Applications of Blocked Signal Processing (BSP) in Pd

Frank Barknecht
Cologne,
Germany,
fbar@footils.org

Abstract

Sample processing in Pure Data generally is block-based, while control or message data are computed one by one. Block computation in Pd can be suspended or blocked to save CPU cycles. Such "blocked signals" can be used as an optimization technique for computation of control data. This paper explores possible applications for this "Blocked Signal Processing" (BSP) technique and presents a system for physical modelling and for feature extraction as examples.

Keywords

Pure Data, parallel computation, algorithmic composition, optimization, physical modelling, feature extraction

1 Introduction

Software for computer music and realtime synthesis has to deal with two competing requirements: It continuously has to compute audio samples at a rate specified by the underlying samplerate (like 44.1 kHz as "CD quality") and it has to deal with sporadic events, for instance note events coming from midi streams. In many computer music system, the events of such control streams are computed less often than the actual audio samples: In addition to the sample rate as a defining parameter of a music software a slower *control rate* was added: "The *control rate* is the speed at which significant changes in the sound synthesis process occur. For example, the control rate in the simplest program would be the rate at which the notes are played. [...] The idea of a control rate is possible because many parameters of a sound are 'slowly varying'."[1]

The separation of control and sample rate has been a part of computer music software since its early days: "Among the languages of the Music *N* family, Music IV and its derivatives (including Music 4C) are sample-oriented, whereas Music V and Cmusic are block-oriented. The Csound language is also block-oriented, since it updates synthesis parameters at a *control rate* set by users."[2]

The introduction of different rates for audio and control streams makes specific optimizations for each domain possible. For the sporadic events of control streams, that only happen rarely compared to the calculation of audio samples, redundant or unnecessary calculations can be omitted.

Audio data however can be computed in blocks of samples. Instead of computing every sample and then the next, several samples are computed in one go, which significantly reduces the overhead in systems based on "unit generators": "This is done to increase the efficiency of individual audio operations (such as Csound's unit generators and Max/MSP and Pd's tilde objects). Each unit generator or tilde object incurs overhead each time it is called, equal to perhaps twenty times the cost of computing one sample on average. If the block size is one, this means an overhead of 2,000%; if it is sixty-four (as in Pd by default), the overhead is only some 30%."[3]

Apart from avoiding the overhead of function-calls for each unit generator, blocked processing also can be implemented in a way that reduces the number of memory allocations necessary. For this, the block size has to be constant over a sufficient time.

But block computation also has a major disadvantage: It adds latency of at least one block duration. "Sample-oriented compilers are more flexible, since every aspect of the computation can change for any sample".[4]

Contrary to audio data control streams usually are not computed in blocks. As most con-

[1][Dodge and Jerse, 1985, p. 70]

[2][Roads, 1996, p. 801-802]

[3][Puckette, 2007, p. 63]

[4][Roads, 1996, p. 801]

trol events happen only sporadically there may not be enough data to fill a block that would be useful to compute. For instance midi notes may be produced only every quarter beat which at a tempo of 120 BPM would amount to only one note event every 500 milliseconds. Filling a block of 60 quarter notes would then take half a minute - no sane musician would accept a latency of that duration.

While this is an extreme and admittedly a bit silly example, the number of control events often is small enough to not let the overhead of calling the unit generators for every event affect the overall performance of the system.

But this is only valid, as long as the number of events to compute stays small. Especially in algorithmic composition, in simulations or similar cases the amount of data in a "score" can become very big. The overhead of control information now becomes a problem. Computation in blocks may yield a significant performance gain. The results of the control computations may still only be needed infrequently compared to the rate of audio signals. Some modern languages like LuaAV or ChucK are designed to deal with this problem right from the start. In ChucK, "the timing mechanism allows for the control rate to be fully throttled by the programmer - audio rates, control rates, and high-level musical timing are unified under the same timing mechanism."[5]

Pure Data was not designed with variable control rates in mind, but a peculiar feature of Pd can be exploited to do block computations on control data, confessedly in a limited way. Pd can suspend its own audio computations locally using the [switch~] object. This object can stop all sample computations inside of a Pd subpatch or canvas and restart it on demand. A common use is to activate parts of a Pd patch only when needed, for example to manage voices in a polyphonic synthesizer: inactive voices can be switched off when not in use to save CPU cycles.

A not so common use case for switched-off subpatches is introduced and explored in this paper. We use subpatches to perform block-computations at rates much lower than the audio samplerate. These computations will employ the unit generators originally intended to do audio signal computations. The computation rate is adjusted by suspending blocks locally. We will call this approach "Blocked Signal Processing" or BSP to have a catchy and short buzzword available.

[5][Wang and Cook, 2004]

2 Blocked Signal Processing

A very simple example will now show the basic principle of BSP in Pd, how it can optimize certain actions and how it compares to traditional message computations. The task solved by the following two Pd code examples is simple: Read 4096 numbers stored in a table "ORIG", add 0.5 to it and store the result in table "RESULT".

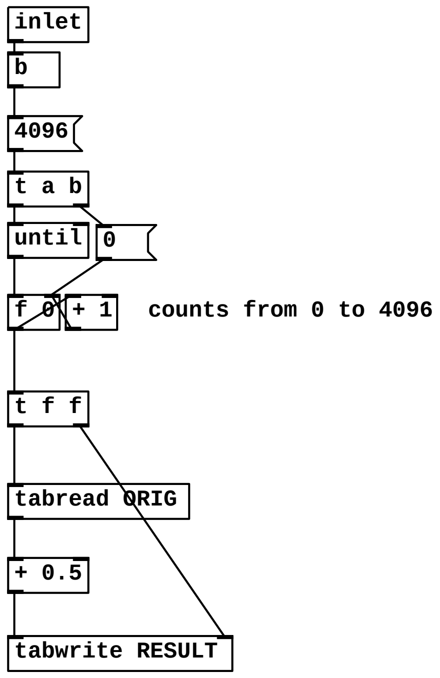


Figure 1: Transposing a table by 0.5 using message computations

Implemented with message objects, a counter is started by an incoming bang-message, its index number is used to read out the table data, a control-rate addition object adds 0.5 to it, then the value is stored.

The BSP implementation uses a pair of [tabreceive~] and [tabsend~] objects to constantly read and write the tables as a block of samples. The signal-rate addition object adds 0.5 in between. The [switch~ 4096 1 1] object resizes the blocksize to 4096 for this sub-canvas, so that the table-accessing objects can process that many samples. Additionally it switches off the DSP computation inside the subpatch at the beginning. So unless some messages to [switch~] switch on the computation, this part of the

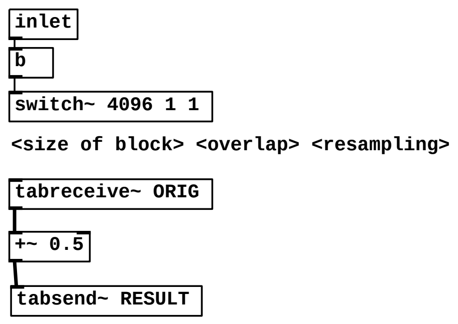


Figure 2: Transposing a table by 0.5 using BSP computations

patch doesn't consume any CPU resources. By sending a "bang"-message to [switch~], the subpatch is switched on for exactly one block of samples, then it's switched off again. During the time it is on, the actual computation happens, so the end result is the same as for the message version.

One small difference is important to note here: Probably for performance reasons Pd defers updates of the graphics in arrays or tables that are used with active [tabsend~] objects. Changes to the included data are only visible when the graph is closed and opened again.

The two implementations of the "transposer" don't show much difference in CPU usage. This is expected, as the simple calculations made here do not involve many objects so the overhead is negligible. Also both the BSP and the traditional method avoid any memory allocation by directly working on pre-allocated tables. When dealing with lists of numbers of arbitrary size, a common idiom in Pd is to build these lists by pre- or appending elements to existing lists stored in [list] objects.[6] For longer lists this can become a major cause of slowdown in Pd patches.

3 [physigs] - Physical Modelling implemented with BSP

The BSP technique should show more of its potential when applied to algorithms involving a bigger number of unit generators and more parallel tasks. We will now turn to such a use case and take a look at a physical modelling system based on spring-connected particles.

A particle simulation applies the Newtonian laws of mechanics to a simulation of point masses. The physically-inspired rules are used to calculate velocities and accelerations of point particles, that are defined by vectors describing their positions and impulses. Every particle also includes a force accumulator that holds any external forces applied to a particles. Forces, positions and impulses fully specify the current state of a particle system. Transitions from one state to a next are calculated in discrete steps: Usually a world clock is employed that advances one simulation step and initiates a new run of the physics calculations to find the new positions of the particles. As the same set of physical rules has to be applied to many particles, the problem is an ideal candidate for doing the calculations in blocks of particles.

With PMPD[7] and MSD[8], two implementations for this already exist as extensions to Pd (so called externals). For this paper a BSP-implementation of a particle system called "[physigs]" was written in Pd and is tested against the MSD and PMPD implementations. It's help file is shown in Fig. 7 at the end of this paper.

[physigs] is a particle simulation in two dimensions. In consists of a main Pd abstraction called [physigs] that can be called with a prefix-tag to make using the object several times in a single project possible. The state of the system is held in a number of Pd [table] objects for positions, velocities, masses, forces and a table that holds meta-data, currently only the mobility state of masses is watched: A mass can be mobile or fixed. A particle is identified by an integer number which is used as a lookup index into the state-holding tables. Particle 10 for example would hold its x-position in the 10th element of the table "pos-x", its y-position would be the 10th element in table "pos-y" and so on for tables "force-x", "force-y", "mobile" or "mass".

The size of these tables controls the size of the system: Tables of size 64 can control up to 64 particles. All 64 particles are computed regardless of particles actually being used. The table size can be selected when creating the [physigs] object.

In Figure 3 the calculations that advance the simulation one step are shown.

At the top left, the current x-positions and

[6]See the list-help.pd file in Pd's documentation for an example. This file also shows the opposite operation of serializing a list with [list split] which is slow as well.

[7][Henry, 2004]

[8][Montgermont, 2005]

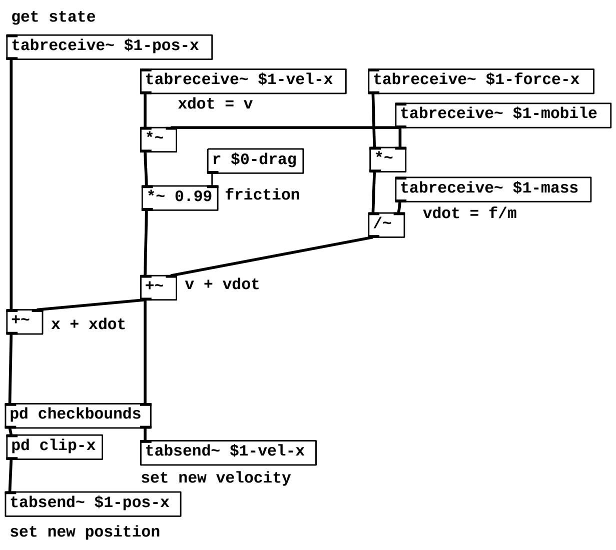


Figure 3: Advancing the world simulation one time step

x-velocities are read using [tabreceive~] objects. Respective [tabsend~] objects write the new positions back, after the physics laws have been applied. A table "force-x" holds an accumulation of all forces that have been applied to particles. The "mobile" table is 0 when a particle should be fixed, and 1 when it is mobile. Weights are stored in a "mass" table. Some friction is applied to fight instabilities and simulate air drag. The actual algorithm is rather simple: Any force on a particle is converted to an acceleration by dividing the mass, the acceleration is added to the velocity, which in turn is added to the position. The forces then get reset to zero, which is a step not shown in the figure. The subpatches "checkbounds" and "clip" restrict the possible positions to a configurable range and possibly flip the velocity to simulate bouncing of the world's ends. The same calculations are made for the y-coordinates as well.

The system is driven by an outside [metro] whose speed specifies the speed of the physical simulations. Metro periods in "haptic" ranges of 20 to 100 milliseconds are good choices. The [metro] then regularly generates bang-messages to switch on and off the DSP-signal computations in the subpatch holding the simulation step.

A similar construction in [physigs] calculates the forces of visco-elastic springs connecting two particles. These springs are defined by a number of tables again, where each spring indexes tables with its integer-ID. Any spring links two particles. The ID numbers of these two particles are held in two state tables called "link-m1" and "link-m2". Other state tables hold parameters like damping, stiffness, relaxed length and so on.

Springs generate forces that have to be accumulated into the tables holding the forces for each particle. This currently is realised in a separate step that doesn't use the BSP technique, because the "vanilla" version of Pd doesn't offer an object that can write into a table at a position specified by an audio signal. Miller Puckette intends to include such an object in future versions of Pd, the C-code for the object is rather trivial and it already exists as an external. Unfortunately falling back to traditional control-rate calculations in this part destroy many of the performance gains as we will see in the benchmark section.

3.1 Performance comparison

To compare the different implementations of a particle system, a test system has been devised that creates a configurable number of particles at random positions and with random mass and daisy-chains them with spring links. Such a performance test is part of the MSD distribution. Pd's built-in "Load meter" has been used to get rough CPU usage values. The world advances one step every 50 milliseconds (or 2 * 25 msec in [physigs] for link and mass computations each). Table 1 shows the results of several test runs.

It turned out, that [physigs] runs much slower than both MSD and PMPD when the control-rate calculations are used to distribute the spring-forces to each particle. Switching off this part of the patch in the BSP implementation will let [physigs] catch up to MSD and beat PMPD.

MSD doesn't have any overhead, as it computes the full simulation in just a single object. PMPD however has separate objects for each particle and link. In the example patch 4096 objects for the simulation participants alone are used. So here the overhead is significant as expected.

[physigs], especially when used without the control-rate workaround for a missing "write to table"-object, turns out to be a capable contender for physical modelling in Pd-vanilla or Pd-almost-vanilla environments. Note that these benchmarks are only meant to give a per-

Simulation Type	CPU Usage
MSD	5
physigs	30
physigs (w/o force distribution)	5
PMPD	16

Table 1: Simulation of 2048 particles and 2048 springs at metro-period of 50 ms using three different implementations

formance estimation and should not be taken as "hard numbers". But the author has successfully run [physigs] on the "RjDj" version of Pure Data on the iPhone with its much slower CPU compared to standard computers.

4 Feature extraction and analysis with BSP

[physigs] generates data-heavy control streams in a completely artificial manner. Similar amounts of data points have to be handled when analysing external audio coming in over the ADC (soundcard) and looking for certain features to guide a musical composition. Pitch tracking or onset detection work on single sound objects and must be prepared to react quickly. The [sigmund~] or [bonk~] objects that are part of Pd, thus run at audio rate. In this case, applying BSP would not be viable. But if a composer is interested in features on a "slower" time scale, BSP can be applied.

As an example lets consider the differences in time scale of physically moving between two rooms compared with playing two different notes on a clarinet. Changing rooms takes much more time than playing notes on instruments. To detect the clarinet's pitch changes, the software has to be constantly "alarmed" of the spectral content registered through the soundcard. However when trying to detect if a person holding a microphone changes rooms by analysing the spectral or reverberant characteristics of the two environments, spectral snapshots can be made and compared much less frequently than in the case of pitch detection.

The author has developed a set of BSP feature extraction (FE) objects mainly intended to be used in the RjDj version of Pd that runs on mobile devices. The timbreID collection of Pd externals by William Brent[9] served as an inspiration for these. Detecting "changed rooms" is a common need of composers working for RjDj. The analysis rate of the FE objects is configurable similar to the [physigs] object by adapting the speed of a [metro] object that blocks and activates them. Just as in the introductory example of a "table transposer" the objects work on shared table data and extract statistical features like centroid, mean, energy or flatness.

If the table to be analysed holds a magnitude spectrum, the extracted features describe the frequency content of the sound environment. Of course the objects may be used to acquire statistical analysis of tables holding other kinds of data as well.

As is well-known spectrum analysis with Fourier transformations is a relatively costly operation. In Pd it is already carried out in subpatches, so the analysis is a natural candidate to be "blocked" with the BSP technique. Of course the time resolution gets worse in this case, and overlapping analysis is not useful anymore, when there are pauses between adjoining runs of the transformation. But if all that is needed is a "snapshot" of an environment's spectral status, BSP-blocking is a viable way to save CPU resources.

Results of a spectral analysis written to a table can be analysed by several objects at the same time, forming feature vectors that can be post-processed for classification. The costly Fourier transformation has to be run only once for each vector.

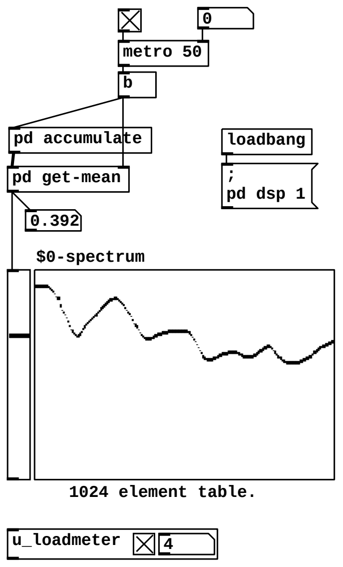


Figure 4: Calculating the arithmetic mean with BSP - main patch

[9][Brent, 2009]. Newer versions of timbreID include both audio- and control-rate versions of its analysis objects.

As an example for an FE-BSP object we will take a closer look at the calculation of the mean of a table. The arithmetic mean needs an accumulation of all table values, which then is divided by the table size.

While it would be possible to divide every single value by the table size before adding them, here the accumulation is calculated first, then a single division is made. This gives a minor speed-up, but here it should also illustrate the way, the execution order for BSP calculations can be controlled.

With Pd's normal message computations, [trigger] objects are used to specify a certain execution order. With signal objects the execution order can only be manipulated by laying out the operations into subpatches, that are connected by signal patchcords.[10] The connected subpatches are calculated by Pd's main scheduler in the order they are connected with objects earlier in the chain calculated first (contrasting Pd's depth first scheduling for messages).

In Figure 4 both subpatches are connected like this, so first every signal object in [pd accum] is run, then every signal object in [pd getmean]. Dummy signal inlets and outlets are included in both subpatches to allow making these order-forcing connections.

The accumulation of all values in a table can be coded using only a handful of objects: Using a one-pole recursive filter as supplied by Pd in the [rpole~] object with a coefficient of 1 will accumulate all data as long as its switched on. The filter is reset to zero after each run using the [bang~] object that outputs a bang after each DSP cycle.

The output of [rpole~] is written into a summing-table using [tabsend~]. The final position in this table will hold the accumulation value.

In the second subpatch (Fig.6) it's a [bang~] object that will transport this value to the division by N before it is reported to the outlet after the block calculation has completed. Mean calculation has a latency of one block of samples or blocksize divided by samplerate. Upsampling inside of subpatches to reduce this latency can be achieved by changing the third argument of the switch-objects or by sending it respective messages.

When implementing the FE objects, the lack of a Pd object to selectively write a value into a table at a certain position specified by an au-

[10][Puckette, 2007, p. 212-216]

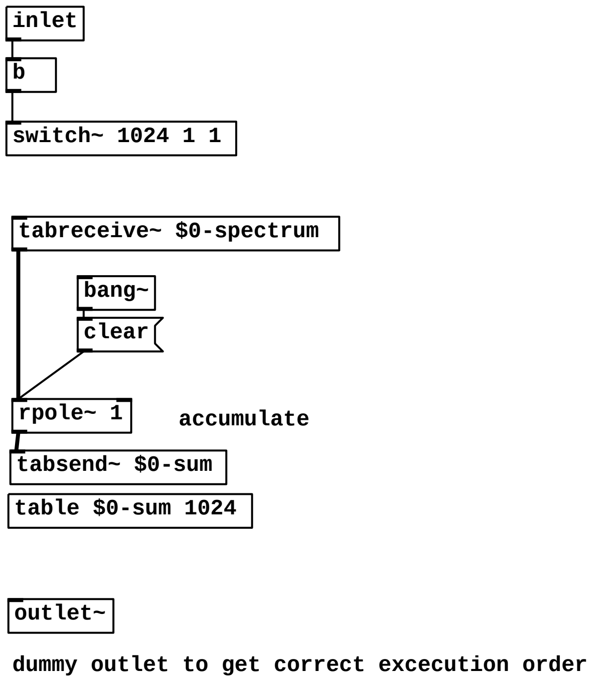


Figure 5: Subpatch "accumulate": Accumulating table values with a one-pole filter

dio signal was especially limiting. For example while finding local extrema (minima or maxima) in a table is easy by scanning through the table once and comparing two adjacent table values, the author hasn't yet found a way to efficiently store the locations of these extrema for later use.

Indexed table writing would also make another workaround or "hack" in the FE objects unnecessary: To calculate the geometric mean the product of all values in a table is needed, but there is no "vanilla"-way to reuse the result of the multiplication of previous samples again as it is with the one-pole filter that adds previous results (output values) back to its input: the next sample in a block. The workaround currently applied is to transform the multiplications to additions using logarithms.

5 Other Applications of BSP

The BSP technique can be applied to various other areas, where parallel, block-based processing is needed. Examples would be Cellular Automata, L-Systems or swarm/flock systems.

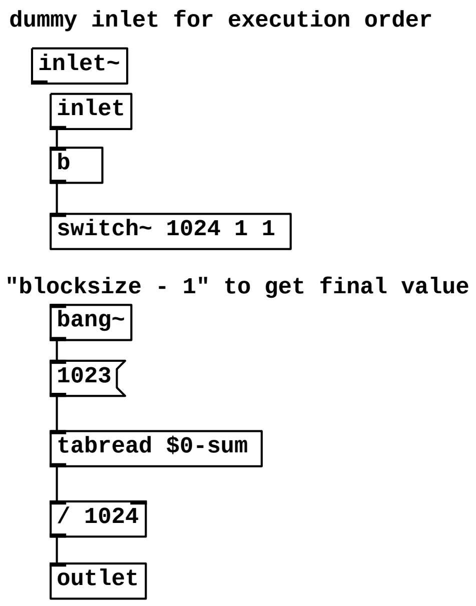

dummy inlet for execution order
inlet~
inlet
b
switch~ 1024 1 1
"blocksize - 1" to get final value
bang~
1023
tabread $0-sum
/ 1024
outlet

Figure 6: Subpatch "get-mean": Calculating the mean from the accumulated table and the table size.

But even for much simpler daily tasks in the life of a Pd user, BSP is worth a look. For example to copy the content of one table to another one, a blocked combination of tabplay~ and tabwrite~ can be used.

6 Limitations of BSP

As many optimization techniques, BSP has several limitations that have to be weighted against the possible performance gains. One important problem is execution order. Pd alternates audio and message computations. BSP however lives in a grey area between the audio signal and control computations. New results will be computed in the signal pass, where no other control calculations happen. It is not possible to trigger other control-events in the middle of a BSP-run.

The algorithms to be used in BSP cannot use recursion inside one block, because feedback of results computed at a later point in the DSP tree to an earlier point is not possible. The minimal feedback delay time in Pd is one block, other constructs result in "DSP loop detected"-errors in Pd. This limit is expected to complicate applying BSP in recursive algorithms like L-Systems. The inclusion of a suitable table-writing object in Pd vanilla as mentioned above could alleviate this problem a bit.

BSP deals only with numbers, so it's applicability to text processing is very limited, which affects the use of formal grammars. The symbols used in alphabets of L-Systems or similar systems based on rewrite rules have to be converted to numbers by implementing a translation map.

7 Conclusions and future work

BSP has been successfully applied to a simple physical simulation for this paper and a growing set of feature extraction objects. The [physigs] object will be refined and published under an open source license, while the feature extraction objects will become a part of the "rj" library developed for the RjDj application. While BSP is a powerful and so far under-used technique in Pd, it cannot magically transform Pd to become a true multiple-rate software. Music compilers like ChucK, SND-RT or LuaAV still deal with the competing demands of variable control rates in a cleaner and more flexible way.

8 Acknowledgements

Parts of the research for this paper was kindly supported by Reality Jockey Ltd./RjDj.

References

William Brent. 2009. Cepstral analysis tools for percussive timbre identification. In *Proceedings of the 3rd Pure Data Convention in Sao Paulo.*

Charles Dodge and Thomas A. Jerse. 1985. *Computer Music.* Schirmer Books, New York, NY, USA.

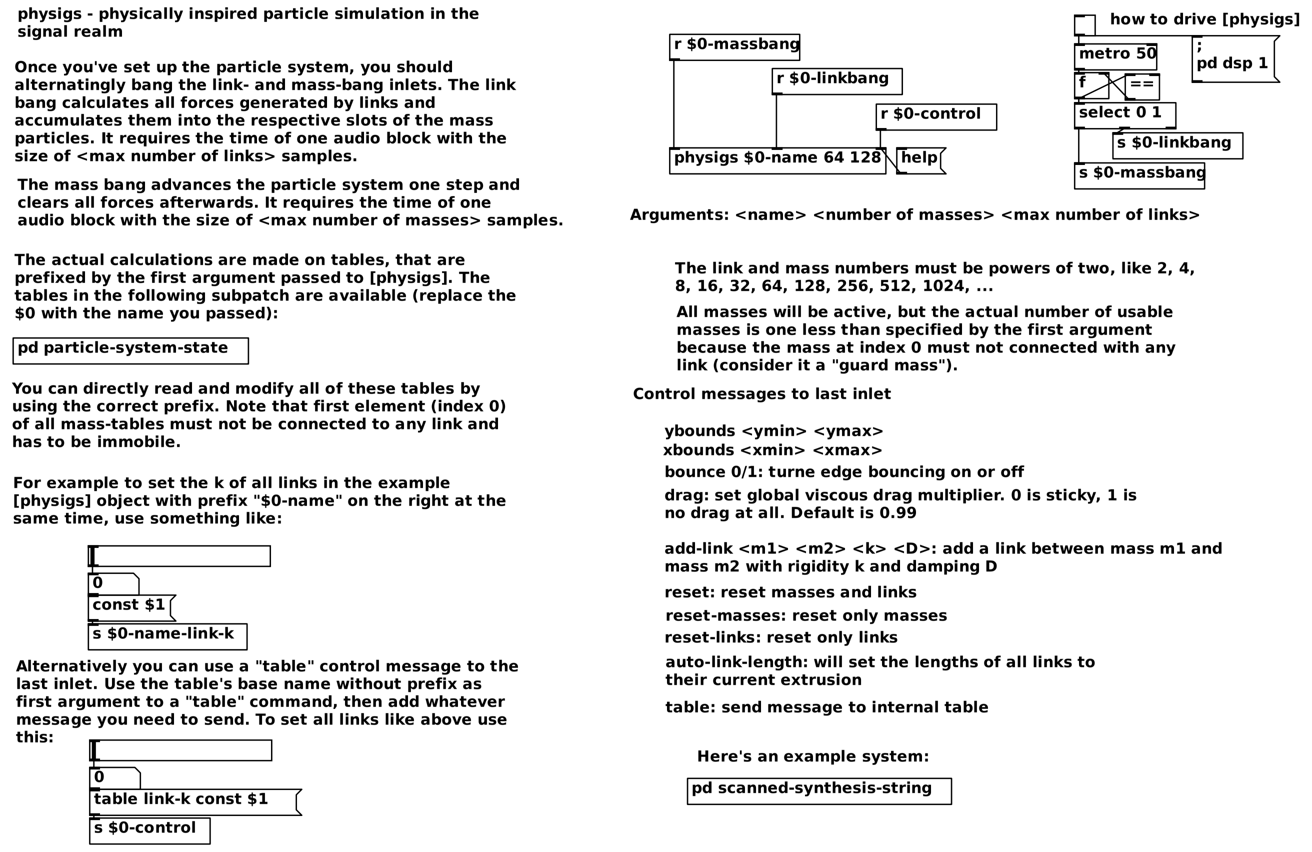
Cyrille Henry. 2004. Physical modeling for pure data (pmpd) and real time interaction with an audio synthesis. In *Sound and Music Computing '04.*

Nicolas Montgermont. 2005. Modeles physiques particulaires en environnement temps-reel: Application au controle des parametres de synthese. Master's thesis, Universite Pierre et Marie Curie.

Miller Puckette. 2007. *The Theory and Technique of Electronic Music.* World Scientific Press.

Curtis Roads. 1996. *The Computer Music Tutorial.* MIT Press, Cambridge, MA, USA.

Ge Wang and Perry R. Cook. 2004. On-the-fly programming: Using code as an expressive musical instrument. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 138–143.



physigs - physically inspired particle simulation in the signal realm

Once you've set up the particle system, you should alternatingly bang the link- and mass-bang inlets. The link bang calculates all forces generated by links and accumulates them into the respective slots of the mass particles. It requires the time of one audio block with the size of <max number of links> samples.

The mass bang advances the particle system one step and clears all forces afterwards. It requires the time of one audio block with the size of <max number of masses> samples.

The actual calculations are made on tables, that are prefixed by the first argument passed to [physigs]. The tables in the following subpatch are available (replace the $0 with the name you passed):

```
[pd particle-system-state]
```

You can directly read and modify all of these tables by using the correct prefix. Note that first element (index 0) of all mass-tables must not be connected to any link and has to be immobile.

For example to set the k of all links in the example [physigs] object with prefix "$0-name" on the right at the same time, use something like:

```
0
const $1
s $0-name-link-k
```

Alternatively you can use a "table" control message to the last inlet. Use the table's base name without prefix as first argument to a "table" command, then add whatever message you need to send. To set all links like above use this:

```
0
table link-k const $1
s $0-control
```

```
r $0-massbang
r $0-linkbang
r $0-control
physigs $0-name 64 128
help
```

how to drive [physigs]

```
metro 50
;
pd dsp 1
f
==
select 0 1
s $0-linkbang
s $0-massbang
```

Arguments: <name> <number of masses> <max number of links>

The link and mass numbers must be powers of two, like 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...

All masses will be active, but the actual number of usable masses is one less than specified by the first argument because the mass at index 0 must not connected with any link (consider it a "guard mass").

Control messages to last inlet

ybounds <ymin> <ymax>
xbounds <xmin> <xmax>
bounce 0/1: turne edge bouncing on or off
drag: set global viscous drag multiplier. 0 is sticky, 1 is no drag at all. Default is 0.99

add-link <m1> <m2> <k> <D>: add a link between mass m1 and mass m2 with rigidity k and damping D
reset: reset masses and links
reset-masses: reset only masses
reset-links: reset only links
auto-link-length: will set the lengths of all links to their current extrusion
table: send message to internal table

Here's an example system:

```
[pd scanned-synthesis-string]
```

Figure 7: Help-file for the [physigs] abstraction