Algorithm Animation with AGAT

Olivier Arsac Stéphane Dalmas Marc Gaëtano

INRIA Sophia Antipolis, projet SAFIR
2004 route des lucioles BP 93
06902 Sophia-Antipolis CEDEX
France
E-mail: stephane.dalmas@sophia.inria.fr
marc.gaetano@sophia.inria.fr
olivier.arsac@sophia.inria.fr

Abstract

In this paper, we describe *agat*, a new system for algorithm animation. *agat* differs from other such systems by its client-server architecture and the use of specialized programming language to describe the animation itself, independently of the animated program. It is thus much easier to animate an algorithm and to tune the animation. *agat* has been successfully used in our research group to investigate the behavior of complex algorithms like a binomial Groebner basis algorithm and an identification problem involving high degree polynomials for the design of microwave filters. We think that such a system can be a very useful tool for studying and understanding the complex algorithms that are routinely used and designed in computer algebra. *agat* runs on a wide variety of computers with the X window system.

1 Introduction

Algorithm animation is a powerful tool for exploring a program's behavior. It is used in various areas of computer science, such as teaching, design and analysis of algorithms, performance tuning. Algorithm animation systems provide a form of program visualization that deals with dynamic graphical displays of a program's operations. They offer many facilities for users to view and interact with an animated display of an algorithm, by giving ways to control the data given to algorithms and their execution through multiple views.

In computer algebra as in many applications, it is at least as important to observe a program's progress as it is to obtain a final result, if indeed there is any. For instance, the performance of algorithms to compute the Gröbner basis of an ideal of polynomials is sensitive to the ordering of variables. Animating the computation of such algorithms could indicate heuristics for deducing a good ordering from a given input.

The common approach to animating algorithms specified in high-level procedural languages was pioneered in BALSA in the early 1980's ([Bro88]). System such as BALSA were constrained by a lack of computational power for real-time two-dimensional graphics. In the mid- 1980's, systems like Animus ([Dui86]) used smooth transformations of 2D images to animate small examples. TANGO in the late 1980's ([Sta90]), and then XTANGO provide an elegant framework for specifying 2D animations. More recently, the Zeus system uses sound and 3D interactive graphics to reach higher levels of expressiveness ([Bro92], [BH91], [BN93]).

Although algorithm animation systems are becoming more powerful and easier for programmers to use, the task of using these tools to create effective dynamic visualizations of algorithms still involves excessive work. Libraries like XTANGO provide a powerful two-dimensional graphics package to build very sophisticated animations but require significant and sometimes rather complex additional programming to describe the animation. In systems like Zeus, the client code to be animated has to be implemented in a particular programming language (Modula 3 for Zeus); thus making the animation of an existing program a bit tedious to write.

The *agat* system described in this paper provides support for watching a program in action, through the use of a *stream processor*. The programmer animating a program provides a description of the program's important values using *streams*. Streams can be combined to make other streams. The output of a stream can be piped onto the input of a *graphical operator* to produce a display. Whenever an *interesting* value

is sent on a stream by the program, the value is propagated along all the combined streams, and each stream output updates its visual display appropriately. Another interesting feature of *agat* is the client-server architecture it is based upon : the animation server is a separate process and only a few lines of code have to be added to the source program in order to animate it. The animation itself is described within an animation file to be loaded in the stream processor (see figure 1). This architecture makes the description of the animation independent from the program to be animated, thus eliminating the need to rewrite an existing program in a special language.

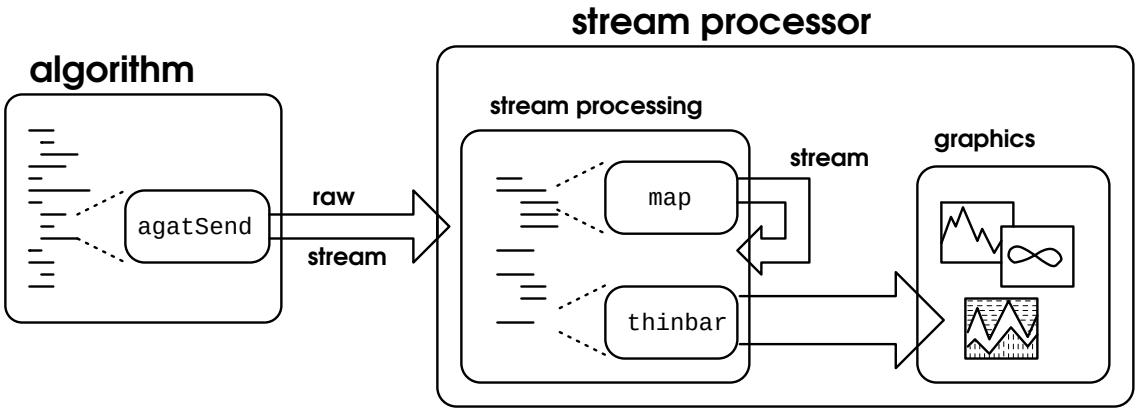

Figure 1: The *agat* architecture

2 Using AGAT

We will present the use, functionalities and architecture of *agat* on a very simple example, a program written in C to investigate the behavior of a random number generator.

2.1 Preparing a program for animation

To prepare a program for animation, the programmer uses the functions in the *agat* library to send the interesting data manipulated by the program on *streams*. He is free to use as many streams as he wants and to name them as suits him (a stream is just identified by a character string).

In our simple example, the program loops, repeatedly sending a new random number on a stream that we choose to name `i`. This is done by calling the `agatSendDouble` function (as the random numbers are of type `double`). Here is the instrumented code :

```
/* aleat.c, a pseudo random number generator, instrumented for agat */
#include <stdio.h>
#include <math.h>
#include "libAGAT.h"

double          drand48(void);

/* modify this function to change generators */
double          myRand(void)
{                                                                    10
  return drand48();
}

int             main(int argc, char **argv)
{
  int             i, j, nbn;
  double          v;
```

```
  nbn = atoi(argv[1]);
  for (i = 1; i <= nbn; i++) {
    v = myRand();
    agatSendDouble("i", v);
  }
}
```

The user needs then to link its program with the *agat* library, where the function `agatSendDouble` is defined (note that we had also included the `libASAP.h` header file where prototypes of the *agat* functions can be found). It can be done with a command line like `gcc -o aleat aleat.c -lAGAT`.

2.2 Animating a program

As the final step to obtain an animation, the programmer has to write a file in a specialized little programming language, the *agat* language, to describe how the streams coming from its program should be altered or combined, what exactly should be displayed and how... Here is an example *agat* file that can be used to animate the `aleat.c` program :

```
-- file animAleat.agt

include "stat.agt","misc.agt"; -- include some predefined agat functions

let ci=account(i);      -- count the number of values emitted on stream "i"
let avi=acaverage(i); -- average value on "i"
let inci=increase(i); -- increment between two values

-- we "delay" the streams to be able to have a value and the preceeding one together
let di=delay(i);
let dci=delay(ci);
let davi=delay(avi);
let dinci=delay(inci);

-- and then we visualize all the streams...

-- an histogram view of "i"
thinbar(i);
-- various plots (points in the plane)
plot(ci, i);
plot(ci, avi);
plot(di, i);
plot(davi, avi);
plot(dinci, inci);
```

Here, from the raw stream `i` of the random numbers, we choose to compute (construct) three other streams. `ci` is a stream counting the number of random values produced (it will thus contain 1, 2, 3, 4...). The stream `avi` will contain the successive averages of the generated numbers and `inci` the differences between two successive random numbers.

As we are interested in comparing the current value on these streams with the preceding one, we "delay" them, creating three other streams, `di`, `dinci` and `davi`. The values on the stream `di` are the values on `i` with a delay of one value. There is a natural notion of time in *agat*. A tick of the clock corresponds with the arrival of a value emitted by the program. The streams are therefore naturally synchronized through this clock. This means that, if `i` is 0.2 at time t_0, 0.3 at t_1, 0.1 at t_2 and so on, `di` will be 0.2 at time t_1, 0.3 at t_2... (and will contain no value at t_0).

The functions like `account`, and `delay` are written in *agat* and defined in library files (included with the `include` directive at the top of the *agat* program).

For the animation itself, we choose to plot an histogram of the random values produced (the `thinbar` primitive that looks a lot like the `xload` application) and several simple plots. `plot(ci, i)` produces a point for each pair of values arriving on the streams `ci` and `i` (a point for each pair of (`ci`, `i`) values). All these graphical operators dynamically choose the right scale and ranges and can display axes with smart tic

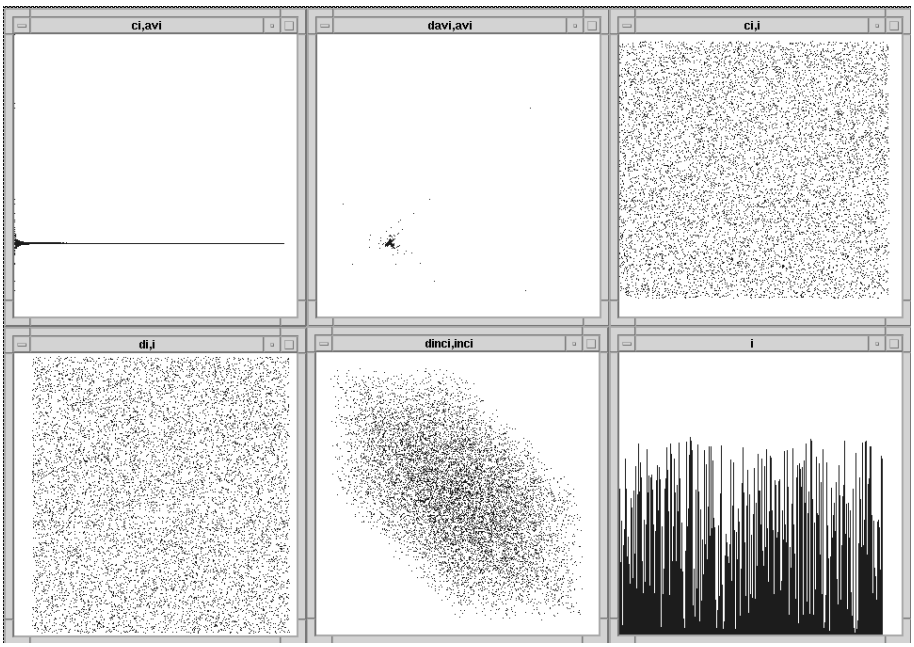

Figure 2: The result of an animation

marks. There also exists operators that link successive points with lines, display boxes of varying height and width and show the ratio of the values on different streams.

The animation is then obtained by running the `aleat` program under the control of the `agat` animation server. The simple command line `agat -f animSimple.agt aleat` do the job and six windows then appears on the user screen (as six graphical operators are called in the *agat* program), showing the evolution of the six streams. Figure 2 shows such an animation. Figure 3 shows what is obtained if we replace the classical generator used in our `alear.c` program (`drand48`) with another less another one. We can clearly see that this is a less random one, demonstrating the interest of *agat* for finding non trivial facts about algorithms.

This example, as simple as it is, illustrates the two most interesting features or our animation tool. First, animating an existing program is very simple : you just include calls to the *agat* library at each point where interesting data are generated. Contrary to all the animation tools that we know, you do not specify the animation inside your source file. Second, changing an animation is very easy because of the client-server architecture of our tool. The animated program runs as a client of the *agat* animation server. The program sends the requested data to the server that is in charge of interpreting the *agat* program and displaying the animation windows. To change an animation, there is no need to recompile the program (as long as you don't need some other data, of course). You just change the *agat* program describing the animation and rerun the *agat* server with this new specification.

This architecture makes it also very easy to adapt *agat* to the animation of programs written in other programming languages (for example, an interface to FORTRAN has been done). Writing a full *agat* library is just a matter of a few dozens lines of code (much less if the C functions of the provided library can be linked) : there are only six functions with a very simple interface.

Some simple guidelines on how to instrument the code should be followed to avoid recompilation and make the best uses of *agat* features : only raw data should be sent, as processing can be done at the *agat* level (the most common mathematical functions are available in *agat*). The use of several streams is strongly advised (two streams can easily be combined into one in *agat*). Sending more data than actually needed is harmless and can become useful (ignoring a stream is very easy: you just don't mention it in the *agat* program).

Note that including calls to *agat* library functions in your program does not modify its behavior (at least if it is a correct program). It can be run without the control of the *agat* server and the results will be similar to what would be obtained without the calls to the *agat* library.

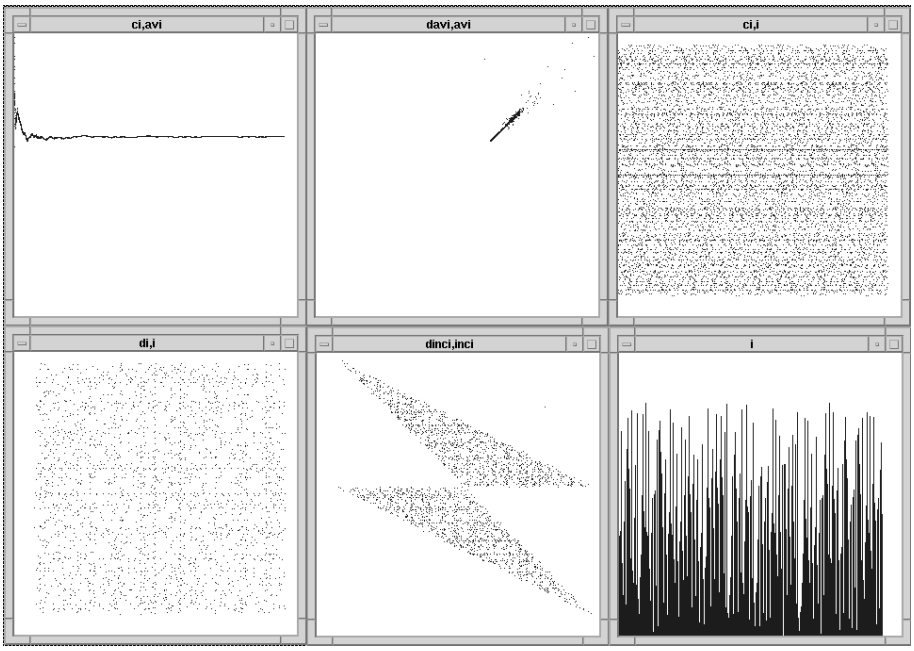

Figure 3: A "bad" generator

Another interesting feature of the *agat* server is the ability to record the values emitted by the program. It is then possible to try many different animations from these data without running the program each time. The speed at which the animations are replayed can be tuned (the time at of the emission of each data is recorded so that we can keep a totally accurate image of the progress of the algorithm).

3 The AGAT language

As seen in the preceding section, to animate a program with *agat*, the user has to go through three different steps :

- adding *send* commands into the program, to create raw streams of interesting values
- describing an *animation* using a configuration file, to build new streams connected with graphical display operators
- invoking the animation

The first step is important, because the choice of input data strongly influences the message that the animation conveys. The second step is the most difficult part because the user has to decide what the *view* should look like in order to show interesting information. A *view* results of the connection of a stream with a graphic display. An interesting point with *agat* is that the user can elaborate multiple views from a few basic streams. These views can be changed with no modification of the animated program. The *agat* system offers a great variety of constructs to combine raw streams of values into new streams. These constructs are described with a small but powerful language which allows to express subtle mappings on streams (see [ASS85] for an introduction to programming with streams and and [Wat91] for a more complete treatment). A simple instruction is the one that defines a new stream using a predefined function, like in `let ci = account(i)`. The values on the stream `ci` are the successive numbers of values emitted on the stream `i` (that is 1,2,3,4,...). The predefined function `account` is actually written in *agat*, and uses the basic construct `map`. For example, given a stream `f`, then `let logf = map(x -> log(x)) f;` defines a new stream `logf` of the logarithms of the values of `f`. This means that each time a value `v` is emitted on the stream `f`, the value `log(v)` is emitted *at the same time* on the stream `logf`. We will give later more details

on what *"at the same time"* means. The operator `map(x -> log(x))` maps the mathematical function `log` on each value of the stream it is applied to. The left-side of the arrow is called the *activation pattern* and the right-side is the *action* to be performed on the values. The activation pattern describes the *state* of the stream(s) to be mapped. For a given stream, this state could be one of the following :

- *new value* : a new value is emitted on/by the stream (actually, the propagation of a value on a stream is *instantaneous* in *agat*)
- *no value* : no (new) value has been emitted on the stream. Older values have been already emitted or not.
- *old value* : no (new) value has been emitted on the stream but at least one older value did.
- *not yet a value* : no value at all has never been emitted on the stream.
- *first value* : a new value is emitted on the stream and it is the first one.

Activation patterns are very useful for mappings with multiple inputs. For example, the stream `h` defined by `let h = map(a,b -> a+b) f,g;` maps the function `+` on the two input streams `f` and `g`. The simple activation pattern `a,b` implies that the values emitted on `f` and `g` are *buffered* until there are two new values that are available on `f` and `g`. Since there is no time spent for propagation on a stream, the pattern `a,b` never matches if `f` and `g` are two raw streams. Let `hh` be the stream defined by :

```
let hh = map( a,b -> a+b
             |a,* -> a
             |*,b -> b ) f,g;
```

The star `*` matches the state *no value*. If only a new value is emitted on the stream `f`, the pattern `a,b` does not match (like in the stream `h`), but the pattern `a,*` does. Notice that if `f` and `g` are two raw streams, the pattern `a,b` never matches, and `hh` is the *merged* stream of `f` and `g`. Another example of a multiple inputs mapping with special activation pattern is given by the stream `average` defined by :

```
let average = map( a ,b  -> (a+b)/2
                  |a ,b+ -> (a+b)/2
                  |a+,b  -> (a+b)/2 ) f,g;
```

Given the streams `f` and `g`, the stream `average` carries the average values of the two more recent values emitted by `f` and `g`. The pattern matching and the corresponding action performed on the inputs of a map construct are *instantaneous*. Thus, the outputs of `average`, `f` and `g` are *synchronized*, that is, they occur *"at the same time"*. We can summarize the behavior of the streams `h`,`hh` and `average` (`av`) on a table. The values which appear vertically centered are synchronized :

TIME
$\longrightarrow$

f	1		2	3		4
g		10	20		30	40
h		11	22		33	44
hh	1	10	22	3	30	44
av		5.5	11.0	11.5	16.5	22.0

Introducing a new map construct using *registers*, the following defines the stream function `delay` that creates a delayed copy of a stream :

```
let delay(i) =
    map {last,tmp}
        (  x^ -> last:=x, null
         | x  -> tmp:=last, last:=x, tmp ) i;
```

Notice that `delay` is not a stream, but a *function* that takes a stream as input and returns a new stream. The character ^ matches the state *first value*. The identifiers appearing inside brackets are called *registers*. They act as local variables for the map construct they appear in. When a sequence like `tmp:=last, last:=x, tmp` occurs, each instruction is evaluated and the last expression is the value output by the stream. The `null` instruction is used when there is no value to output. An important rule is that inside a map, the patterns are checked in the order they appear. Then, in the example of `delay`, the first value emitted by the stream parameter `i` matches the first pattern and allows to initialize the local variable `last` to its proper value. In addition to the predefined functions (like `+`) and the assignment (`:=`), the *agat* language provides a *conditional* statement. The stream function `positive` creates the stream of the positive values of a given stream :

```
let positive(i) =
    map ( a -> if a>0 then a else null) i;
```

As registers allow to build *accumulators*, the conditional statement allows to create *filters*. Mapping, filtering and accumulating are the basic functions required for a powerful stream language.

Although the user can easily write its own functions, the *agat* system comes with a library of predefined stream functions. For example, the function `extremafilter` which extracts the local extrema of a given stream is in the library :

```
let extremafilter(i) =
    map {slope,last,tmp}
        (  x^ -> last:=x, slope:=0, null
         | x  -> if sgn(x-last) != slope
                 then tmp:=last, slope:=sgn(x-last), last:=x, tmp
                 else last:=x, null
                 end ) i;
```

4 Conclusion

agat has been successfully used in our research group to investigate the behavior of complex algorithms and also as an occasional debugging tool as well as an interactive (on-line) plotter. For example, a binomial Groebner basis algorithm has been studied as well as a very complex identification problem involving high degree polynomials for the design of microwave filters. It has been proven a very valuable tool. For our uses, *agat* could not have been replaced by any other more sophisticated animation tool. They would have required too much efforts to set up interesting animations for our colleagues to accept them.

From a language design point of view, some people will object that we certainly don't need another programming language, even for algorithm animation... When we designed the language, we had considered extending existing "little" languages (like *scheme* or TCL). Two main reasons make us choose to design yet another language. The first is that we did not see how the main features of *agat*, streams processing with pattern-matching and synchronization, could have been nicely integrated in such a language (some difficult semantics issues arise that we can control in the context of the limited constructs of *agat*). The second one is the desire to keep the language as simple as possible to be accessible to the largest possible class of users. For the moment, learning *agat* has never been a problem for our users (even for rather "math oriented" people). Of course, in the long term, the risk is that the needs to extend *agat* could reach the point where it should become a real "full-featured" programming language. This could forced us to reconsider our decision but we hope to be able to carefully control the evolution of *agat* to keep it simple enough.

In the near future, we plan to add more sophisticated graphical operators, including 3D graphics, especially to provide richer visualizations of arrays. There is certainly a lot to do with 3D graphics for algorithm animation (in the line of [BN93] and [SW93]). Considering our approach of providing relatively simple graphical operators, it is not clear that going from 2D to 3D would really be a very valuable improvement (in the sense that the use of these new functionalities could be rather low, the simple things being often the most useful). We are also investigating the possibility of adding primitives to create and move around simple objects (following the work of [KS89]). The challenge here is to keep the *agat* language, and most interesting *agat* programs that should use these new functionalities, simple enough.

An interesting long-term project can be the embedding of *agat* in a source level debugger (like gdb). The basic idea is to offer full access to all the *agat* functionalities to animate the evolutions of the interesting data of a program running under the control of the debugger. The *prism* programming environment of *Thinking Machines Corporation* for Connection Machine systems already includes a debugger that has the ability to visualize arrays in several graphical ways but this cannot be considered as a real animation with the possibilities of *agat*. Another (easier) idea could be to turn the *agat* server into a real interactive interpreter where the user could stop the computation, introduce new animation code and restart the program at will. This can be considered as a nice extension of the "debugging by printing" method.

Even if *agat* can only deal with numerical values, and not complex symbolic objects, we think that it could be a very effective tool to look at complex symbolic algorithms (as exemplified by our experience with the Groebner basis algorithm), because, after all, the evolution of simple numerical data is all that most people can easily handle ([Ber83]).

References

[ASS85] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structures and Interpretation of Computer Programs*. MIT Press, 1985.

[Ber83] Jacques Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[BH91] Mark H. Brown and John Hershberger. Color and sound in algorithm animation. Digital SRC Research Report 76a, DEC, 1991.

[BN93] Mark H. Brown and Marc A. Najork. Algorithm animation using 3D interactive graphics. Digital SRC Research Report 110a, DEC, 1993.

[Bro88] Mark H. Brown. Exploring algorithms using BALSA-II. *IEEE computer*, 21(5):14–36, 1988.

[Bro92] Mark H. Brown. Zeus : A system for algorithm animation and multi-view editing. Digital SRC Research Report 75, DEC, 1992.

[Dui86] Robert Duisberg. Animated graphical interfaces using temporal constraints. In *ACM CHI '86 Conference on Human Factors in Computing*, pages 131–136, 1986.

[KS89] Mukkai S. Krishnamoorthy and Ramesh Swaminathan. Programs tools for algorithm animation. *Software–Practice and Experience*, 19(6):505–513, 1989.

[Sta90] John T. Stasko. Tango : a framework and system for algorithm animation. *IEEE computer*, 23(9):27–39, September 1990.

[SW93] John T. Stasko and Joseph F. Wehrli. Three-dimensional computation visualization. In *1993 IEEE symposium on Visual Languages*, pages 100–107, 1993.

[Wat91] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):53–98, janvier 1991.