

Czochralski Process Development and Control with DSIM

*Richard Walker**

Abstract

This paper presents a simple timestep-based simulator (DSIM) providing the core functionality of Labview in less than 700 lines of C. Linear and non-linear analog elements such as integrators, differentiators, gain elements, comparators, PID controllers and so forth are typically implemented in less than a dozen lines of C-code. A Czochralski crystal growth is simulated using a simple crystal growth model and simulated furnace thermal response. A simple Recipe subroutine is called at every simulation point to update crystal Diameter and Average Pull Rate based on table lookup. After the simulation algorithm is refined using a synthetic model, it is anticipated that each software stub be replaced by an equivalent call to actual hardware. This can be done in stages. For example, the Temperature control loop can be run on a real furnace with the IR temp sense data being fed to a synthetic software crystal growth model. Because the core simulator is stripped of all the unnecessary graphic overhead of Labview, the resulting system is easily modified, fully transparent in operation, and capable of ultra-high reliability.

1 Introduction

Simulating a complex system can easily turn into a tangled mess of computer code. However, the simple algorithm described here is capable of handling all the communication between multiple system components in a structured, guaranteed stable way. The algorithm (called DSIM) and was invented and first described as a simulator for non-linear bang-bang phased-lock-loop circuits¹. The methodology for simulation and control used in DSIM will be quite familiar to users acquainted with other systems such as Labview and SPICE. A simple DSIM implementation of a Czochralski crystal growth only requires 700 lines of C, including the simulator itself and all required models!

To setup a DSIM simulation, a functional block diagram of the system must be created. Each of the wires or signals interconnecting each block is given a unique name and number. The functionality of each block is captured in a C-subroutine. Each subroutine will be called at every time step. The parameters to the subroutines will be the node number of the interconnecting signals for that node. The number of each node indexes into a data structure which allows the block to access the process value for that signal for the previous and current time step. In addition, a future value is expected to be updated for each block which has an output signal. The core algorithm handles all the communication between functional blocks, timestep control and data I/O.

The next section describes the core algorithm in detail and gives examples of a number of different functional blocks.

*Richard Walker, Consulting. <walker@omnisterra.com>

¹ Richard Walker, "Clock and Data Recovery for Serial Data Communications", pp 62-65, BCTM tutorial, September 27, 1988. (<http://www.omnisterra.com/walker/pdfs/talks/bctm2.maker.pdf>).

2 Simulation Algorithm

The core of the simulator is a set of three floating point arrays, indexed by node numbers. The three arrays contain the current, past and calculated future value of every state variable used by the simulation. The code below allocates one master table of size $3 \times \text{NUMNODES}$ and then indexes into the table by three base pointers called “old”, “node”, and “new”. The reason for using pointers will be obvious later as the algorithm is described.

```
double  nodetable[3*NUMNODES];
double *old, *node, *new;

main() {
    ...
    for (i=0; i<3*NUMNODES; i++) {
        nodold = nodetable;
    }
    old = nodetable;
    node = nodetable+NUMNODES;
    new = nodetable+2*NUMNODES;
    ...
}
```

The behavior of every element in the simulation must be defined by C-code subroutine. Each element will be called at each timestep and has certain responsibilities. The element will be called with an argument list of the node numbers to which it is connected. The element will use these numbers to index into the three arrays to compute a new output value. Consider a simple amplifier that could be described by the following prototype:

```
void amplifier(input, output, gain, ref)
int input, output;
double gain, ref;
...
```

The amplifier can access the current input voltage (or process variable) by “node[input]”, the value of the process variable at the previous timestep by “old[input]”, and can set its output by writing to “new[output]”. Here is the code for the body of the amplifier

```
new[out] = (gain * (node[in] - reference)) + reference;
```

Because the two previous timestep values are always available for every node, it is possible to implement a discrete time version of any second order system. For example, here is the code for an integrator using trapezoidal integration:

```
void integrate(in, out, gain)
int in, out; double gain;
{
    double chunk; /* new integrated portion of signal */
    chunk = ((gain/2.0) * (old[in] + node[in]) * stepsize);
    new[out] = node[out] + chunk;
}
```

The integration is performed by taking the previous value of the input voltage “old[in]”, adding it to the current value of the input voltage “node[in]”, dividing by two and then adding it to the current output voltage “node[out]”. The new value is then used to update the value of the output node for the next (future) timestep “new[out]”.

Each element is called in any order at each time step. The network topology is restricted to have one and only one element driving each node. Many elements may share the same input node. All the elements can be thought of as having infinite input impedance and zero output impedance.

The core control algorithm for the simulator is simple. It simply calls update repeatedly to allow each block to update the new[] array based on the values of the old[] and current node[] array values. Then it simply shifts the pointers so that the current node[] array becomes the old array, and the newly written new[] values become the current node[] values. Because the global arrays were defined with node pointers, there is no actual copying that occurs. The pointers are simply shuffled:

```

for (simtime=STARTTIME; simtime<=STOPTIME; simtime=simtime+dt) {
    update();          /* update nodes each tstep */
    if (simtime-SAVETIME >= savestep*points_plotted) {
        output();
        points_plotted++;
    }
    /* swap pointers to avoid copying data arrays */
    temp = old;
    old = node;
    node = new;
    new = temp;
}

```

The output() routine can be called at full or decimated resolution to write a log file that can then be plotted with a graphing tool such as pdplot².

3 Crystal Growth Example

A block diagram for the Adema crystal growth system is shown in Figure 1.

A system definition file for this block diagram is shown below:

```

#define DD      1      // Diameter Target
#define D       2      // actual Diameter
#define PP      3      // APR target
#define P       4      // actual APR
#define TSET    5      // TEMP target
#define T       6      // actual TEMP
#define LEN     7      // length
#define KVA     8      // KVA
#define APR     9      // APR
#define DAVG   10     // average diameter

void update() /* responsible for updating node[] */ {
    recipe(LEN, DD, PP);
    model(P, LEN, T, D);          /* calculate d, given p,l,t */
    dia(D, DAVG);                /* filter diameter data */
    pid(DD, DAVG, P, &pid1);      /* setpoint, current, drive */
    integrator(P, LEN, 1.0/3600.0); /* convert APR to length */
    apr(P, APR, 300.0);
    tps(APR, PP, TSET);
    pid(TSET, T, KVA, &pid2);     /* setpoint, current, drive */
    therm(KVA, T);                /* thermal time constant */
}

```

The order of the blocks is arbitrary. The simulation/control algorithm manages all the interconnecting signals. Each node must be driven by exactly one block. We will now go through and describe each block.

² www.omnisterra.com/linux/pdplot

ADEMA Furnace Control System
 07/26/06

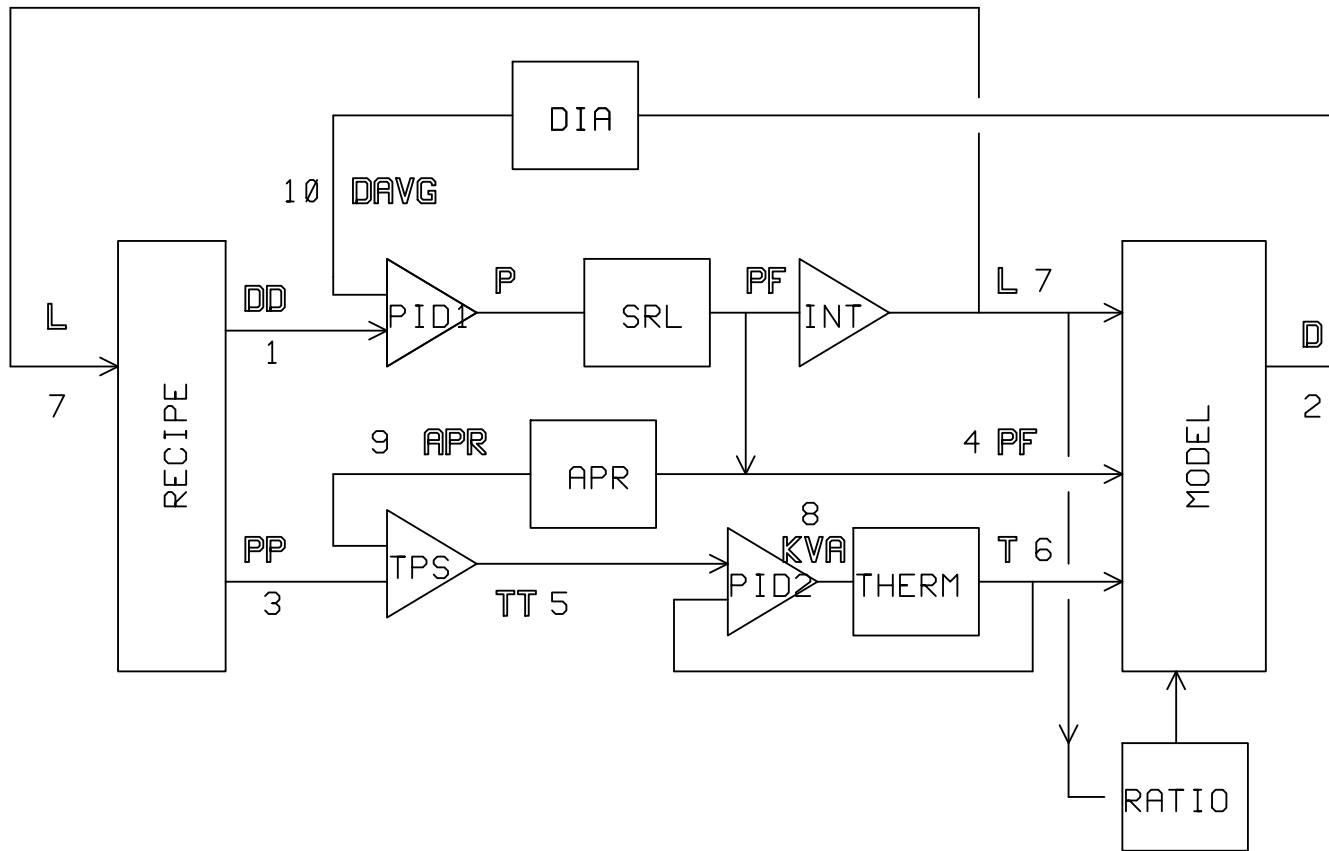


Fig. 1: Adema Czochralski Growth Algorithm (body mode)

3.1 Recipe module

The recipe module is responsible for changing system target values as the process proceeds. Primarily it will use the current crystal length (LEN) as the index into the recipe table. For this simple simulation of body growth, the recipe simply takes the length from the crystal growth model (called L in the figure, but LEN in the code), and outputs a target diameter DD and a target pull-rate PP.

```
void recipe(int l, int diameter, int pull) {
    /* later on, these will be functions of l */
    /* for now, just set target diameter and pull rate */
    new[diameter] = 15.0;      /* 15cm */
    new[pull] = 7.0;          /* cm/hour */
}
```

3.2 Crystal Growth module

The crystal growth module is used to simulate how the crystal diameter varies with temperature and pull rate. This simple model is ad-hoc and was not based on deep theory. For simulation accuracy, this is the most critical section of code. The model gets three pieces of information passed to it. They are the current pull rate: P, the current crystal length: LEN, and the current melt temperature: T. It is then responsible for updating the current diameter: D.

The strategy is to compare the melt temperature with the freezing point of Silicon (1400C), and a maximum temperature at which it is assumed that the crystal will simply no longer condense (1800C). Bounds are also put on the minimum and maximum pull rate. It is assumed that a pull rate higher than the maximum will cause the crystal diameter to shrink to zero. We then compute a temperature index “a” which varies between 0 and 1 as the temperature varies from freezing to melting. We also compute a pull rate factor “b” which varies from 0 to 1 as pull rate varies from min to max. The new diameter is then set to a weighted value between DMIN and DMAX based on $\sqrt{a*b}$.

This simple model is sufficient to demonstrate the simulation framework. It however does not consider the time dependence on crystal growth rate or the 2nd order heatsinking effect of length on the critical temperature.

```
void model(p, l, t, d) /* calculate d, based on p,l,t */
int p; /* pullrate */
int l; /* length */
int t; /* temperature */
int d; /* diameter */
{
    /* length dependance is 2nd order, ignore for now */
    /* also, leave out any time dependance for D */
    double TC = 1400.0; /* critical melt temp */
    double TM = 1800.0; /* maximum condensing temp */
    double PMIN = 0.0;
    double PMAX = 14.0;
    double DMIN = 0.0;
    double DMAX = 30.0;
    double x;
    double a,b;
    if (node[t] < TC) { /* below freezing ? */
        a=0;
    } else if (node[t] > TM) { /* beyond condensing ? */
        a=1;
    } else {
```

```

    a=(node[t]-TC)/(TM-TC);
}
if (node[p] < PMIN) {          /* below min pull rate ? */
    b=0;
} else if (node[p] > PMAX) {
    b=1;
} else {
    b=(node[p]-PMIN)/(PMAX-PMIN);
}
new[d] = sqrt(a*b)*(DMAX-DMIN) + DMIN;
}

```

3.3 Diameter filter module

In the legacy Adema 8085 multibus controller, there is a large noise factor on the diameter reading. This module is responsible for doing a moving average on the diameter data to reduce the noise. In addition, it is possible to use various non-linear digital filtering techniques such as median filtering here. For this simulation, we just pass the data through without modification.

```

void dia(int d, int davg) {    /* filter diameter readings */
    new[davg] = node[d];      /* for now, just pass through */
}

```

3.4 PID module

There are two traditional Proportional/Integral/Differential (PID) controllers in the algorithm. These are implemented with the same subroutine called with different PID parameter structures at run time. The parameters are set in an SPid structure

```

typedef struct {
    double dstate;          /* last position input */
    double ystate;          /* integrator state */
    double imax, imin;     /* max, min integrator state */
    double pgain;          /* proportional gain */
    double igain;          /* integral gain */
    double dgain;          /* derivative gain */
} SPid;

SPid pid1;                 /* create pid control structures */
SPid pid2;                 /* create pid control structures */

```

PID control algorithm implementation vary widely. The specific implementation chosen here is one that includes anti-windup on the integral controller, and computes the differential term on the process variable instead of the state variable. This strategy reduces extreme process glitches when there is a step change in command input.

```

void pid(int command, int position, int drive, SPid *pid) {
    double pterm;
    double iterm;
    double dterm;
    double error;
}

```

```

error = node[command]-node[ position ];

pterm = pid->pgain*error;          /* proportional */
pid->istate += error;              /* integral */

if (pid->istate > pid->imax) {      /* anti wind-up */
    pid->istate = pid->imax;
} else if (pid->istate < pid->imin) {
    pid->istate = pid->imin;
}

iterm = pid->igain * pid->istate;
dterm = pid->dgain * (node[ position ] - pid->dstate);
pid->dstate = node[ position ];
new[drive] = (pterm + iterm -dterm);
}

```

At startup, PID1 is initialized to integral windup limits of +/- 100.0 and PID parameters of 0.4, 0.3 and 0.0. PID2 is initialized to the same windup limits and PID parameters of 3.0, 3.0 and 100.0;

3.5 integrator module

The integrator module is responsible for converting the pull rate into length. It is simply a generic trapezoidal integration function with a gain coefficient:

```

void integrator(in, out, gain)
int in;
int out;
double gain;
{
    double chunk; /* new integrated portion of signal */
    chunk = ((gain/2.0) * (old[in] + node[in]) * dt);
    new[out] = node[out] + chunk;
}

```

3.6 APR module

The APR module simply takes snapshots of the pull rate at a fixed interval. By keeping APR fixed for, in this case, 300 seconds or 5 minutes at a time, any instability due to APR updating is eliminated. The legacy controller had a more complex algorithm due to a high error rate in the multibus ADC circuit, but it is not expected that the legacy algorithm will be needed in a new control loop with good data integrity.

```

void apr(int p, int avgpr, double deltatime) {
    /* compute average pull rate */
    static double nextupdate=0.0;
    static double hold;
    if (simtime >= nextupdate) {
        nextupdate+=deltatime;
        hold = node[p]; /* take a snapshot */
    }
    new[avgpr] = hold;
}

```

3.7 TPS module

The function of the TPS module is to create a control input for the kva loop when given a pull rate P, and target pull rate PP. If the current pull rate is low, then the KVA set point will be lowered to produce faster crystalization. If the pull rate is too high, and risking crystal defects, the KVA will be raised to reduce the rate of crystal aggregation.

TPS is non-critical over a range of values. It is important that it rapidly converge without oscillation. For these reasons, the TPS algorithm was chosen to be first order bang-bang loop controller. This type of controller rapidly converges with very small overshoot and is extremely aggressive about maintaining lock. This implementation does not include a deadband, but this might be useful in the actual process controller.

```
void tps(int p, int pp, int tt) {
    double delta;
    if (node[pp] > node[p]) {
        delta = -0.03*dt;
    } else {
        delta = 0.03*dt;
    }
    new[tt] = node[tt]+delta;
}
```

3.8 Thermal module

The thermal module is responsible for modelling the furnace temperature given KVA input over time. The code includes several measured constants. The first set of constants is the maximum allowed KVA (P_{MAX}) and the steady state temperature (T_{MAX}) that would result from maximum input. The second constant is the average chill water temperature which tells the module what the furnace will converge to when no power is applied. Then there is a time constant express in reciprical units of the simulation timestep. From this a simple discrete differential equation gives the furnace temperature as a function of time and KVA assuming a single order time constant response function.

```
void therm(p, t) /* thermal lag of furnace given kva */
int p; /* power in kva */
int t; /* furnace temperature node */
{
    double PMAX = 150.0; /* max furnace KVA */
    double TMAX = 1800.0; /* steady state T at PMAX */
    double TCHILL = 20.0; /* temp of coolant water */
    double E = .001; /* 1/time constant */
    double power;
    if (node[p] > PMAX) {
        power = PMAX;
    } else {
        power = node[p];
    }
    new[t] = ((E*dt)*((TMAX-TCHILL)*(power/PMAX) + TCHILL));
    new[t] += (1.0-(E*dt))*node[t];
}
```

4 Results

Running the simulator involves compiling and linking two files which contain a total of 294 lines of C. The entire build process and simulation time for a 2-hour virtual crystal growth is 0.8 seconds of computer time. The quick

speed to simulate an entire crystal growth process enables rapid process development and allows the system designer to easily gain insight into the process parameters by quickly running many “what-if” scenarios.

The simulations show the actual temperature approaching the target temperature with a first order time constant. Only when the temperature is within the correct process range does the diameter increase towards the target diameter. The time quantization of the APR module is clearly seen. The total length grows monotonically as the APR is integrated, and eventually trends towards a constant growth rate and diameter.

5 Conclusions

The DSIM algorithm is a simple and efficient method for managing the complexity of a complex non-linear control system of multiple interacting blocks. It operates on an easily understood block diagram model and a set of labelled process variables that are the shared signals among the functional blocks. Although this report has described the algorithm in terms of a simulator, it should be evident that the same structure serves as a *control* algorithm just as well.

To convert the simulator into a controller, the algorithm simply sends any computed values from internal computation blocks such as PID to the actual process systems. It is proposed in another document that the control system used should be an RS-485 network to allow the process computer to directly send and receive packets for reading and writing every system hardware component. Any required input to the control algorithm is similarly read from the appropriate hardware unit. For example, this simple control algorithm would require a camera module to read Diameter, an IR temperature sensor to read melt temperature, and a length encoder to compute Length and Pull Rate. The RS-485 network would have commands for setting pull rate, rotation rate, and KVA.

A more complex recipe routine will be required which is capable of interpolating the information currently loaded in by a Datakey as a function of time and length. In addition, the simple block diagram will need to be slightly modified to support seed, neck, shoulder, body and tail-off modes. It is also a simple matter to create operational modes that are partially manual in which certain of the feedback systems (eg: TPS) are turned off and only certain systems enabled (eg: KVA or Temperature control).

The DSIM algorithm is robust and simple, while providing the essential core functionality of more complex, proprietary graphical systems such as Labview. Because DSIM does not come with a mult-megabyte graphical input front-end, the overall reliability of the system is greater than that of a more complex package. Because the number of functional blocks required for furnace control is small, the large library of a commercial system is not advantageous, while the operation of the hand-coded DSIM blocks are easily modified and deeply understood.

For reliability, it is recommended that the control system be architected as two independent programs: a simple DSIM control core running with a very small, reliable set of control code and a second graphical front end that provides the user interface to the core controller. Linux provides numerous interprocess communication primitives to implement this architecture efficiently. The core controller would then provide a textual command interface somewhat like a MySQL server. A developer can debug the core by changing process parameters on the fly by typing commands to the controller. In a running production system, the controller is sent commands by the GUI front end. The advantage of the textual control link is that all process variable updates can be logged to a file for debugging the system operation.

Because the central DSIM controller is separate, it will be able to maintain process stability if the more complex GUI system crashes. A well-designed system will tolerate a crash and quick restart of the GUI backend without any detrimental effect on the process operation. In addition, the modular architecture allows easily changing the GUI without affecting the control algorithm itself.

The complete source code to the DSIM core along with all the ADEMA-specific library modules is available from the author. It is written in ANSI C and compiles under the Gnu C compiler on a Fedora 10 Linux system.

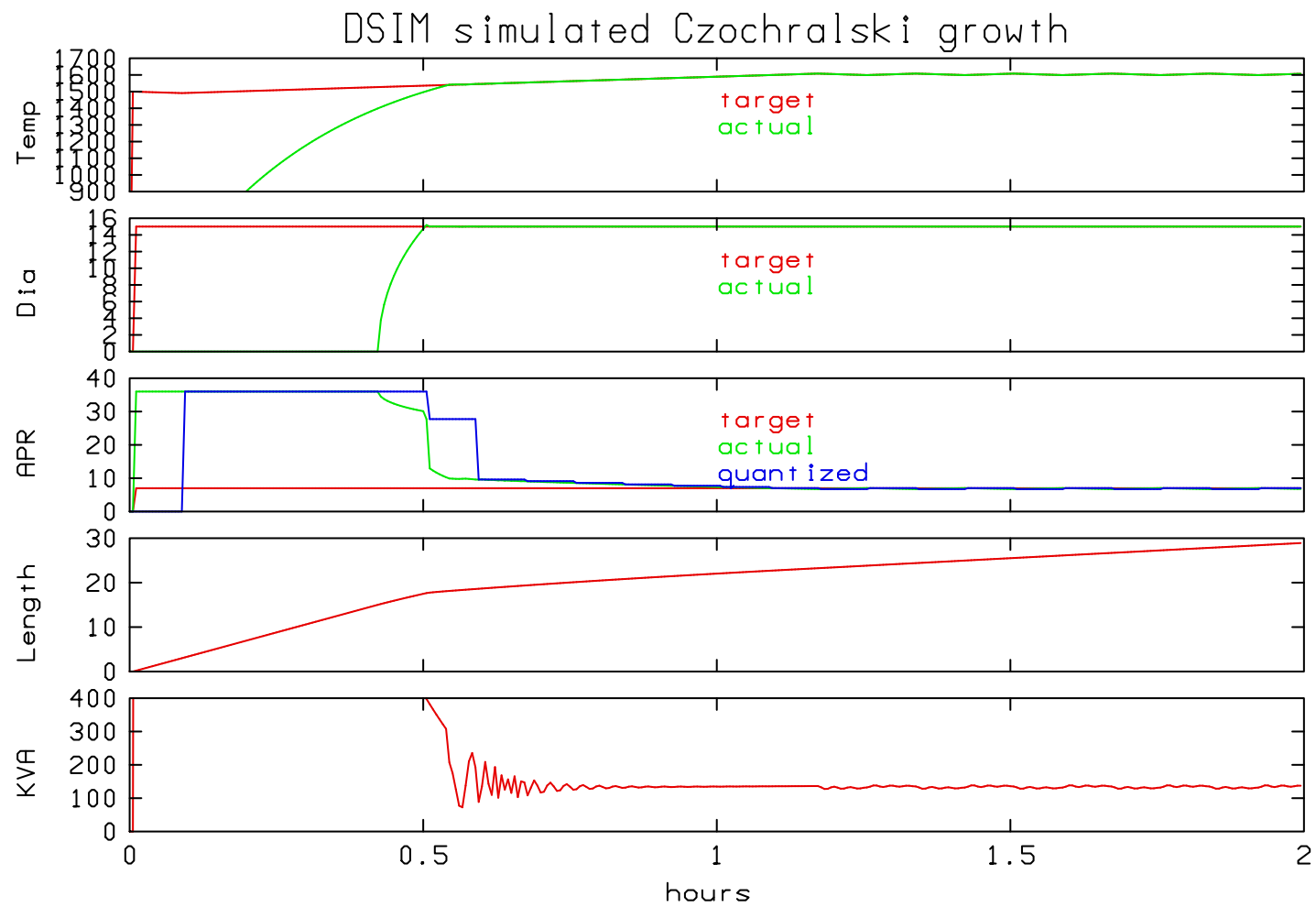


Fig. 2: Adema Czochralski Growth Algorithm (body mode)