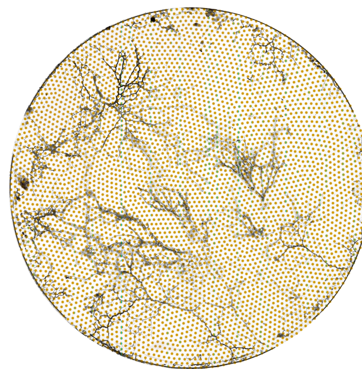

Figure 10. Picture of Slime Molds after several days of growth in a petri dish superimposed on a CataRT space of 4796 points uniformly distributed in a circular region. *Slime molds picture credit: Laurane Le Goff*

6. DISCUSSION AND CONCLUSION

We presented the theoretical basis and the Python implementation of the *polyspring* toolbox. It allows for the design of sound corpora where 2-D sound clouds are distributed over a specific space while maintaining local neighborhood relationships between sound parameters. We reported recent applications of this method in movement-based sound control, both for research and artistic endeavors. This opens new possibilities by enabling more control, for example, on the CataRT 2D representation. Beyond the examples we described, other use cases might take advantage of the toolkit, such as the one proposed by Jensenius et al., where the position of a musician on stage, tracked by a camera, is used to control a CataRT space [7]. With a simple one-to-one mapping between the location on the floor and the 2D sound parameters (centroid and periodicity), the performer triggered segmented violin sounds. It is easy to imagine an extension of such a paradigm with the possibility offered by *polyspring*: other sound parameters could be easily explored while assuring that all the sound cloud points are well distributed in a performance space of any geometrical form.

Polyspring is sufficiently generic to be used in a variety of applications. However, our main use cases always made use of the MuBu package. Communicating between Max and Python through OSC is not optimal in terms of performance. To improve this, we are considering implementing this toolbox as a MuBu external that would run in the Max process and should be substantially faster than the Python implementation.

Finally, the *unispring* algorithm should be generalizable to 3-D or even higher-dimensional spaces. Allowing for mapping between several movement descriptors and sound space dimensions in the case of gesture control for instance. Uniformization over N dimensions would guarantee the presence of sound points in all directions, which is particularly interesting when exploring high-dimensional spaces that are difficult, if not impossible, to visualize. Delaunay triangulation can be performed in n-Dimensions, which is already implemented in the *scipy.spatial.Delaunay* function and the repulsive action between two points could therefore

be calculated using the n-D euclidean distance. However, ensuring the convergence of the generalized model is not trivial, the region borders would be harder to define and to keep the simulation stable with a forward Euler method would require a lot of experimenting with the parameters.

Acknowledgments

This work is partially funded by the Agence Nationale de la Recherche grant ANR-18-CE33-0002 ”Enabling Learnability in Embodied Movement Interaction” (ELEMENT). We thank Diemo Schwarz for his help in using CataRT and the Unispring algorithm, and Laurane Le Goff and the dancers/co-creators of the Sympoiesis performance for their contribution in the dance experiments.

7. REFERENCES

[1] D. Schwarz, G. Beller, B. Verbrugghe, and S. Britton, “Real-time corpus-based concatenative synthesis with catart,” in *DAFx*, 2006, pp. 279–282.

[2] G. Bernardes, “Interfacing sounds: Hierarchical audio-content morphologies for creative re-purposing in eargram,” in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2020.

[3] L. Garber, T. Ciccola, and J. Amusategui, “Audiostellar, an open source corpus-based musical instrument for latent sound structure discovery and sonic experimentation,” in *Proceedings of ICMC*, 2020.

[4] P. A. Tremblay, O. Green, G. Roma, and A. Harker, “From collections to corpora: Exploring sounds through fluid decomposition,” in *International Computer Music Conference and New York City Electroacoustic Music Festival*. International Computer Music Association, 2019, pp. 223–228.

[5] D. Schwarz, “The sound space as musical instrument: Playing corpus-based concatenative synthesis,” in *New Interfaces for Musical Expression (NIME)*, 2012, pp. 250–253.

[6] G. Coleman, “Mused: Navigating the personal sample library.” in *ICMC*, 2007.

[7] A. R. Jensenius and V. Johnson, “A video based analysis system for realtime control of concatenative sound synthesis and spatialisation,” in *Proceedings of the second Norwegian Artificial Intelligence Symposium*, 2010.

[8] F. T. van Vugt and D. J. Ostry, “Early stages of sensorimotor map acquisition: learning with free exploration, without active movement or global structure,” *Journal of Neurophysiology*, vol. 122, no. 4, pp. 1708–1720, Oct. 2019. [Online]. Available: https://www.physiology.org/doi/10.1152/jn.00429.2019

[9] I. Lallemand and D. Schwarz, “Interaction-optimized Sound Database Representation,” in *DAFx*, 2011, pp. 292–300.

[10] A. Einbond, D. Schwarz, R. Borghesi, and N. Schnell, “Introducing catoracle: Corpus-based concatenative improvisation with the audio oracle algorithm,” in *Proceedings of the international computer music conference*, 2016, pp. 140–146.

[11] M. Zbyszyński, B. Di Donato, F. G. Visi, and A. Tanaka, “Gesture-timbre space: Multidimensional feature mapping using machine learning and concatenative synthesis,” in *Perception, Representations, Image, Sound, Music: 14th International Symposium, CMMR 2019, Marseille, France, October 14–18, 2019, Revised Selected Papers 14*. Springer, 2021, pp. 600–622.

[12] D. Schwarz, “Corpus-based concatenative synthesis,” *IEEE signal processing magazine*, vol. 24, no. 2, pp. 92–104, 2007.

[13] P. A. Tremblay, G. Roma, and O. Green, “Digging it: Programmatic data mining as musicking,” in *International Computer Music Conference 2021*. International Computer Music Association, 2021, pp. 295–300.

[14] A. Hunt, M. M. Wanderley, and R. Kirk, “Towards a model for instrumental mapping in expert musical interaction,” in *ICMC*, 2000.

[15] A. Momeni and C. Henry, “Dynamic independent mapping layers for concurrent control of audio and video synthesis,” *Computer Music Journal*, vol. 30, no. 1, pp. 49–66, 2006.

[16] A. Johnston, L. Candy, and E. Edmonds, “Designing and evaluating virtual musical instruments: facilitating conversational user interaction,” *Design Studies*, vol. 29, no. 6, pp. 556–571, 2008.

[17] R. Fiebrink, P. R. Cook, and D. Trueman, “Human model evaluation in interactive supervised learning,” in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2011, pp. 147–156.

[18] D. Van Nort, M. M. Wanderley, and P. Depalle, “Mapping control structures for sound synthesis: Functional and topological perspectives,” *Computer Music Journal*, vol. 38, no. 3, pp. 6–22, 2014.

[19] G. Roma, O. Green, and P. A. Tremblay, “Adaptive mapping of sound collections for data-driven musical interfaces.” in *NIME*, 2019, pp. 313–318.

[20] G. Roma, A. Xambó, O. Green, and P. A. Tremblay, “A general framework for visualization of sound collections in musical interfaces,” *Applied Sciences*, vol. 11, no. 24, p. 11926, 2021.

[21] P.-O. Persson and G. Strang, “A Simple Mesh Generator in MATLAB,” *SIAM Review*, vol. 46, no. 2, pp. 329–345, Jan. 2004.

[22] J. T. Bonner, “Cellular slime molds,” in *Cellular Slime Molds*. Princeton University Press, 2015.