Using Beatbox: further details

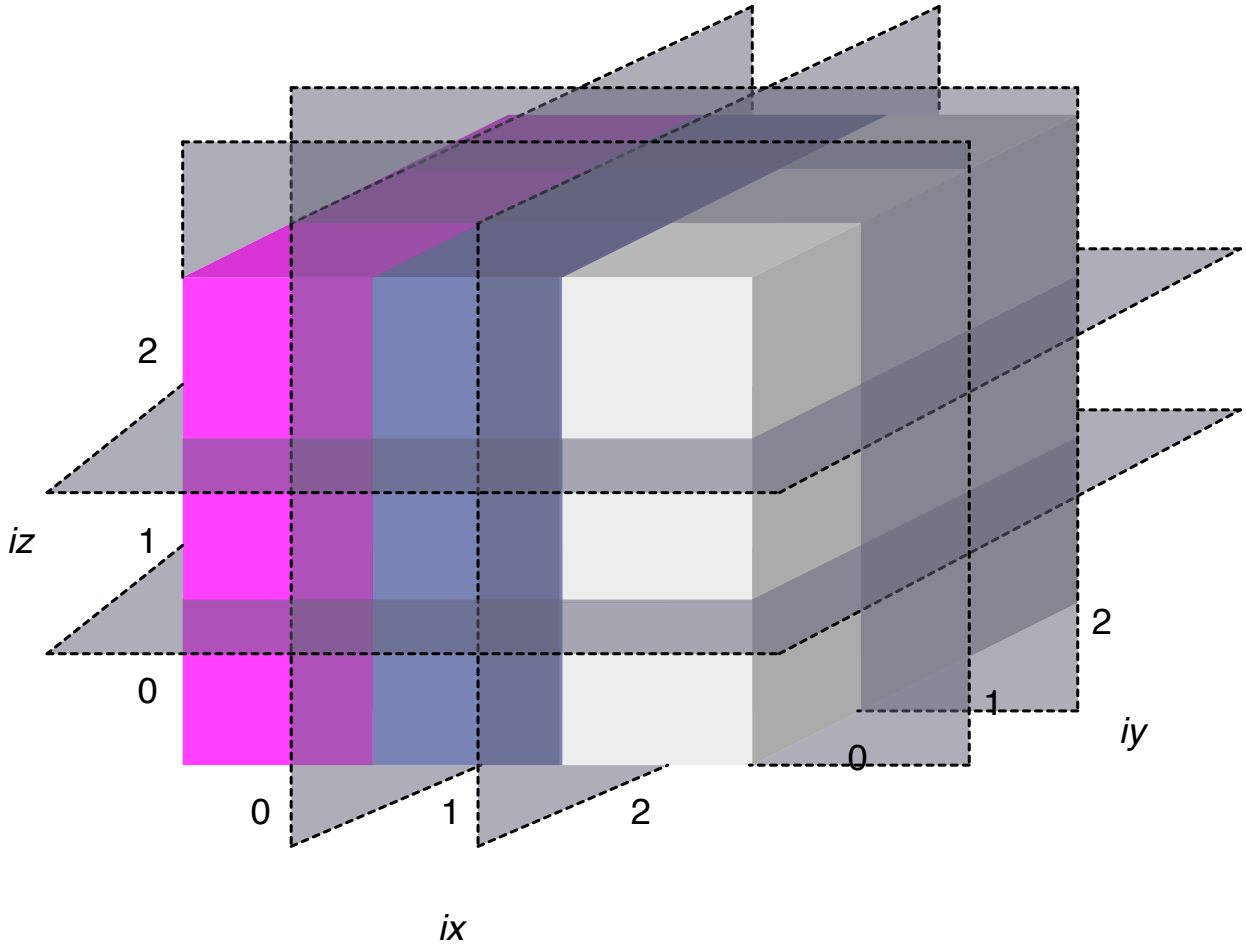
Plan

- Parallel BeatBox
- Complex geometries
- More about BBS syntax
- More advanced scripts: fhn0
- More advanced scripts: fhn1
- More advanced scripts: fhn2

(pictures thanks to Ross McFarlane)

PARALLEL BEATBOX

Domain Decomposition Supergrid with Superindices



Launching BeatBox: MPI mode

> mpirun –np 27 BeatBox somescript.bbs

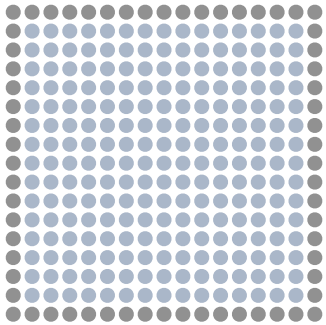
for automatic domain decomposition,

> mpirun –np 27 BeatBox somescript.bbs –decomp 3x3x3

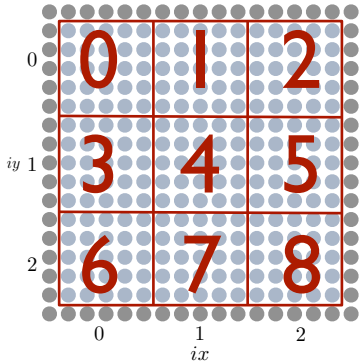
for explicit domain decomposition formula. Note that in this example, 3x3x3=27: each subdomain is allocated one process to work with it.

As in sequential mode, the script can be followed take extra parameters and further options.

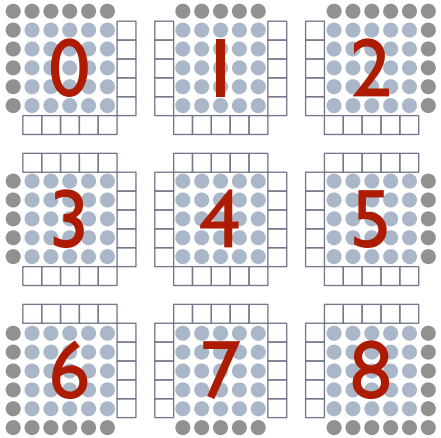
Domain decomposition in 2D



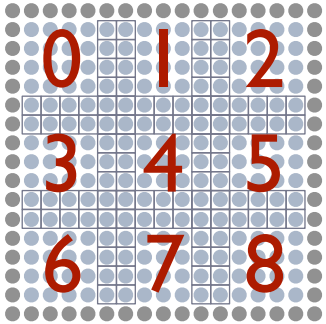
The grid and boundary points



The grid and the supergrid



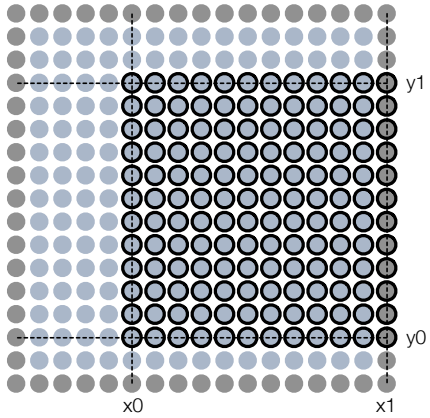
Subdomains with halos



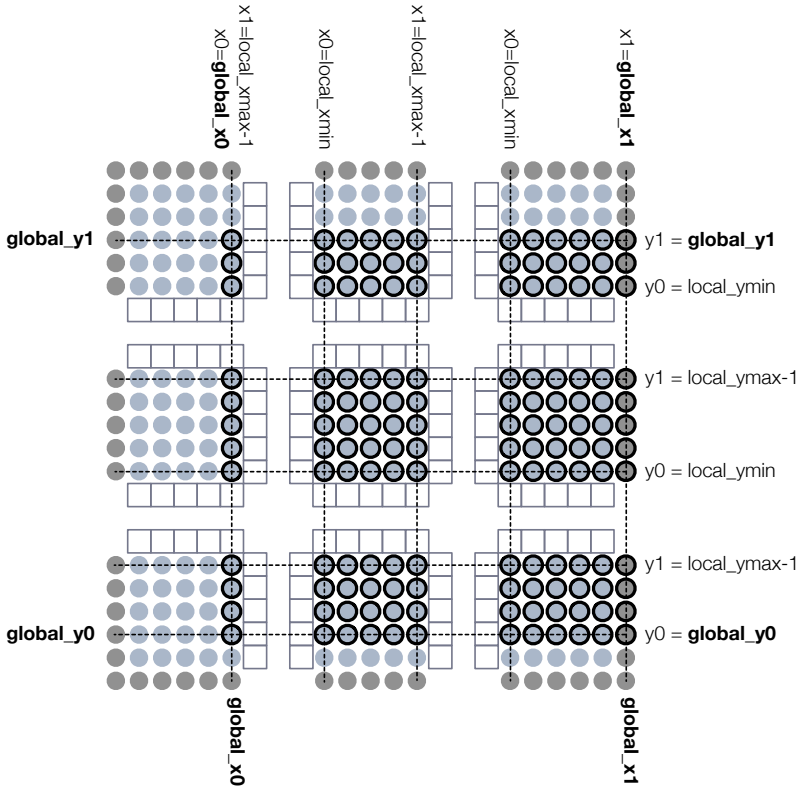
Subdomains united

The concept of a device space

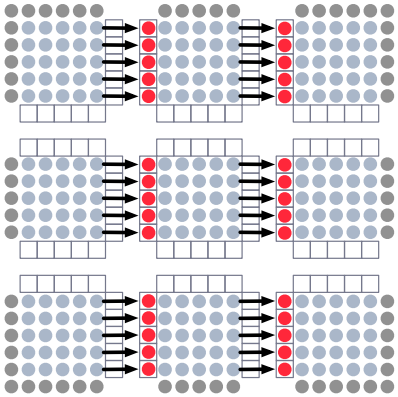
Sequential



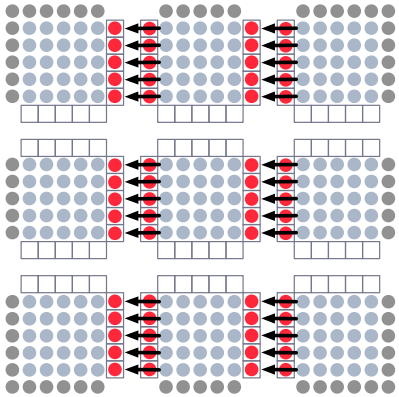
Parallel



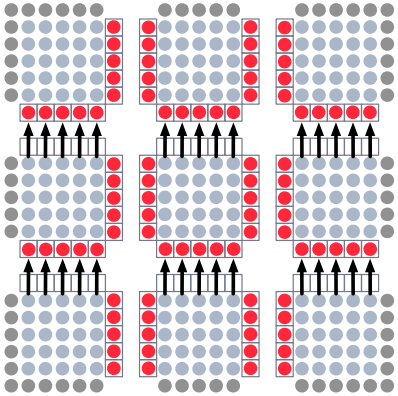
Halo swap in 2D



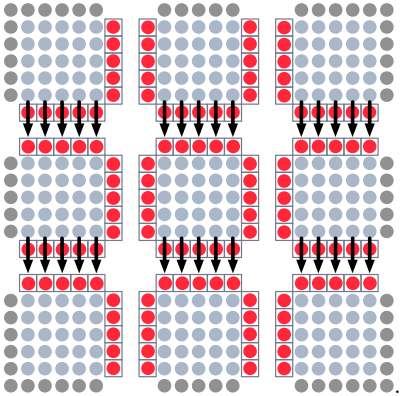
(a) Step 1



(b) Step 2



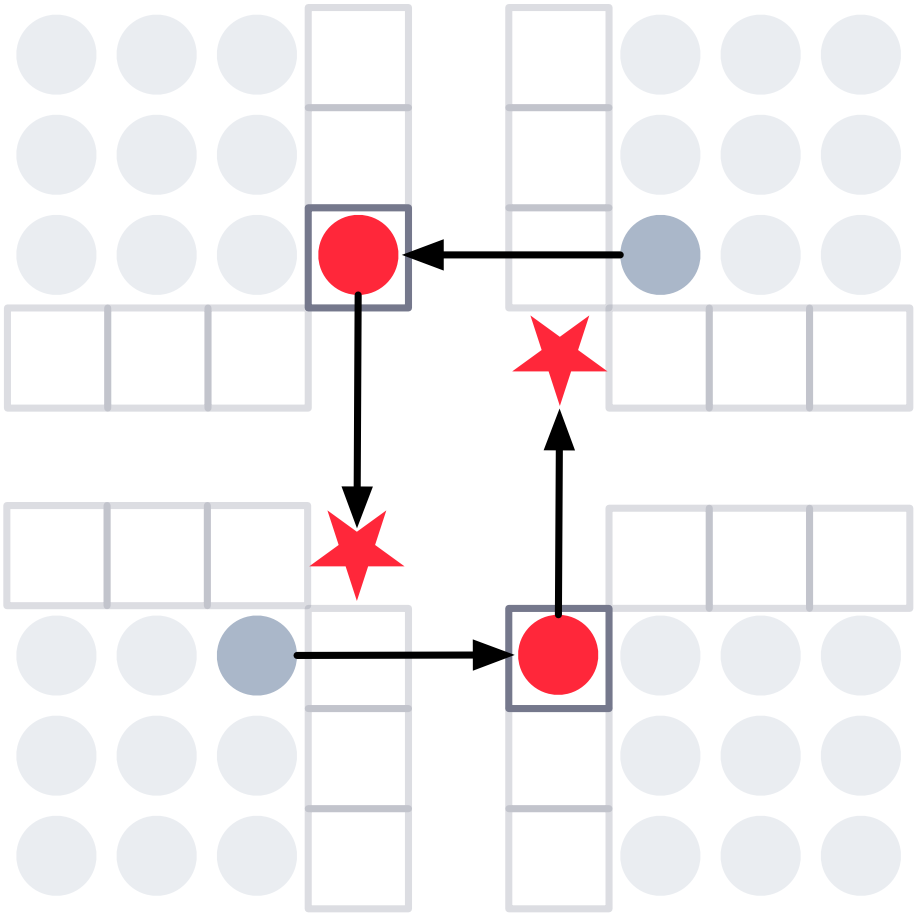
(c) Step 3



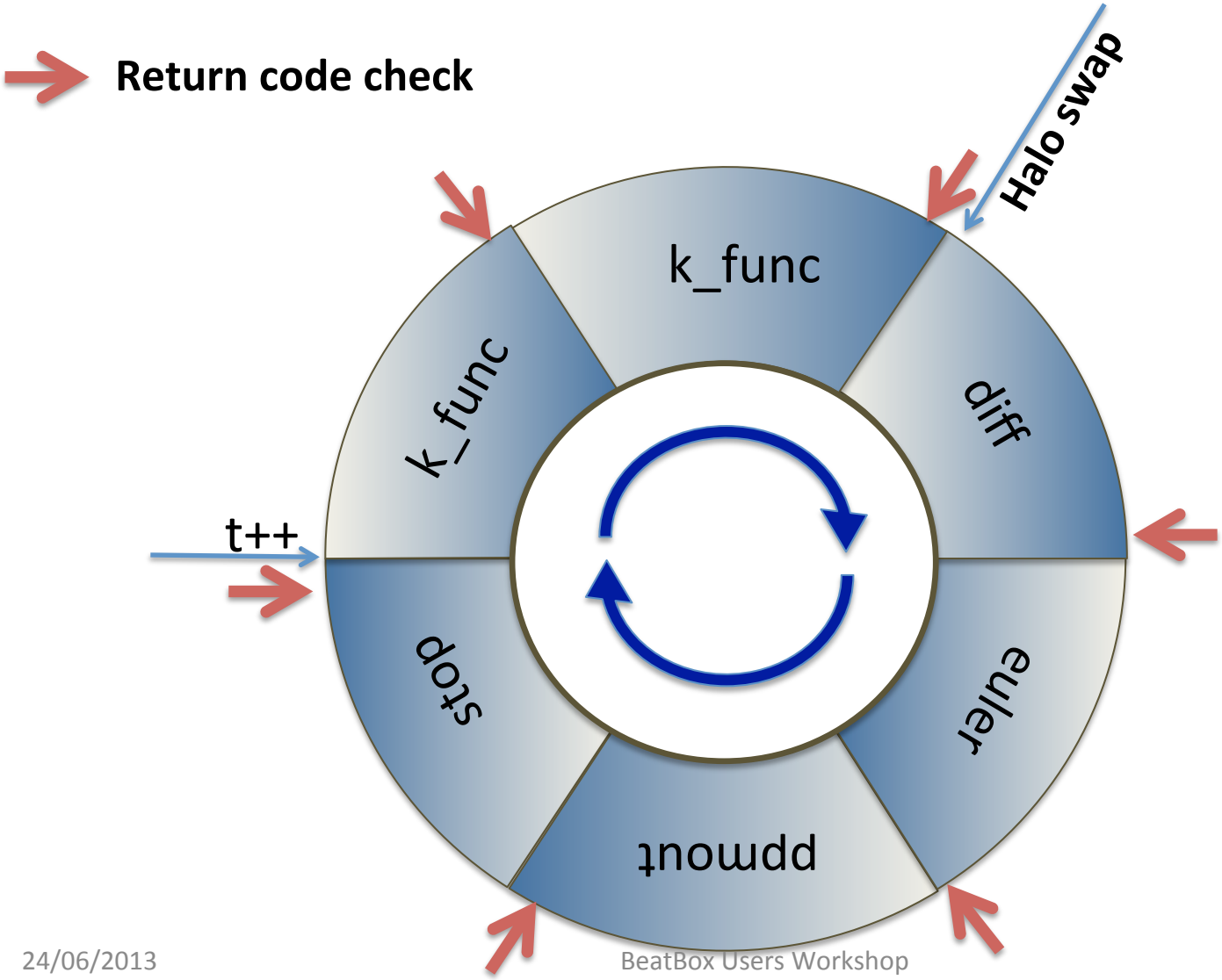
(d) Step 4

Magic corners

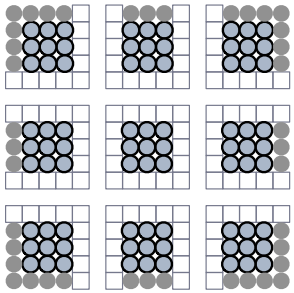
- Exchange with diagonal neighbours does not require extra action if halo swapping is done in the right order



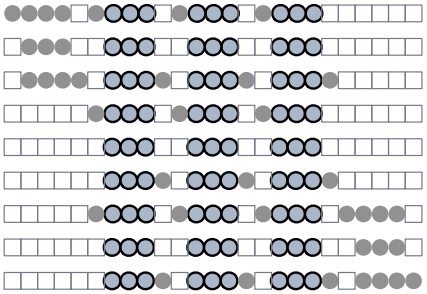
The ring of devices in MPI



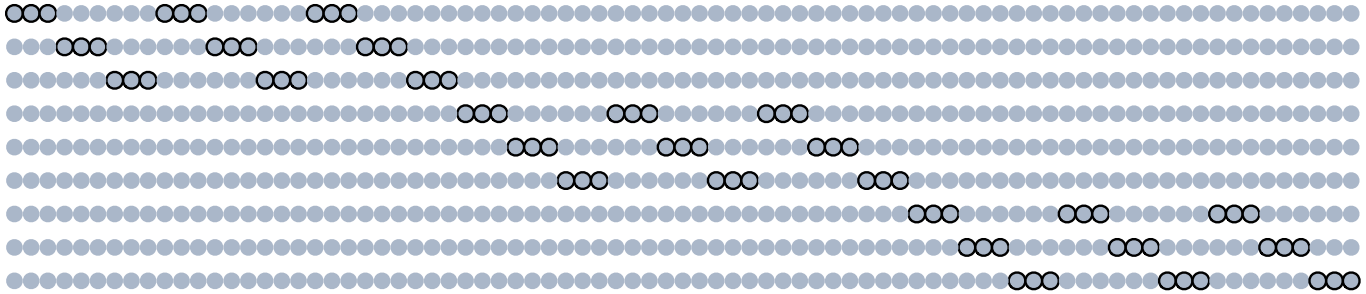
Parallel output: MPI source types and file types



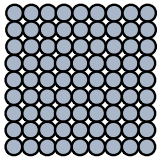
(a) The decomposed space to be written to file.



(b) Source types for each process, defining the local space as part of `New`.



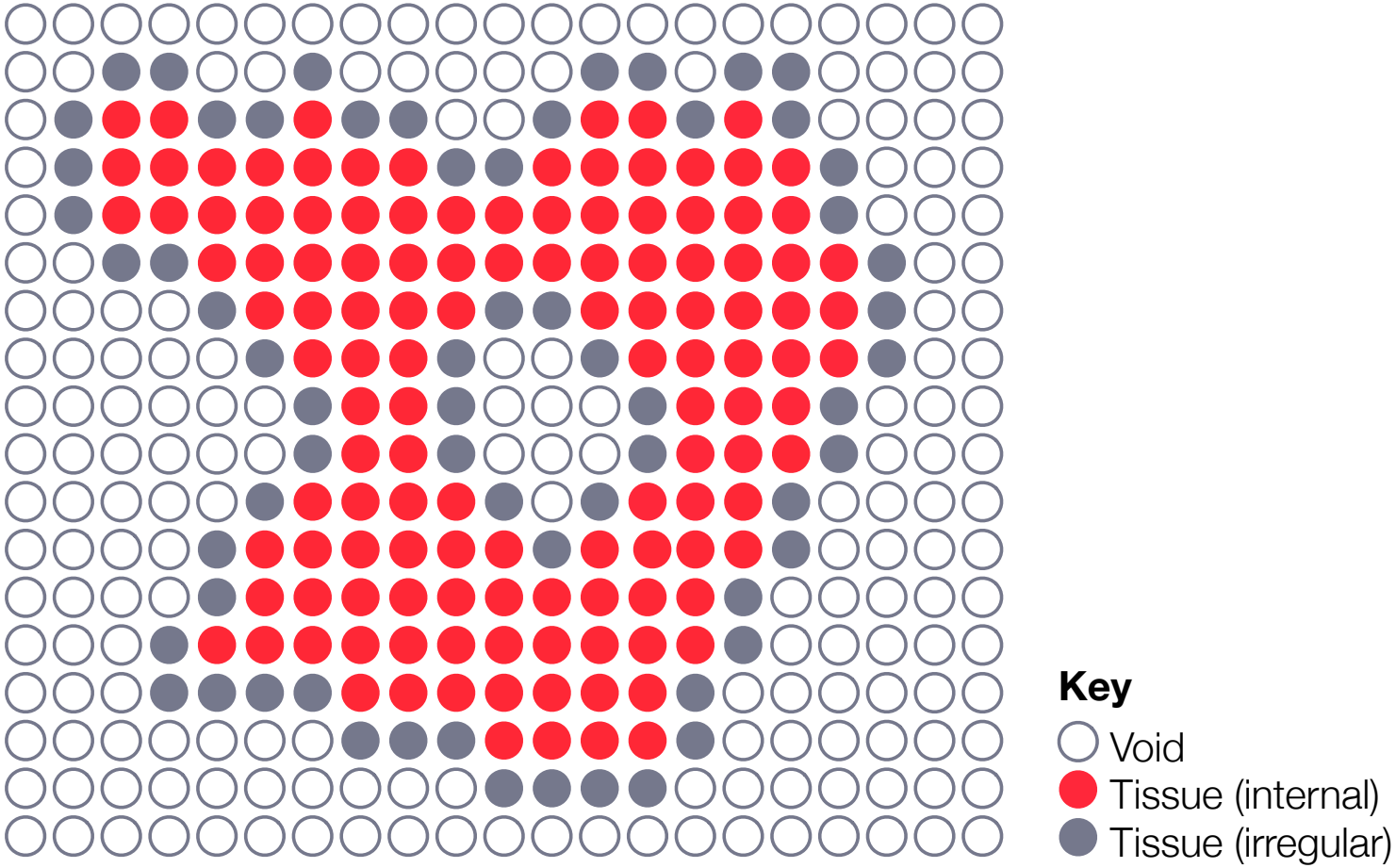
(c) Filetypes for each process, defining the local space as part of the global space.



(d) The completed file.

COMPLEX GEOMETRIES

Arbitrary domain in Cartesian grid



Format of a geometry file

- Comma-separated (in recent versions can be space-separated) text file with four or seven columns.
- Each row describes a point of the geometry.
- Columns 1-3: integer x,y,z coordinates of the point
- Column 4: integer point type. 0 means void (non-tissue). Nonzero values are all equivalent (this may change in later versions)
- Columns 5-7: real components of a vector defining the fibre direction at the point. May or may not be normalized.

```
66,16,50,1,-0.455,1,0.278
67,16,50,1,-0.3,1,0.32
68,16,50,1,-0.098,1,0.336
69,16,50,1,0.216,1,0.328
70,16,50,1,0.353,1,0.35
71,16,50,1,0.489,1,0.373
72,16,50,1,0.61,1,0.388
73,16,50,1,0.711,1,0.386
65,17,50,1,-0.524,1,0.284
66,17,50,1,-0.458,1,0.29
67,17,50,1,-0.383,1,0.26
68,17,50,1,-0.277,1,0.224
69,17,50,1,-0.187,1,0.2.
…
```

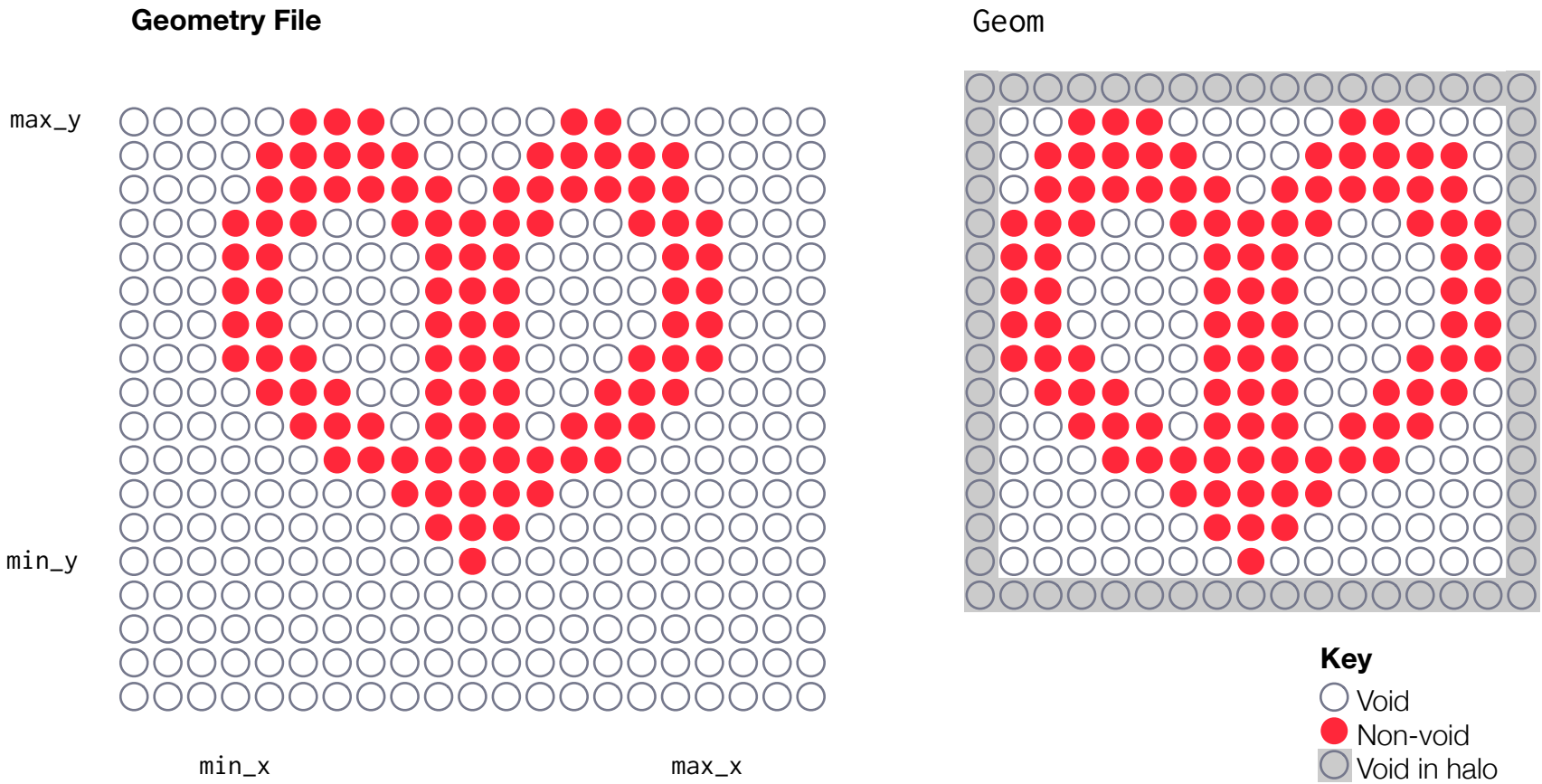
state command
with realistic geometry

state geometry=<file.bbg> [vmax=<value>]

[anisotropy=<value> [normaliseVectors=<value>]] ;

- The enclosing box size is calculated automatically. Hence xmax, ymax, zmax are not only not required, but are not allowed.
- vmax defaults to 2 (as in FitzHugh-Nagumo).
- anisotropy defaults to 0 (means no)
- normalizeVectors defaults to 0 (means no).

Adjusting the grid to fit the geometry



Launching BeatBox in MPI mode with geometry

> mpirun –np 96 BeatBox humanAtrium.bbs –decomp 5x5x5

With complex geometries, some of the subdomains may happen to contain no tissue points at all. In that case, it is not necessary to allocate processors for those points, and number of processors (96 in this example) may be less than the number of subdomains in the decomposition (5x5x5=125 in this example).

Filling quality

```
547 21:14:33 FitzHughNagumo_model_IVB$ mpirun -np 32 Beatbox fhn_ffr.bbs
#! BeatBox 1.3 revision 692M, MPI compile Jun 20 2013 21:14:25
…
/* Domain decomposed as (002,004,004). */
/* 31 processes will be active, 1 will be idle. */
/* 1 subdomains are empty. */
/* Loading geometry data and normalising vectors...done. */
/* Filling quality 50.6%, 0=0.0% processes with empty subdomains */
```

- The speed of computations is defined by the speed of the slowest processor, i.e. the one with the fullest subdomain.
- Filling quality is the ratio of the average load per active process to the load of the busiest process. It is a rough indication of slow-down due to unevenness of the load.
- This can be taken into account when choosing the decomposition formula.

MORE ABOUT BBS SYNTAX

Preprocessing

- skipping comments (already covered)
- including other Beatbox scripts,
- calling system commands and
- expanding string macros.

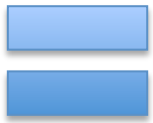
Including other BeatBox scripts

main.bbs:

```
// This is my own script
<useful.bbs>
// Here's some more of my own code.
def real cpar=3;
```

useful.bbs:

```
// Here is a lot of useful code...
def real apar=1;
def real bpar=2;
```

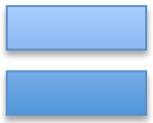


Effective result:

```
def real apar=1;
def real bpar=2;
def real cpar=3;
```

Calling system commands

```
def str now `date '+%Y%m%d-%H:%M:%S'`;
```

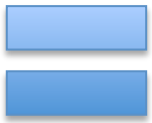


```
def str now 20130624-16:30:55;
```

(assuming it is now 16:30:55 of 24 June 2013)

Expanding string macros

```
def str snack porkpie;
def str hat [snack];
def str headware hat;
```

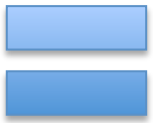


```
def str snack porkpie;
def str hat porkpie;
def str headware hat;
```

Name in square brackets is a macro call.

Expanding string macros

```
def str ninetynine 99;
def int number [ninetynine];
```



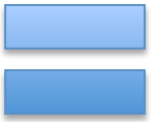
```
def str ninetynine 99;
def int number 99;
```

Macro can expand into anything that can be typed in a script, e.g. a number.

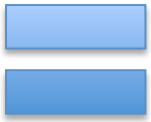
Expanding string macros

```
def int number 99;
def str ninetynine number;
```

```
def int number 99;
def str ninetynine [number];
```



```
def int number 99;
def str ninetynine number;
```



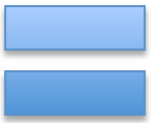
Error, unless you also defined somewhere something like

```
def str number somevalue;
```

A number variable does not, however, make a macro

Expanding string macros

```
def str subset x0=minx x1=maxx
y0=miny y1=maxy;
k_print [subset] when=often file=often.out;
k_print [subset] when=seldom file=seldom.out;
```



```
k_print x0=minx x1=maxx
y0=miny y1=maxy when=often file=often.out;
k_print x0=minx x1=maxx
y0=miny y1=maxy when=seldom file=seldom.out;
```

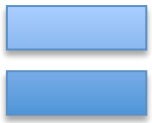
Macro can expand into anything that can be typed in a script, including spaces and several lines of code (excluding a semicolon).

Predefined string macros

Command line: Beatbox foo.bbs 7 8;

foo.bbs:

```
def str outdir [0].dir;
def int apar [1];
def  str  name [2];
…
```



Effective result:

```
def str outdir foo.dir;
def int apar 7;
def  str  name 8;
…
```

MORE ADVANCED SCRIPTS: FHN0

What is this about

- The minimal.bbs script considered earlier is indeed minimal and does the sort of task done by any cardiac simulation package. However, Beatbox's flexibility allows you to do much more than that.
- Now we will consider some sample BeatBox scripts that illustrate further features of BeatBox, both in terms of syntax and in terms of functional capabilities, as well as some useful scripting techniques.
- The scripts themselves can be found in the BeatBox distribution, under data/scripts/, and they are well commented so could be self-explanatory.
- In the presentation we will focus on the novel features, as much as time permits.

fhn0.bbs

- Location: data/scripts/sequential/FitzHughNagumo_model/fhn0.bbs
- Protocol: single cell with FHN kinetics, the action potentials initiated repeatedly. The solution corresponding to the n-th action potential, n=4, is output to file fhn0.rec.

New features:

- Use of string macros.
- sample device to convert grid value into a global k-variable, which is required for the k_func;
- k_func as a feedback control device: stimulating shock is defined as a function of current cell state;
- screen command and k_draw device for run-time graphics output: draw phase trajectory of the system as the solution progresses;
- BeatBox script conditionals;
- record output device to write contents of the grid (in this case, just single cell) to a file.
- clock and k_clock output devices that shows integer simulation time counter.

Using string macros

```
// Allocation of layers to the state varables: u in layer 0, v in
layer 1.
def str u 0;
def str v 1;
state xmax=1 ymax=1 zmax=1 vmax=[v]+1; // that means, vmax=1+1=2
```

- String macros are defined by def command, same as arithmetic variables, but with a type str.
- They can be used in place of the script, not only in k-expressions. To tell them from the normal text, the square brackets […] are used.
- It is convenient to give names to layers rather than keeping in mind what layer keeps what field. We shall see that string macros are more convenient for layer names than arithmetic variables.

sample control device

```
sample name=Ust when=begin v0=[u] result=Ust; // No x0=... etc  are needed
sample name=Vst when=begin v0=[v] result=Vst; // as there is only one point
                                              // in the grid.

// Get the values of U and V as global variables, at all t.
sample name=U v0=[u] result=U;
sample name=V v0=[v] result=V;
```

- Function: assign a value from the grid to a global k-variable. This cannot be done by a k_func device due to a parallelizability limitation (more about it later).
- Only x0,y0,z0,v0 space coordinates are used: they identify the grid value.
- Parameter result should be assigned to a real k-variable, not an expression. NB: unlike most device parameters, this is not an assignment as such (not done at run time), but remembering the name of the k-variable. In that sense it is similar to the generic when parameter.

sample device

```
sample name=Ust when=begin v0=[u] result=Ust; // No x0=... etc  are needed
sample name=Vst when=begin v0=[v] result=Vst; // as there is only one point
                                              // in the grid.

// Get the values of U and V as global variables, at all t.
sample name=U v0=[u] result=U;
sample name=V v0=[v] result=V;
```

- In this example, name=Ust and name=Vst only work once in the beginning before any calculations, so they record the initial value (the resting state).
- The name=U and name=V device work every time, and their global variables are used as the run-time feedback for the stimulation protocol.

k_func as feedback control device

```
// Feed-back stimulation: stim on when the cell is close to the resting state;
k_func name=feedback nowhere=1 pgm={
  Iu= lt(U,Ust+0.05)*lt(V,Vst+0.05)*1.0; // 1.0 if both U and V are close to
                                         // resting state.
  pulsecount=pulsecount+gt(Iu,0);        // Increment the counter if shock.
  chosenpulse=eq(pulsecount,4)*often;    // Check if this is the chosen pulse.
  end=end+gt(pulsecount,4);              // Update the end condition: nothing
                                         // to do after the chosen pulse.
  /* in the expression above, '+' serves as logical 'or' */
};
```

- Reminder: nowhere=1 means that this device operates with and only with global variables.

k_func as feedback control device

```
// Feed-back stimulation: stim on when the cell is close to the resting state;
k_func name=feedback nowhere=1 pgm={
  Iu= lt(U,Ust+0.05)*lt(V,Vst+0.05)*1.0; // 1.0 if both U and V are close to
                                         // resting state.
  pulsecount=pulsecount+gt(Iu,0);        // Increment the counter if shock.
  chosenpulse=eq(pulsecount,4)*often;    // Check if this is the chosen pulse.
  end=end+gt(pulsecount,4);              // Update the end condition: nothing
                                         // to do after the chosen pulse.
  /* in the expression above, '+' serves as logical 'or' */
};
```

- The first assigment uses U,V,Ust,Vst obtained through earlier sample device to give value of 0 or 1 to k-variable Ist. It will be 1, if simultaneously the current U is less than Ust+0.5 and the current V is less than Vst+0.5, i.e. the phase point has approached the resting state sufficiently closely.

k_func as feedback control device

```
// Feed-back stimulation: stim on when the cell is close to the resting state;
k_func name=feedback nowhere=1 pgm={
  Iu= lt(U,Ust+0.05)*lt(V,Vst+0.05)*1.0; // 1.0 if both U and V are close to
                                         // resting state.
  pulsecount=pulsecount+gt(Iu,0);        // Increment the counter if shock.
  chosenpulse=eq(pulsecount,4)*often;    // Check if this is the chosen pulse.
  end=end+gt(pulsecount,4);              // Update the end condition: nothing
                                         // to do after the chosen pulse.
  /* in the expression above, '+' serves as logical 'or' */
};
```

- The second assignment increments k-variable pulsecount by 1 if variable Iu (as a result of the previous assignment) is positive (it could be only 0 or 1, remember). Assuming (!) that this shall happen only once per pulse, the value of pulsecount will correspond to its name.

k_func as feedback control device

```
// Feed-back stimulation: stim on when the cell is close to the resting state;
k_func name=feedback nowhere=1 pgm={
  Iu= lt(U,Ust+0.05)*lt(V,Vst+0.05)*1.0; // 1.0 if both U and V are close to
                                         // resting state.
  pulsecount=pulsecount+gt(Iu,0);        // Increment the counter if shock.
  chosenpulse=eq(pulsecount,4)*often;    // Check if this is the chosen pulse.
  end=end+gt(pulsecount,4);              // Update the end condition: nothing
                                         // to do after the chosen pulse.
  /* in the expression above, '+' serves as logical 'or' */
};
```

- The third assignment will make variable chosenpulse nonzero iff variable pulsecount value equals 4, and at the same time, variable often is nonzero.
- Variable often has been defined by k_func name=timing to be nonzero at every 10th step.
- Variable chosenpulse will be used to control output: it will happen only during the chosen pulse, and only at every 10th timestep.

k_func as feedback control device

```
// Feed-back stimulation: stim on when the cell is close to the resting state;
k_func name=feedback nowhere=1 pgm={
  Iu= lt(U,Ust+0.05)*lt(V,Vst+0.05)*1.0; // 1.0 if both U and V are close to
                                         // resting state.
  pulsecount=pulsecount+gt(Iu,0);        // Increment the counter if shock.
  chosenpulse=eq(pulsecount,4)*often;    // Check if this is the chosen pulse.
  end=end+gt(pulsecount,4);              // Update the end condition: nothing
                                         // to do after the chosen pulse.
  /* in the expression above, '+' serves as logical 'or' */
};
```

- And finally, variable end will be nonzero if it was already nonzero (as a result of k_func name=timing) or pulsecount variable exceeded 4.
- The first of these conditions is necessary to ensure against infinite runs when for some reason pulses do not happen so pulsecount is not incremented. The second condition is to avoid doing computations when they are no longer needed.

k_func as feedback control device

```
// Feed-back stimulation: stim on when the cell is close to the resting state;
k_func name=feedback nowhere=1 pgm={
  Iu= lt(U,Ust+0.05)*lt(V,Vst+0.05)*1.0; // 1.0 if both U and V are close to
                                         // resting state.
  pulsecount=pulsecount+gt(Iu,0);        // Increment the counter if shock.
  chosenpulse=eq(pulsecount,4)*often;    // Check if this is the chosen pulse.
  end=end+gt(pulsecount,4);              // Update the end condition: nothing
                                         // to do after the chosen pulse.
  /* in the expression above, '+' serves as logical 'or' */
};
```

- An alternative method would be to have two stop devices: one controlled by a k-variable that is raised due to time expired, the other controlled by another k-variable that is raised by the pulse count. Neither method is wrong, but adding one assignment to in this (global) k_func may be cheaper than having one more device.

k_func as feedback control device

```
// Shock to elicit the action potential.
k_func name=stim when=Iu pgm={u[u]=u[u] + Iu} debug=stdout; // Debug parameter
                                                           // says print results of
                                                           // calculations to stdout.
```

- The variable Iu assigned a value by the preceding name=feedback k_func, is used in this one to kick the voltage.
- Note it has no space parameters so by default applies to all inner points (there is one point in this 0D example, though). And it only works when Iu is nonzero.
- As a result of the two k_func devices working together, Iu is raised only when the phase point comes close to the resting state, and once it is raised, immediately *u* variable is pushed away from the resting state, thus starting a new pulse. Hence the assumption that Iu is nonzero only once per pulse, which we made earlier, is justified.
- Since the k-program has any effect only when Iu is nonzero, it makes sense to execute this k_func only when Iu is nonzero.

screen command

```
//***********************************************
// The graphics output window will be with 600x600 resolution with
// 10-pixel rims, located 10 pixels from the right and 10 pixels
// from the top of the screen.
screen WINX=-10 WINY=10 XMAX=620 YMAX=640;

// The coordinates of the output zone
def int row0 30; def int row1 629;
def int col0 10; def int col1 609;
```

- Sets a window for run-time graphics output.
- Only one per script (just like state command)
- XMAX, YMAX: window size in pixels. Default is 640x480 (standard screen size of computer displays at some ancient time).
- WINX, WINY: coordinates of the window on the screen, in pixels. WINX: from the left (if positive) or from the right (if negative) end of the screen, similarly for WINY: from top if positive and from bottom if negative.
- The four parameters are then available as integer k-variables.
- In this example, we set the limits of graphics output within the screen.

Running BeatBox with and without graphics

- With graphics:

```
685 20:55:22 FitzHughNagumo_model$ Beatbox_SEQ fhn0.bbs
#! BeatBox 1.2 revision 529:530M, sequential compile Mar 29 2013 19:17:17
…
This run took 1.579033 seconds.
BeatBox 1.2 finished at t=71738 by device 13 "stop"
Thu Jun 20 20:56:15 2013
=====================================================
686 20:56:15 FitzHughNagumo_model$
```

- Without graphics:

```
686 20:57:33 FitzHughNagumo_model$ Beatbox_SEQ fhn0.bbs -nograph
#! BeatBox 1.2 revision 529:530M, sequential compile Mar 29 2013 19:17:17
…
This run took 0.387113 seconds.
BeatBox 1.2 finished at t=71738 by device 11 "stop"
Thu Jun 20 20:57:40 2013
=====================================================
687 20:57:40 FitzHughNagumo_model$
```

Conditionals in BeatBox scripts

```
// If the X11 graphics is switched off, then
// these clock devices would still print time labels to the stdout,
// so the "if" clause at the front disables that.

// This shows the integer step counter.
if Graph clock   when=often color=WHITE row0=1 col0=1;

// This shows real model time, the row, col coords of these are in
// characters, not pixels
if Graph k_clock when=often color=WHITE row0=1 col0=21 code=t*ht
                format="T=%4.1f ms";
```

- Syntax: if <k-variable> <device-command>;
- Semantics: iff the k-variable is zero at *parse time*, the device command is skipped (not inserted into the ring).
- In this example: Graph is a global read-only variable which is 1 iff run-time graphics is switched on.

clock device

```
// This shows the integer step counter.
if Graph clock   when=often color=WHITE row0=1 col0=1;
```

- Very simple output device: displays the current value of t counter in the graphical window if one is defined, or prints it to standard output if it is not.
- In this case, in the non-graphical mode it will not be running at all, because of the if Graph clause.
- WHITE is a predefined read-only k-variable which equals 15, the VGA code for white colour. Note that the default background of the graphical window is black.
- Parameters row0 and col0 are measured in characters, not in pixels. This device will put the t=... message in the top row of the screen, from the leftmost position.

k_clock device

```
// This shows real model time, the row, col coords of these are in
// characters, not pixels
if Graph k_clock when=often color=WHITE row0=1 col0=21 code=t*ht
                format="T=%4.1f ms";
```

- Slightly more sophisticated version of clock device. This one will print not just t but the value of any k-expression, given by the code parameter (string), evaluated at run time.
- You can control the way it is printed using format parameter, also a string. In our case, it will be a fixed-point representation to one decimal place, preceded by "T=" and succeeded by " ms".
- Note that the value of format parameter here is enclosed in double quotes, "…". This is necessary because it contains a space.
- It will be shown starting from character position col0=21 which leaves plenty enough space on the left for the message of the clock device.

k_draw device

```
// Draw the phase trajectory
k_draw when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // Defines the part of the graph
                                          // window for this output.
  color=WHITE*16+WHITE      // Colour of the "window" border.
  absmin=Umin absmax=Umax   // Limits for the abscissa
  ordmin=Vmin ordmax=Vmax   //   and for the ordinate of the plot.
  lines=0.5              // Join the dots unless jump for 1/2 of the
                                 // window or more.
  pgm={                  // Program that defines the drawing algorithm.
  col=WHITE*mod(t,1000)/1000;    // Colour will be cycling through VGA palette
                                 // ever 1000 steps.
  abs=U;            // Abscissa is the U variable.
  ord=V;            // Ordinate is the V variable.
  };
```

- This is one of the “essential” run-time graphical output devices.
- It draws a curve on the screen, by adding one segment to it at a time.
- Remember often is nonzero at every 10th time step; this is how often this device will be executed.

k_draw device

```
// Draw the phase trajectory
k_draw when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // Defines the part of the graph
                                          // window for this output.
  color=WHITE*16+WHITE      // Colour of the "window" border.
  absmin=Umin absmax=Umax   // Limits for the abscissa
  ordmin=Vmin ordmax=Vmax   //   and for the ordinate of the plot.
  lines=0.5              // Join the dots unless jump for 1/2 of the
                                 // window or more.
  pgm={                  // Program that defines the drawing algorithm.
  col=WHITE*mod(t,1000)/1000;    // Colour will be cycling through VGA palette
                                 // ever 1000 steps.
  abs=U;            // Abscissa is the U variable.
  ord=V;            // Ordinate is the V variable.
  };
```

- Parameters col0, col1, row0, row1 (measured in pixels) define the area of the graphical screen to which this device will draw.
- Note their values are k-variables homonymic to the parameters. This is choice rather than law: on the left, are device parameters, on the right are k-variables (could be any k-expressions), completely different things!

k_draw device

```
// Draw the phase trajectory
k_draw when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // Defines the part of the graph
                                          // window for this output.
  color=WHITE*16+WHITE      // Colour of the "window" border.
  absmin=Umin absmax=Umax   // Limits for the abscissa
  ordmin=Vmin ordmax=Vmax   //   and for the ordinate of the plot.
  lines=0.5              // Join the dots unless jump for 1/2 of the
                                 // window or more.
  pgm={                  // Program that defines the drawing algorithm.
  col=WHITE*mod(t,1000)/1000;    // Colour will be cycling through VGA palette
                                 // ever 1000 steps.
  abs=U;            // Abscissa is the U variable.
  ord=V;            // Ordinate is the V variable.
  };
```

- Parameter color defines the colour of the frame to be drawn, designating the “sub-window”, the area of the window allocated for this device. The exotic form in which it is specified is an atavism and is due for elimination in future releases.

k_draw device

```
// Draw the phase trajectory
k_draw when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // Defines the part of the graph
                                          // window for this output.
  color=WHITE*16+WHITE      // Colour of the "window" border.
  absmin=Umin absmax=Umax   // Limits for the abscissa
  ordmin=Vmin ordmax=Vmax   //   and for the ordinate of the plot.
  lines=0.5              // Join the dots unless jump for 1/2 of the
                                 // window or more.
  pgm={                  // Program that defines the drawing algorithm.
  col=WHITE*mod(t,1000)/1000;    // Colour will be cycling through VGA palette
                                 // ever 1000 steps.
  abs=U;            // Abscissa is the U variable.
  ord=V;            // Ordinate is the V variable.
  };
```

- The horizontal coordinate of this device is called abscissa, and the vertical is called ordinate (not to confuse with x and y used in many other places).
- Parameter absmin=Umin means that the leftmost edge of the device sub-window will correspond to the abscissa value Umin (which is a k-variable assigned earlier). Likewise for absmax, ordmin, ordmax.

k_draw device

```
// Draw the phase trajectory
k_draw when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // Defines the part of the graph
                                          // window for this output.
  color=WHITE*16+WHITE      // Colour of the "window" border.
  absmin=Umin absmax=Umax   // Limits for the abscissa
  ordmin=Vmin ordmax=Vmax   //   and for the ordinate of the plot.
  lines=0.5             // Join the dots unless jump for 1/2 of the
                                // window or more.
  pgm={                 // Program that defines the drawing algorithm.
  col=WHITE*mod(t,1000)/1000;    // Colour will be cycling through VGA palette
                                // ever 1000 steps.
  abs=U;            // Abscissa is the U variable.
  ord=V;            // Ordinate is the V variable.
  };
```

- This parameter controls whether the points of the curve will be joined or not. The rule is: if the distance between successive points, relative to the sub-window size, exceeds line (one half, in our case), then it will not be joined. This is to reduce mess when discontinuous data are fed to this device.

k_draw device

```
k_draw when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // Defines the part of the graph
                                          // window for this output.
  color=WHITE*16+WHITE      // Colour of the "window" border.
  absmin=Umin absmax=Umax   // Limits for the abscissa
  ordmin=Vmin ordmax=Vmax   //   and for the ordinate of the plot.
  lines=0.5              // Join the dots unless jump for 1/2 of the
                                 // window or more.
  pgm={                  // Program that defines the drawing algorithm.
  col=WHITE*mod(t,1000)/1000;    // Colour will be cycling through VGA palette
                                 // ever 1000 steps.
  abs=U;            // Abscissa is the U variable.
  ord=V;            // Ordinate is the V variable.
  };
```

- The block parameter pgm defines what is to be plotted and how. The syntax is the same as in k_func device, but here it can (should!) use local variables abs, ord and col, which define the coordinates of the next point and its colour.
- As with k_func, the right-hand sides here are expressions that will be evaluated at *run time*. Note that the colour col depends on t!

update device

```
update when=often;  // This signals that the graphics output buffer is
                    // flushed to the screen.
```

- This is a simple but important device: it tells X-windows to update the contents of the graphics screen. Usually you would like to do that at every step at which you did some outputs to the screen, which is the condition when=often here is the same as in the preceding clock, k_clock and k_draw devices. If you forget this device, you will be disappointed as you will not see anything in your graphical window.
- This is not made automatic since sometimes it may be better to update not on every step when there was graphics, or even more than once on some steps.

record device

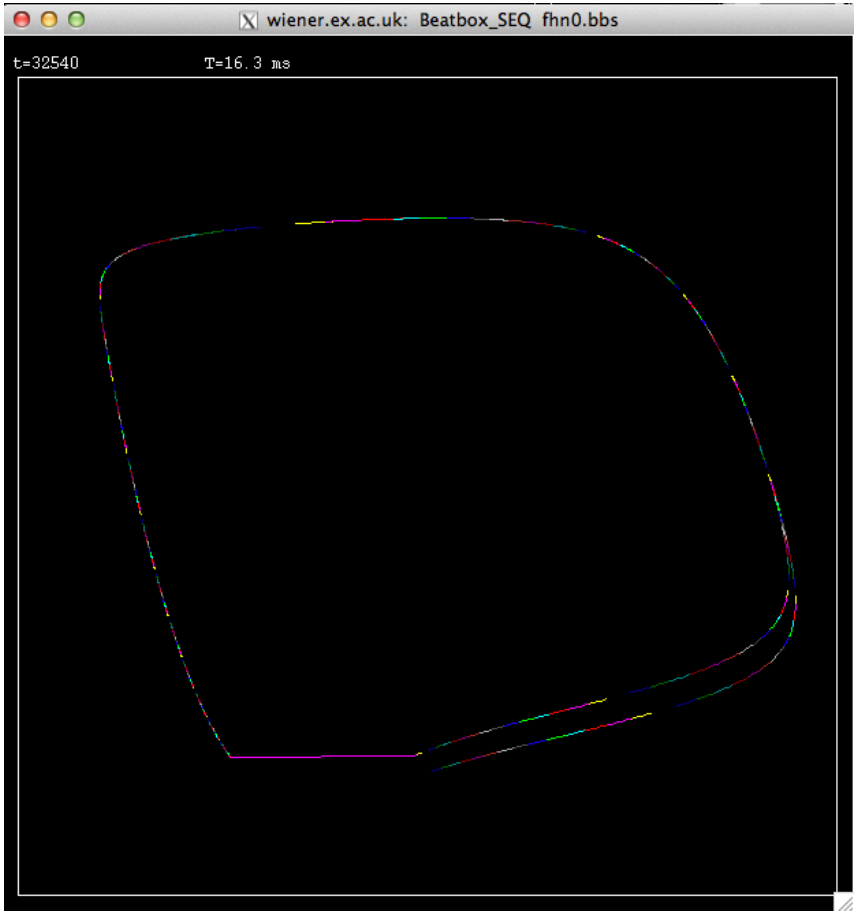
```
 // flushed to the screen.

// Output (record) all FHN dynamical variables into the file fhn0.rec
// at the time step "chosenpulse".
record when=chosenpulse file=[0].rec append=0 v0=[u] v1=[v];
```

- This is one of the textual (rather than binary or graphical) output devices.
- The function is to print, in human-readable form, the values in the device's space (in this case just the values from layer [u]=0 through to layer [v]=1 of the only point of the grid) to the specified disk file.
- The file will be overwritten since append=0; otherwise it would be appended to.
- The string macro reference [0] here refers to the name of the script without the bbs extension if any, so in our case the value of parameter file will be fhn0.rec. More about this sort of macros later.

Snapshot of graphics from fhn0.bbs

- The window title shows hostname on which the program was executed and the command line.
- Near the top, we see the current value of t and the value of T=t*ht.
- The white frame shows the rectangle corresponding to *u* from Umin to Umax and to *v* from Vmin to Vmax.
- And within that frame is the main output, the curve. The colour changes with every point so we can tell fast pieces of the phase trajectory from slow pieces.



Textual output from fhn0.bbs

- And this is the beginning of the file fhn0.rec produced by the record device. It has two values per line, which are the *u* and *v* values, so record has produced one line per call.
- The separators between layer-values, x-, y- and z-values and between individual outputs ('records') are changeable parameters (all two-character strings, by law). In this case we had the default vsep=" " and recordsep="\n".
- This record file will be used in the next script we consider.

-0.334310	-0.581535
-0.329961	-0.580533
-0.325563	-0.579525
-0.321117	-0.578511
-0.316620	-0.577491
-0.312073	-0.576465
-0.307475	-0.575433
-0.302823	-0.574395
-0.298119	-0.573350
-0.293360	-0.572300
-0.288545	-0.571243
-0.283675	-0.570179
-0.278747	-0.569109
-0.273761	-0.568032
-0.268716	-0.566949
-0.263610	-0.565859
-0.258443	-0.564761
...	

MORE ADVANCED SCRIPTS: FHN1

fhn1.bbs

- Location: data/scripts/sequential/FitzHughNagumo_model/fhn1.bbs
- Protocol: initiate propagating pulses in a 1D cable with FHN kinetics, using non-homogeneous Dirichlet boundary conditions on the left end of the cable. The values for these conditions are from the file fhn0.rec, obtained as a results of the previous run of fhn0.bbs. The n-th action potential, n=4, is output to file fhn1.rec. We also measure and print the propagation speed.

New features:

- <..> to include contents of another bbs file;
- `..` to capture output of a child process;
- Use of *record files* and *magic variables* in k_func device;
- k_plot run-time graphics device: to plot spatial profiles of the dynamic fields at selected moments of time.
- k_poincare control device to detect arrivals of wavefronts at selected points, which is then used both to control execution and to calculate the propagation speed;
- pause control device for suspending execution.

Inclusion of another script

```
// Include the file fhn.par with input parameters.
<fhn.par>

// the size of 1D strand
def int nx 100;
```

With account of the contents of fhn.par, this results in the following effective text:

```
def real ht 0.005;
def real hx 1.0/3.0;
def real D 1.0;
def real eps 0.30; def real bet 0.71;
def real gam 0.50;
def str u 0;
def str v 1;
def str i 2;
def real umin -2.0; def real umax 2.0;def real umid 0.0;
def real vmin -1.0; def real vmax 1.5;def real vmid 0.5;
def int nx 100;
```

Capturing output of another process

```
// Get the pacelength = number of lines in file fhn0.rec.
// This is needed to make the period of waves here the same
// as the period of oscillations generated by fhn0.bbs.
def real pacelength `cat fhn0.rec | wc -l`;
...
k_func name=bc x0=1 x1=1 file=fhn0.rec pgm={phasep=(2*pi*t)/pacelength;u0=p0;};
```

- In this way, the real variable pacelength equals the total number of lines in fhn0.rec.
- This number is used later in the name=bc k_func device which implements the non-homogeneous Dirichlet boundary condition.

Record files and magic variables in k_func

```
k_func name=bc x0=1 x1=1 file=fhn0.rec pgm={phasep=(2*pi*t)/pacelength;u0=p0;};
```

- This function assigns a prescribed value to the 0^{th} layer (*u*-variable) of the leftmost internal point of the grid, thus imposing Dirichlet boundary conditions at that point.

Record files and magic variables in k_func

```
k_func name=bc x0=1 x1=1 file=fhn0.rec pgm={phasep=(2*pi*t)/pacelength;u0=p0;};
```

- This function will also “know” the contents of the given file. This file will be read, and interpreted as a space-separated table. The number of columns in the table is detected based on the first line.
- The contents of this file can then be used for the work of the ‘magic variable” phasep.

Record files and magic variables in k_func

```
k_func name=bc x0=1 x1=1 file=fhn0.rec pgm={phasep=(2*pi*t)/pacelength;u0=p0;};
```

- This local variable phasep has the following property. Every time it is assigned a value within the program of the k_func device, this magically leads to calculation of local variables p0, …, p<*N*-1>, where *N* is the number of columns in the record file. Say, if there are three columns, assignment of a value to phasep will result into assignment of certain values to p0, p1, and p2.
- The assigment occurs by the following rule. For variable p<*j*>, the contents of the j^{th} column is interpreted as a real-valued function, defined on the interval $[0,2\pi)$. So p<*j*> is assigned the value of this function at the point phasep mod 2π (using linear interpolation).

Record files and magic variables in k_func

```
k_func name=bc x0=1 x1=1 file=fhn0.rec pgm={phasep=(2*pi*t)/pacelength;u0=p0;};
```

- As a result, the contents of the 0th layer of the point x=1, i.e. the leftmost value of the *u*-variable of the FHN 1D model, at any moment of time will be the same as the value of the *u*-variable in the FHN 0D simulation, only periodically continued in time.
- Note that for this to work as described, one line in fhn0.rec should correspond to one time step in the current model simulation. This is `incidentally' true for our two simulations: the time step in the fhn0.bbs was ht=0.0005, and the output to fhn0.rec was at every 10th time step, and the time step in this simulation is ht=0.005, i.e. precisely 10*0.0005. If that was not the case, we would need to correct the formula for phasep correspondingly.

k_plot device

```
…
k_plot name=uplot when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // defines the part of the graph window
  color=WHITE*16+WHITE              // colour of the "window" border
  lines=1                        // connect the dots
  clean=1                                // clean window before drawing this graph
  ordmin=umin ordmax=umax           // limits for the ordinate of the plot
  N=nx                         // the abscissa will be integer running from 1 to nx
  pgm={ord=u(abs,0,0,[u]);col=LIGHTRED};  // ordinate is value of u variable
…
```

- This is similar to k_draw device considered earlier, only this one draws a whole curve at a time, rather than extending the curve point by point from one call to the next.

k_plot device

```
…
k_plot name=uplot when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // defines the part of the graph window
  color=WHITE*16+WHITE              // colour of the "window" border
  lines=1                        // connect the dots
  clean=1                                // clean window before drawing this graph
  ordmin=umin ordmax=umax           // limits for the ordinate of the plot
  N=nx                         // the abscissa will be integer running from 1 to nx
  pgm={ord=u(abs,0,0,[u]);col=LIGHTRED};  // ordinate is value of u variable
…
```

- The difference from the k_draw interface is that device parameter N is required (which will also be local read-only k-variable), and the abscissa of the curve is fixed: it is an integer number running from 1 to N.
- In this example, the abscissa will run from 1 to nx inclusive, i.e. though the integer coordinates of the inner points of the grid.

k_plot device

```
…
k_plot name=uplot when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // defines the part of the graph window
  color=WHITE*16+WHITE              // colour of the "window" border
  lines=1                        // connect the dots
  clean=1                                // clean window before drawing this graph
  ordmin=umin ordmax=umax           // limits for the ordinate of the plot
  N=nx                         // the abscissa will be integer running from 1 to nx
  pgm={ord=u(abs,0,0,[u]);col=LIGHTRED};  // ordinate is value of u variable
…
```

- Correspondingly, local k-variable abs in k_plot is read-only, and the assignment only done for the ordinate ord and colour col.
- In this example, the ordinate will be the value of the layer [u]=0 (i.e. *u*-variable) at the point with x-coordinate given by the abscissa abs, and the colour of the curve is constant and is light-red (VGA code 9).

k_plot device

```
…
k_plot name=uplot when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // defines the part of the graph window
  color=WHITE*16+WHITE              // colour of the "window" border
  lines=1                        // connect the dots
  clean=1                                // clean window before drawing this graph
  ordmin=umin ordmax=umax           // limits for the ordinate of the plot
  N=nx                         // the abscissa will be integer running from 1 to nx
  pgm={ord=u(abs,0,0,[u]);col=LIGHTRED};  // ordinate is value of u variable
…
```

- Finally, device parameter clean defines whether the sub-window of this device should be cleaned to its black background before plotting the curve. In this case, clean=1 means yes: the profile of the u-variable will be plotted `on the clean slate'.

k_plot device

```
…
k_plot name=vplot when=often
  col0=col0 col1=col1 row0=row0 row1=row1 // in the same window
  color=WHITE*16+WHITE                    //
  lines=1                                 //
  clean=0                                 // but do not clean it beforehand
  ordmin=vmin ordmax=vmax                 // limits for the ordinate
  N=nx                                    //
  pgm={ord=u(abs,0,0,[v]);col=LIGHTBLUE}; // which is now v variable, and plotted
…
```

- The next k_plot device plots the profile of the *v*-variable ([v]=1) in light blue colour, in addition to the previously plotted profie of the *u*-variable. Hence no need to clean the sub-window, an clean=0.

k_poincare device

```
sample name=U1 x0=xout1 v0=[u] result=U1;
sample name=U2 x0=xout2 v0=[u] result=U2;
…
// These devices will register arrival of the fronts at points xout1 and xout2
// as variables U1 and U2 have been measured there
k_poincare nowhere=1 which=0 sign=1 pgm={front1=U1-umid;tfront1=t};
k_poincare nowhere=1 which=0 sign=1 pgm={front2=U2-umid;tfront2=t};
```

- This control device implements “Poincare cross-section” of the given stream of data. This means it watches a set of dynamic variables, registers the moments when one of them crosses a prescribed value of 0, and reports the values of other variables at those moments.
- The variables that are watched are the right-hand sides of of the k-program pgm of this device: U1-umid and t in one case, U2-umid and t in the second case.
- The way the results are reported may be somewhat unexpected so requires some attention.

k_poincare device

```
sample name=U1 x0=xout1 v0=[u] result=U1;
sample name=U2 x0=xout2 v0=[u] result=U2;
…
// These devices will register arrival of the fronts at points xout1 and xout2
// as variables U1 and U2 have been measured there
k_poincare nowhere=1 which=0 sign=1 pgm={front1=U1-umid;tfront1=t};
k_poincare nowhere=1 which=0 sign=1 pgm={front2=U2-umid;tfront2=t};
```

- First of all, there is an important device parameter which which defaults to 0 and is not present in the script. With this parameter made explicit, the device commands will look as shown.
- This parameter defines which of the variables is watched for cross-section. The variables are enumerated starting from 0, so this is U1-umid in the first case and U2-umin in the second case. Hence the first device will register when U1 crosses umid, and the second device when U2 crosses umid.

k_poincare device

```
sample name=U1 x0=xout1 v0=[u] result=U1;
sample name=U2 x0=xout2 v0=[u] result=U2;
…
// These devices will register arrival of the fronts at points xout1 and xout2
// as variables U1 and U2 have been measured there
k_poincare nowhere=1 which=0 sign=1 pgm={front1=U1-umid;tfront1=t};
k_poincare nowhere=1 which=0 sign=1 pgm={front2=U2-umid;tfront2=t};
```

- When this happens, all other assignments, except the one pointed by which, will actually take place. There is just one "other assignment" in each case here. That is, tfront1 will be assigned to the current value of t when U1 crosses umid, and tfront2 will be assigned to the current value of t when U2 crosses umid.
- More precisely, since the very moment of crossing will have happened *between* the calls, the assigned value will be obtained by linear interpolation of the value *before* and *after* the crossing.
- That assignment will be done, naturally, *after* the crossing happens. On all other calls, the values of tfront1 and tfront2 remain unchanged.

k_poincare device

```
sample name=U1 x0=xout1 v0=[u] result=U1;
sample name=U2 x0=xout2 v0=[u] result=U2;
…
// These devices will register arrival of the fronts at points xout1 and xout2
// as variables U1 and U2 have been measured there
k_poincare nowhere=1 which=0 sign=1 pgm={front1=U1-umid;tfront1=t};
k_poincare nowhere=1 which=0 sign=1 pgm={front2=U2-umid;tfront2=t};
```

- It remains to discuss the first assignment, the one pointed at by which parameter. If it is easy to see, if this is done by the general rule, then the left-hand sides would always be assigned 0 when crossing has happened, or the value would remain unchanged if it has not. Not very interesting.
- Therefore, this assignment is done by a different rule: the left-hand side (k-variables front1 and front2) will be given value 1 if the crossing has happened, and 0 if it has not. In this way, these variable front1 and front2 can be used as condition variables for other devices, say which we want to process the results of these crossings. ..

k_poincare device

```
k_poincare nowhere=1 sign=1 pgm={front1=U1-umid;tfront1=t};
k_poincare nowhere=1 sign=1 pgm={front2=U2-umid;tfront2=t};

// Count the fronts separately, calc arrival times and select Nth front in each
sequence
k_func nowhere=1 when=front1 pgm={nfront1=nfront1+1;
T1=if(eq(nfront1,Np),ht*tfront1,T1)};
k_func nowhere=1 when=front2 pgm={nfront2=nfront2+1;
T2=if(eq(nfront2,Np),ht*tfront2,T2);
  paceout=eq(nfront2,Np); /* if it is the last pulse we output it */
  end=end+gt(nfront2,Np); /* and if more then it is time to stop; '+' works as
"or" */
};
```

- … like here: next k_func will count the times front was recorded at the first point, and remember the time at which the front number Np has been registered as T1. And the next one is measuring T2. This is what we need to measure the speed!

pause device

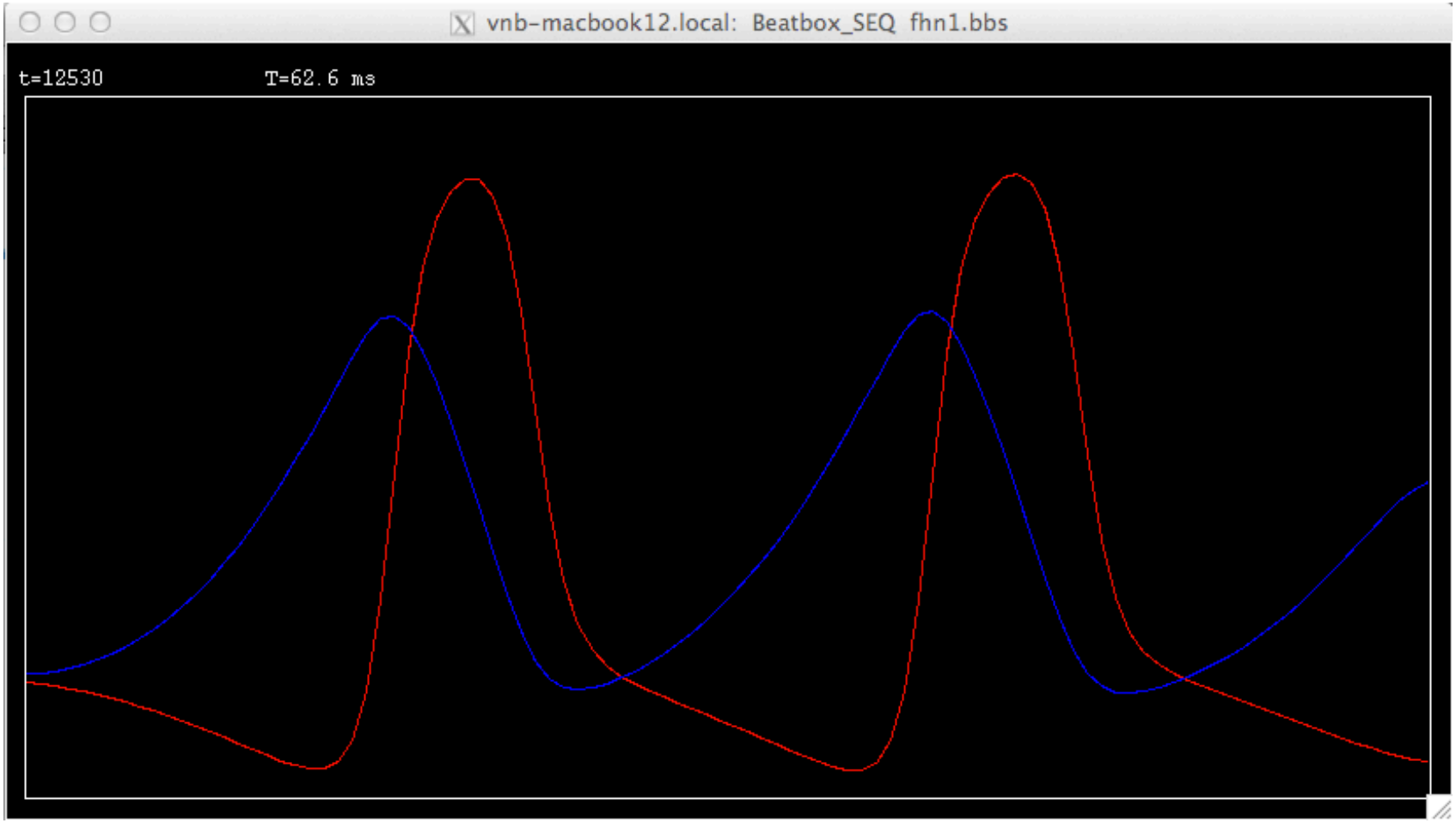
```
// Output the record of dynamical variables u and v at the "samplepoint"
// while "paceout" is nonzero into the file fhn1.rec.
k_print nowhere=1 when=paceout file=[0].rec append=0 valuesep="\t" list={U2;V2};

// Keep the picture on the screen until the user presses Enter
if Graph pause when=end;

/* Stopping criterion. This is mandatory.*/
stop when=end;
end;
```

- … Finally, a simple control device which suspends execution for a given number of seconds (specified by device parameter seconds). The default is -1 (used in this example) which means the program will stop until the user hits Enter in the shell window from which the program was launched.
- This is needed to let user see the final picture in the graphical screen, so it is needed only if graphics is on.

Snapshot of graphics from fhn1.bbs



MORE ADVANCED SCRIPTS: FHN2

fhn2.bbs

- Location: data/scripts/sequential/FitzHughNagumo_model/fhn2.bbs
- Protocol: initiate a spiral wave in a 2D sheet using phase distribution method, and let the spiral rotate, while calculating and outputting the time derivative and the gradient of the *u*-variable, and the trajectory of the tip. New features:
- more extensive use of k-variables to save typing and ease modification;
- use of k_func as a computational device for creating intial conditions by phase distribution method
- computational devices d_dt and grad2d: time derivative and absolute value of spatial gradient of a field;
- k_paint: run-time graphical device to crudely visualize distribution of 2D dynamic fields;
- singz device: to detects spiral wave tips (both computational and output)
- shell output device: to call arbitrary OS command from BeatBox;
- imgout and k_imgout output devices: to output image files.

Phase distribution method for initial conditions

```
…
// Initial conditions by phase distribution method
k_func when=begin  name=initial file=fhn1.rec pgm={
  phasep=atan2(x-x0,y-y0) - 2*pi*hypot(x-x0,y-y0)/lam;
  u0=p0; u1=p1
};
…
```

- Similar to use of k_func in fhn1.bbs for non-homogeneous Dirichlet boundary conditions: the magic variable phasep represents the phase, and it serves as the key for picking up the right row from the table in fhn1.rec.
- There the phase variable phasep was defined as a linear function of time. Here it is a function of x and y such that curves phasep(x,y)=const are Archimedean spirals.

Phase distribution method for initial conditions

```
…
// Initial conditions by phase distribution method
k_func when=begin  name=initial file=fhn1.rec pgm={
  phasep=atan2(x-x0,y-y0) - 2*pi*hypot(x-x0,y-y0)/lam;
  u0=p0; u1=p1
};
…
```

- The same effect could be achieved by using another magic variable, phaseu. It works just like phasep, but make assignments straight to variables u<*j*> (i.e. grid values) rather than local variables p<*j*> :

```
…
// Initial conditions by phase distribution method
k_func when=begin  name=initial file=fhn1.rec pgm={
  phaseu=atan2(x-x0,y-y0) - 2*pi*hypot(x-x0,y-y0)/lam;
};
…
```

About string macros again

```
// The reaction substep: [i] gives the extra term in the right-hand side for u.
// We do not need the resting state here so drop the 'rest' parameter.
euler v0=[u] v1=[v] ht=ht ode=fhncubpar
     par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]};
```

- In this fragment, the string macro i is expanded after the layer substitution operator @. Since this string macro has been defined earlier as

```
def str i 2; // diffusion term will be in layer 2
```

(in fhn.par), the above idiom is equivalent to

```
Iu=@2
```

which means that the value of parameter Iu, at every step and at every point, will be taken from layer 2 – that is where diff has put the value of the diffusion term.

d_dt device

```
k_func name=timing nowhere=1 pgm={ /* Define when to begin and end */
  begin = eq(t,0); // beginning of simulation
  out   = eq(mod(t,tout),0); // time to make outputs every tout steps
  dtime = out + eq(mod(t,tout),tout-1); // when to call d_dt device
  // NB '+' above works as logical 'or'
  end   = ge(t,50000); // end of simulation
};
…
d_dt when=dtime v0=[u] v1=[p] vd=[d] ht=ht;
```

- This device calculates the time derivative of a field recorded in layer v0 and put the result into layer vd. The differentiation backward Euler, i.e. it is simply the difference between the current and the previous value, divided by the time step ht. Layer v1 will be used by this device to store that previous value. The value of ht should be equal to the time step ht of the solver device euler if d_dt is called at consecutive steps.

d_dt device

```
k_func name=timing nowhere=1 pgm={ /* Define when to begin and end */
  begin = eq(t,0); // beginning of simulation
  out   = eq(mod(t,tout),0); // time to make outputs every tout steps
  dtime = out + eq(mod(t,tout),tout-1); // when to call d_dt device
  // NB '+' above works as logical 'or'
  end   = ge(t,50000); // end of simulation
};
…
d_dt when=dtime v0=[u] v1=[p] vd=[d] ht=ht;
```

- In this example, d_dt is called when=out *and also at preceding steps*, so at the steps when=out the above condition is satisfied: the interval between successive calls is 1 step. And this is when it matters, since when=out is condition for all outputs.

grad2d device

```
grad2d when=out v0=[u] v1=[g] hx=hx;
```

- This device calculates the absolute value of the gradient of a 2D field recorded in layer v0 and put the result into layer v1. The differentiation is by central differences, and hx is the space step. This is an instantaneous operation so this device can be called when the result is needed for output or control.
- Note that the ratio of the time derivative of a field to the gradient of that field gives the velocity of the isoline (aka contour line) of that field. So when the gradient is not too small, this can be used for (almost) instantaneous estimates of the conduction speed.

k_paint device

```
// Run-time graphics to paint the field using VGA colour palette
k_paint when=out nabs=nx nord=ny          // how many abscissa and ordinate
values
  col0=col0 col1=col1 row0=row0 row1=row1 // output it into the left panel
  color=WHITE*16+WHITE                    // white borders for the panel
  pgm={                                   // program should calculate colour
                                          // and we take abs=x and ord=y
  col=ifge0(u(abs,ord,0,[u]),LIGHTRED,0)  // red component f u > 0
  + ifge0(u(abs,ord,0,[v]),GREEN,0)};     // and green if v > 0
// NB: lightred+green=yellow, and absence of both is black.
```

- This graphical device continues the line of k_draw and k_point. Now both abs and ord are “independent variables”, running from 1 to nabs and nord respectively, and the only functional dependence is for local variable col, the VGA colour.
- Another difference: it renders not points or lines, but rectangles, so filling in areas in the graphical window.

k_paint device

```
// Run-time graphics to paint the field using VGA colour palette
k_paint when=out nabs=nx nord=ny          // how many abscissa and ordinate
values
  col0=col0 col1=col1 row0=row0 row1=row1 // output it into the left panel
  color=WHITE*16+WHITE                    // white borders for the panel
  pgm={                                   // program should calculate colour
                                          // and we take abs=x and ord=y
  col=ifge0(u(abs,ord,0,[u]),LIGHTRED,0)  // red component f u > 0
  + ifge0(u(abs,ord,0,[v]),GREEN,0)};     // and green if v > 0
// NB: lightred+green=yellow, and absence of both is black.
```

- The image is rendered into a subwindow defined by device parameters col0, col1, row0, row1, which are pixel coordinates within the BeatBox X11 graphics window. The marginal values of abs and ord correspond to left/ right and bottom/top extremities of the subwindow.
- As can be seen from the previous def commands in the script, global k-variables col0 and col1 correspond to the left half of the graphics window.

singz device

```
// Detect the tip of the spiral
singz when=out
  v0=[u] c0=0          // trace intersection of
  v1=[v] c1=0          // u(x,y,t)=0 and v(x,y,t)=0
  col0=col2 col1=col3 // plot it
  row0=row0 row1=row1 // in the right panel
  color=WHITE*16+LIGHTRED // red trace with white head
  file=[0].trj         // and write into file fhn2.trj
;
```

- This device finds the tip(s) of the spiral wave(s), defined as intersection(s) of isolines of two given fields, v0 and v1, at two given levels, c0 and c1 respectively.

singz device

```
// Detect the tip of the spiral
singz when=out
  v0=[u] c0=0          // trace intersection of
  v1=[v] c1=0          // u(x,y,t)=0 and v(x,y,t)=0
  col0=col2 col1=col3 // plot it
  row0=row0 row1=row1 // in the right panel
  color=WHITE*16+LIGHTRED // red trace with white head
  file=[0].trj         // and write into file fhn2.trj
;
```

- The visualization of the found tips will be done into the subwindow defined by device parameters col0, col1, row0, row1, so that the device space (here the default one) maps onto this subwindow.
- According to def commands made earlier, global k-variables col2 and col3 correspond to the right half of the graphics window.

singz device

```
// Detect the tip of the spiral
singz when=out
  v0=[u] c0=0          // trace intersection of
  v1=[v] c1=0          // u(x,y,t)=0 and v(x,y,t)=0
  col0=col2 col1=col3 // plot it
  row0=row0 row1=row1 // in the right panel
  color=WHITE*16+LIGHTRED // red trace with white head
  file=[0].trj         // and write into file fhn2.trj
;
```

- The visualization is done with dots, using a colour for the most recent tip(s) different from those of the previous tips. The device parameter color codes both of them: the VGA colour codes are numbers from 0 to 15, so one such colour makes one hexadecimal digit. The device parameter color has range from 0 to 255, making two hexadecimals. So the recent tip colour is the most significant hex digit of the parameter, and the tail colour is the least significant digit.

singz device

```
// Detect the tip of the spiral
singz when=out
  v0=[u] c0=0          // trace intersection of
  v1=[v] c1=0          // u(x,y,t)=0 and v(x,y,t)=0
  col0=col2 col1=col3 // plot it
  row0=row0 row1=row1 // in the right panel
  color=WHITE*16+LIGHTRED // red trace with white head
  file=[0].trj         // and write into file fhn2.trj
;
```

- This device will also output the data of calculated tip(s) into the file if given.
- In this example, the name of the output file will be fhn2.trj since prefedefined string macro call [0] expands to the name of the script without bbs extension.

The format of the singz text output

- In this example, the output consists of 5 columns: the x, y coordinates of the tip, orientation angle of the gradient of the first field, same for the second field, and cosine of the difference between the two angles.
- The coordinates of the tip are found by bilinear interpolation of the two fields. The angles are those the gradient vectors make with the x axis, in radians.
- There are several parameters to control the output, but that is for a more detailed talk.

```
50.757942 50.763065 0.783060 -0.630097 0.156987
51.842682 52.343216 0.980653 -0.667252 -0.077032
51.964546 53.362553 1.208952 -0.328245 0.033593
51.616615 54.403961 1.363628 0.091823 0.294556
50.858425 55.406269 1.509650 0.339408 0.389929
49.923473 56.288189 1.678501 0.463238 0.348090
48.889259 56.966110 1.870139 0.606913 0.302744
47.863079 57.396740 2.086076 0.796302 0.277338
46.799255 57.524334 2.301500 1.032696 0.297423
45.804020 57.338772 2.529980 1.304789 0.338766
44.961288 56.865452 2.780592 1.601239 0.381522
44.327560 56.223251 3.055104 1.894736 0.399002
43.926010 55.404961 -2.933982 2.207295 0.415860
43.791050 54.558804 -2.623706 2.515665 0.414126
43.940025 53.706367 -2.311339 2.840225 0.425193
44.350746 52.973225 -1.990245 -3.115053 0.431350
44.936375 52.400650 -1.654592 -2.795286 0.416964
45.714485 52.025787 -1.324978 -2.464945 0.417624
46.558998 51.880894 -1.003678 -2.160234 0.402495
47.442360 51.985394 -0.702522 -1.869566 0.392872
```

shell device

```
// Create the directory for the image files
// (no problem if it is already there)
def str outdir [0].dir; // so it will be fhn2.dir
shell nowhere=1 when=never advance=1 cmd="mkdir [outdir]";
// advance=1 means this will be done BEFORE the first step
```

- This device calls a sub-shell to execute a daughter process defined by command string cmd. In that way it is similar to the `…` preprocessing directive, only `…` is done once at parse time, whereas shell command is execute every time the device is run.
- In this particular example, however, for this device when=never (which is permanently tied to zero). But there is a device parameter advance, which if not zero, means that the subshell call is executed right after parsing this device command.

shell device

```
// Create the directory for the image files
// (no problem if it is already there)
def str outdir [0].dir; // so it will be fhn2.dir
shell nowhere=1 when=never advance=1 cmd="mkdir [outdir]";
// advance=1 means this will be done BEFORE the first step
```

- In this example, the command is in creating a directory with the name [outdir]=[0].dir=fhn2.dir. This is the directory into which the subsequent imgout and k_imgout devices will place their outputs. Since those devices check that they can open the output files at parse time, so creation of the directory should be at parse time.

imgout device

```
imgout when=out
  // On-the-fly conversion including flipping top/bottom flip
  filter="pnmflip -tb | pnmtopng > [outdir]/uvi%07.0f.png"
  r=[u]  r0=umin  r1=umax // [u]-layer defines red component
  g=[v]  g0=vmin  g1=vmax // [v]-layer defines green component
  b=[i]  b0=-1    b1=1;  // [i]-layer defines blue component
```

- The function and interface of this device are similar to that of ppmout considered earlier, only here instead of writing the resulting ppm file to disk, we pipe it into a filter, which can be any external command that take the ppm format as input stream.
- The filter parameter is a string which can contain a % sign, which then is interpreted as a C format specifier. This means that the actual filter is obtained each time by a sprintf function, using filter as a format string, and the sprintf'ed expression is given by device parameter string code, which defaults to t, the time loop counter.
- In this case, the output file names will be

  fhn2.dir/uvi0000100.png, fhn2.dir/uvi0000100.png, …

imgout device

```
imgout when=out
  // On-the-fly conversion including flipping top/bottom flip
  filter="pnmflip -tb | pnmtopng > [outdir]/uvi%07.0f.png"
  r=[u]  r0=umin  r1=umax // [u]-layer defines red component
  g=[v]  g0=vmin  g1=vmax // [v]-layer defines green component
  b=[i]  b0=-1    b1=1;  // [i]-layer defines blue component
```

- In this case, the actual filters will be, time after time,

```
pnmflip -tb | pnmtopng > fhn2.dir/uvi0000000.png
pnmflip -tb | pnmtopng > fhn2.dir/uvi0000100.png
                                                …
```

- Programs pnmflip and pnmtopng are executables from the netpbm toolkit freely available for all platforms. The first program flips top of the immage with the bottom, thus converting pixel coordinates in which the smallest y is at the top to the normal orientation. The second program converts ppm format or any of its netpbm relatives to the standard png format, which is then directed to the said files in the fhn2.dir subdirectory.

k_imgout device

```
k_imgout when=out
  filter="pnmflip -tb | pnmtopng > [outdir]/udg%06.0f.png"
  filtercode="t/tout"  // fancy numerating files sequentially
  width=nx height=ny   // i=0..width-1, j=0..height-1
  pgm={ // this program should calculate r,g,b as real numbers in [0..1]
     r=(u(1+i,1+j,0,[u])-umin)/(umax-umin);
     g=fabs(u(1+i,1+j,0,[d]))/2.0;
     b=u(1+i,1+j,0,[g])/2.0;
  };
```

- This is a more sophisticated version of imgout, in which the rgb components are not just linear transformation of grid values at certain layers, but arbitrary k-expressions.
- The local “independent” variables for these expressions are i and j for horizontal and vertical coordinates, running from 0 to width-1 and from 0 to height-1 respectively, where width and height are device parameters.

k_imgout device

```
k_imgout when=out
  filter="pnmflip -tb | pnmtopng > [outdir]/udg%06.0f.png"
  filtercode="t/tout"  // fancy numerating files sequentially
  width=nx height=ny   // i=0..width-1, j=0..height-1
  pgm={ // this program should calculate r,g,b as real numbers in [0..1]
     r=(u(1+i,1+j,0,[u])-umin)/(umax-umin);
     g=fabs(u(1+i,1+j,0,[d]))/2.0;
     b=u(1+i,1+j,0,[g])/2.0;
  };
```

- The expression for the red component corresponds to the same mapping as before: the value in layer [u]=0 (for the *u*-variable) is linearly mapped to 0 if it equals umin, and to 1 if it equals umax. The values of the expression is automatically cropped to the interval [0..1] in any case, so *u*-values below umin will all map to 0 and all above umax will all map to 1.

k_imgout device

```
k_imgout when=out
  filter="pnmflip -tb | pnmtopng > [outdir]/udg%06.0f.png"
  filtercode="t/tout"  // fancy numerating files sequentially
  width=nx height=ny   // i=0..width-1, j=0..height-1
  pgm={ // this program should calculate r,g,b as real numbers in [0..1]
     r=(u(1+i,1+j,0,[u])-umin)/(umax-umin);
     g=fabs(u(1+i,1+j,0,[d]))/2.0;
     b=u(1+i,1+j,0,[g])/2.0;
  };
```

- For the blue component, the expression depends only on layer [g], where the gradient values are kept, so 0 -> 0, 2 -> 0. This also could have been done by imgout.

k_imgout device

```
k_imgout when=out
  filter="pnmflip -tb | pnmtopng > [outdir]/udg%06.0f.png"
  filtercode="t/tout"  // fancy numerating files sequentially
  width=nx height=ny   // i=0..width-1, j=0..height-1
  pgm={ // this program should calculate r,g,b as real numbers in [0..1]
     r=(u(1+i,1+j,0,[u])-umin)/(umax-umin);
     g=fabs(u(1+i,1+j,0,[d]))/2.0;
     b=u(1+i,1+j,0,[g])/2.0;
  };
```

- For the green component, it is something that could not be done by imgout: the expression involves nonlinear function fabs for (floating point) absolute value. So the green component will be proportional to/cropped the absolute value of the time derivative of the *u*-field.

k_imgout device

```
k_imgout when=out
  filter="pnmflip -tb | pnmtopng > [outdir]/udg%06.0f.png"
  filtercode="t/tout"  // fancy numerating files sequentially
  width=nx height=ny   // i=0..width-1, j=0..height-1
  pgm={ // this program should calculate r,g,b as real numbers in [0..1]
     r=(u(1+i,1+j,0,[u])-umin)/(umax-umin);
     g=fabs(u(1+i,1+j,0,[d]))/2.0;
     b=u(1+i,1+j,0,[g])/2.0;
  };
```

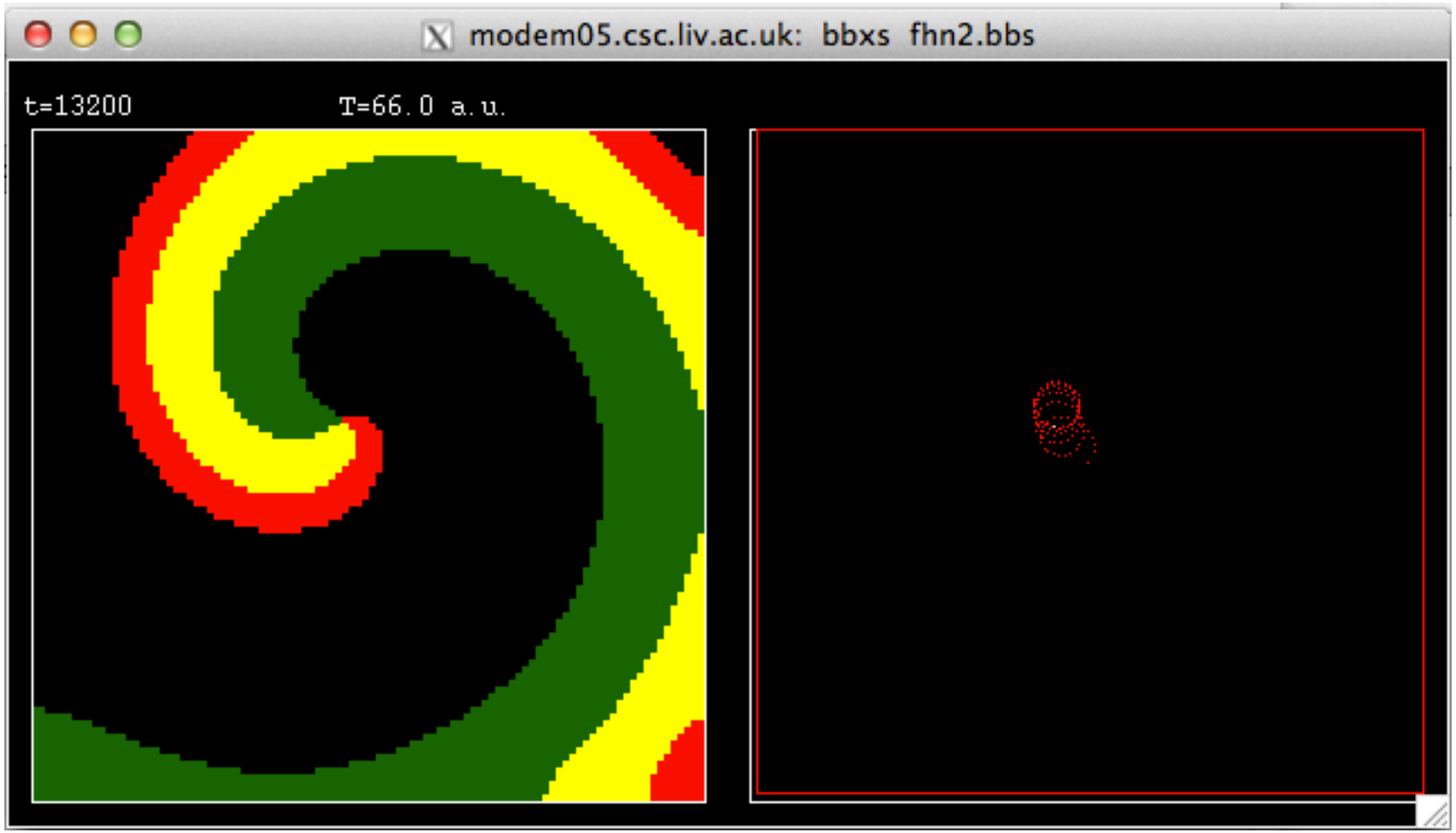
- Note that in all cases, the functions for the colour components are such that the pixels in the output correspond to the inner points of our 2D grid. This makes sense in most cases but syntactically is not necessary. The same device could be used for 2D visualization of 3D calculations by doing cross-sections of 3D field by surfaces defined by arbitrary parametric equations.

k_imgout device

```
k_imgout when=out
  filter="pnmflip -tb | pnmtopng > [outdir]/udg%06.0f.png"
  filtercode="t/tout"  // fancy numerating files sequentially
  width=nx height=ny   // i=0..width-1, j=0..height-1
  pgm={ // this program should calculate r,g,b as real numbers in [0..1]
     r=(u(1+i,1+j,0,[u])-umin)/(umax-umin);
     g=fabs(u(1+i,1+j,0,[d]))/2.0;
     b=u(1+i,1+j,0,[g])/2.0;
  };
```

- And the final touch: the output filter code, here called filtercode, is given value t/tout rather than default t. Since when=out is nonzero precisely at every tout steps, this leads to the output files being numbered consecutively, as udg000000.png, udg000001.png, udg000002.png …

Snapshot of graphics from fhn2.bbs



END