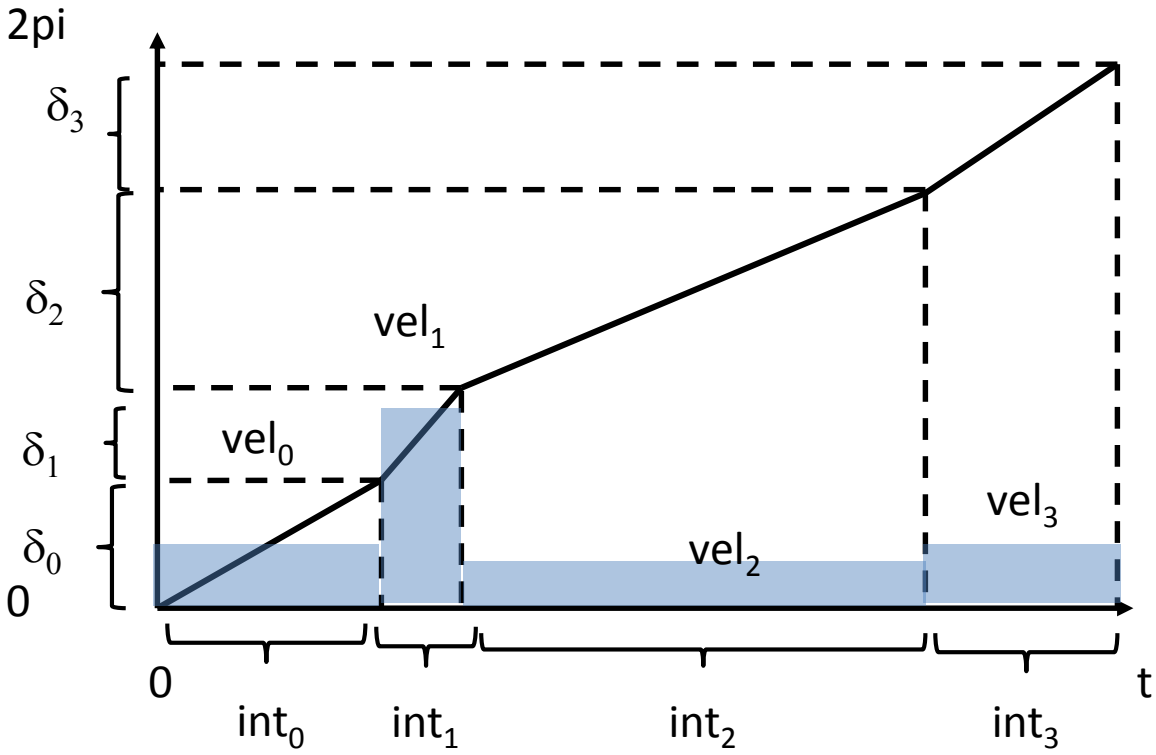


One cycle

at start of interval0:
p_input = p_input + ref position δ_0,
set new δ after every interval (long)
interpolate used to make smoother pos ref setpoints
v_input = vel_0 (should be calibrated to reach δ_0 in int_0 ms)
v_input = ref velocity, set every interval (int)

$vel_i = \delta_i / int_i$

δ= unsigned 16 bit int value
int_i in ms
Python:
velocity
Delta (Hall counts)
Interval (ms)

NOTE: using AustriaMicro Systems AS5048B 14 bit range converted to 16 bit unsigned

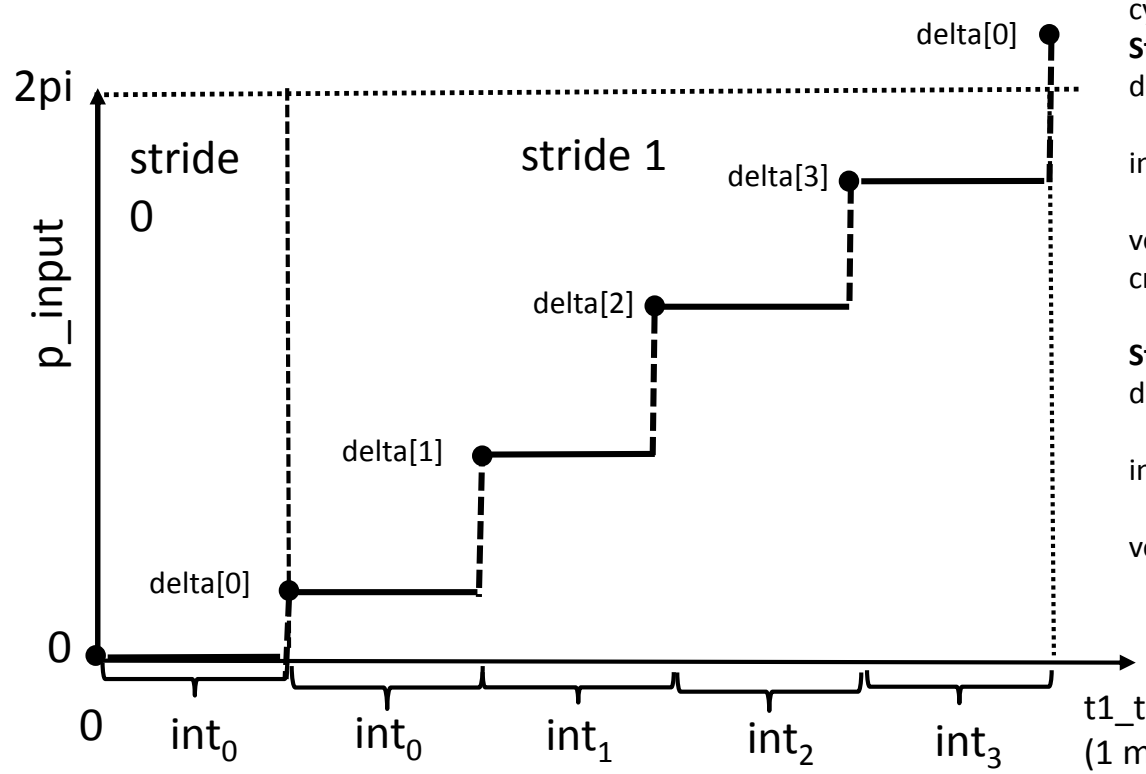
A/D units per rev/sec * 2^16 per ms, scale by >>8
~43/256

$V_{input} = K_{EMF} * vel_i$

$K_{EMF} = 2.50$

Initialization of position setpoint p_input, 45 degrees phase offset

SetThrustClosedLoop() run_time > interval[0]



cycle = 1000 ms

Stride 0: (to get phase offset)

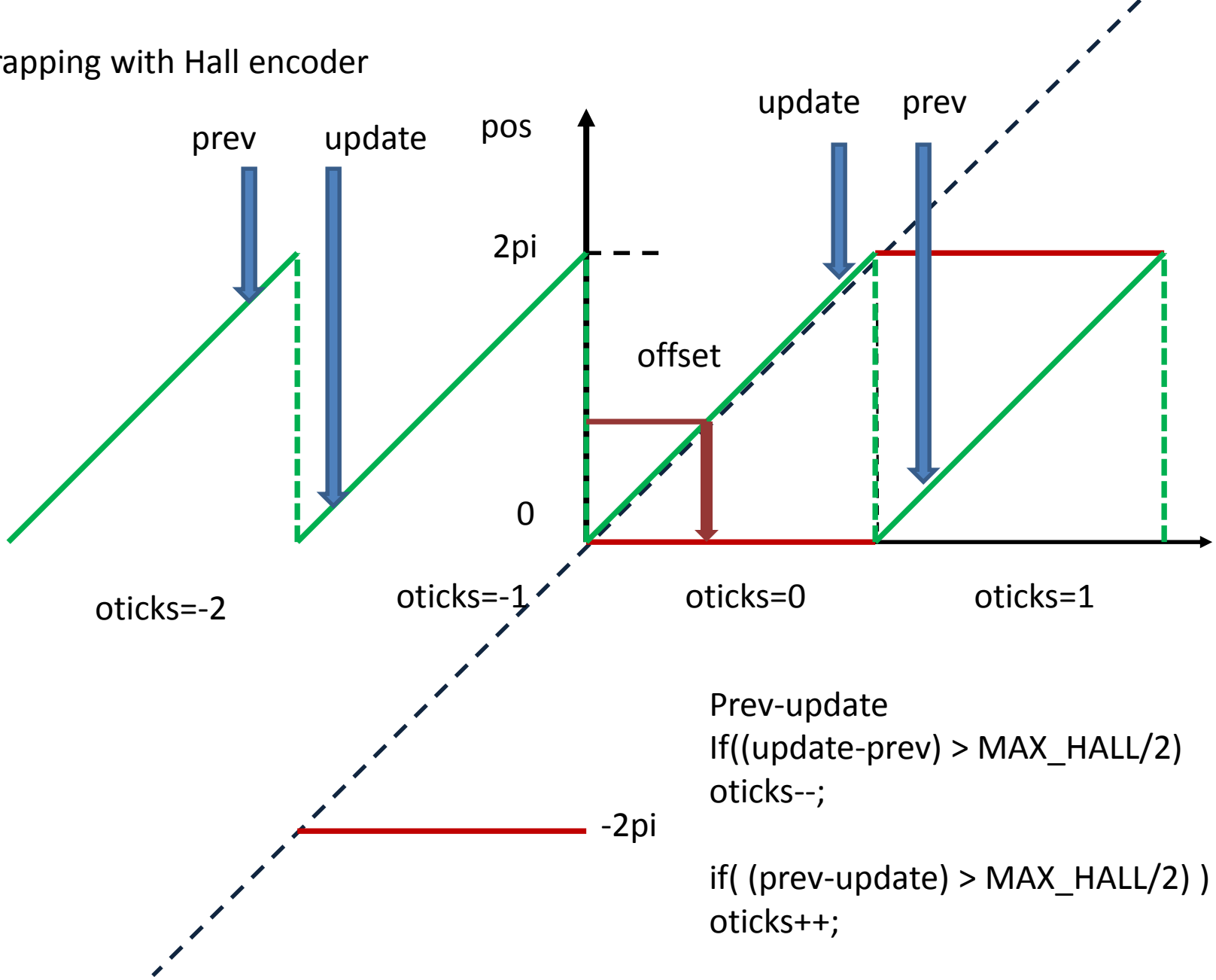
delta=[45,180,270,360] (degrees)
= [8192,24756, 16384,16384]
intervals=[25%,25%,25%,25%]
=[250,250,250,250] ms
vel = [33,99,65,65]
cmdSetThrustClosedLoop(run_time = 250 ms)

Stride 1 to N: (constant velocity)

delta=[90,180,270,360] (degrees)
= [16384, 16384, 16384,16384]
intervals=[25%,25%,25%,25%]
=[250,250,250,250] ms
vel = [65,65,65,65]

main() -> pidSetup() ->
initPIDObjPos(): p_input =0;
initPIDVelProfile(): p_input=0; leg_stride =0;
pidSetInput(pid_num, v_input, run_time):
start_time = t1_ticks; index =0; interpolate=0;
cmdZeroPos() -> pidZeroPos() -> p_input =0; // zeros encoder position and position setpoint
cmdSetThrustClosedLoop() -> pidSetInput(pid_num, v_input, run_time):
start_time = t1_ticks; index =0; interpolate=0;
expire = t1_ticks + interval[0];
T1Interrupt: pidGetSetPoint(): p_input += (long)delta[index]; // update to next setpoint after interval (expire)

2 pi wrapping with Hall encoder



Prev-update
If((update-prev) > MAX_HALL/2)
oticks--;

if((prev-update) > MAX_HALL/2))
oticks++;

offset is subtracted from total position

PID Code pid-rf5.c (9/4/2014)

(from https://github.com/ronf-ucb/turner-ip2.5/blob/cmds/lib/pid-ip2.5.c)

Python:

```
# [Kp Ki Kd Kanti-wind ff]
motorgains = [1000,0,300,0,100, 1000,0,300,0,100]
```

pid-ip2.5c:
at 1 ms rate, {pidGetState(); pidGetSetpoint(); pidSetControl}

Experiment with VelociRoACH transmission	
Velocity (from hall angle sensor) Rad/sec	Back EMF (A/D units)
50	~100
80	~140

pidGetSetPoint: // update desired velocity and position tracking setpoints for each leg. p_input and v_input are set based on `t1_ticks`
pidGetState: // #HALL_SENSOR: 1 if Hall encoder present, prevents code hang if unplugged. If 0, then p_state is not updated.
VEL_BEMF: 1 use back EMF voltage for velocity estimate, if 0, estimate velocity from first difference
pidSetControl: #MAXTHROT 3800 (out of 4095 max. need off time for back emf reading)

long p_error = p_input + interpolate – p_state; // [16].[16], allows only +- 32768 leg steps.
int v_error = v_input – v_state; // back EMF in A/D units, 1 A/D unit ~ 0.5 rad/sec
long i_error[n+1] = i_error[n] + p_error/16; // e.g. a static error of 1 revolution for 16 ticks, would give i_error = 0x1 00 00
Anti windup: if |desired_control | > MAXTHROT, then subtract (Kaw/GAIN_SCALER)*(desired_control - MAXTHROT) from i_error.
command output PWM value is calculated as an int, 16 bits
output = ff + (Kp * p_error)/(2^12) + (Ki * i_error)/(2^16) + (Kd * v_error)/16
Kp, Ki, Kd must be chosen to be have meaningful units, and avoid overflow

Set points:

```
// update desired position between setpoints, scaled by 256, incremented every 1 ms
pidVel[j].interpolate += (long)pidVel[j].vel[index];
//  if time has reached next interpolation point, set new desired velocity and position set point
if (t1_ticks >= pidVel[j].expire)  // time to reach previous setpoint has passed
{                   pidObjs[j].p_input += pidVel[j].delta[index];                   //update to next set point
                    pidObjs[j].v_input = (int)( pidVel[j].vel[(index] * K_EMF)//256; // update to next velocity in A/D units

                    … }
```