Embedded Systems Tutorial 8/Lab 6
Feedback Control, Variable Scope

Tutor: Marcus Pham

marcus.pham@uwa.edu.au
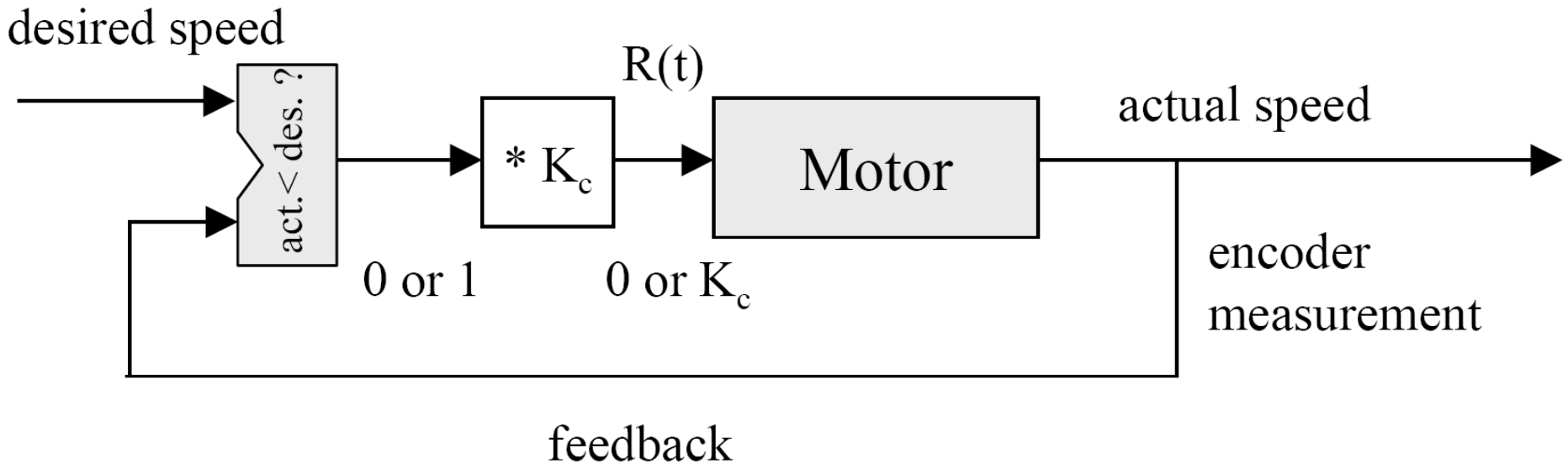
Feedback Control

▸ A control method that takes the system's output as an input

▸ This is used to create an Error Function

▸ Simple on/off (bang bang) control

▸ Proportional control

▸ PID control

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely 2013)

Feedback Control

▸ Example from lecture notes (bang bang control)

desired speed

act. < des. ?

0 or 1

R(t)

* K_c

0 or K_c

Motor

actual speed

encoder measurement

feedback

Bräunl 2008

Bang bang control

- Simplest control method
- Controller has two discrete output states (usu. ON and OFF).
- Called 'bang bang' controller because that's a good description of the output from such a system
- Poor response time
- No steady state is ever achieved
- Cheap to build, can easily be analogue circuit
- eg. Thermostats, water tank pumps
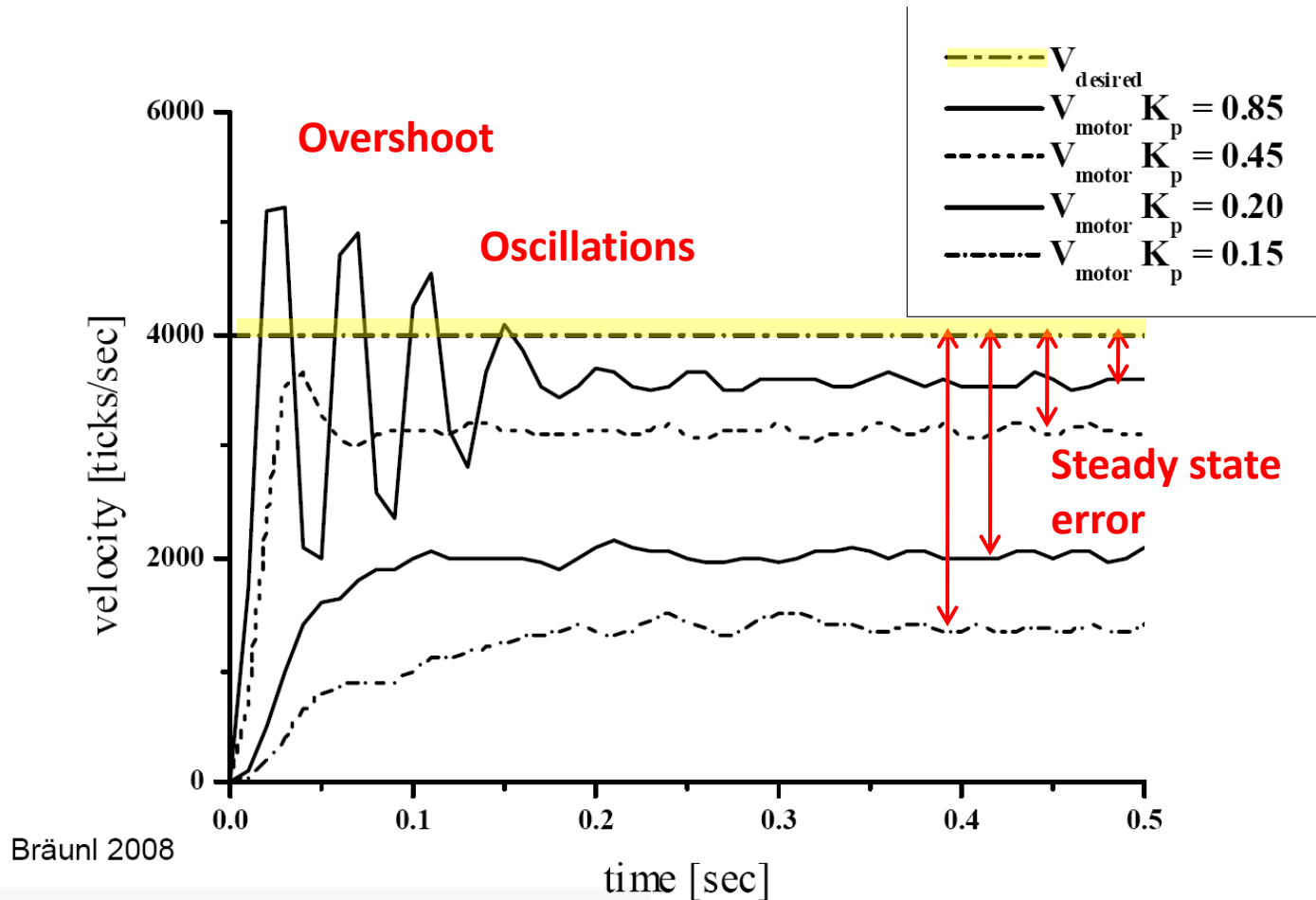
Proportional Controller

- Control using an error function and Control Gain factor.

- Improved control over bang bang

$$R(t) = K_P \times e(t) \qquad\qquad e(t) = v_{des}(t) - v_{act}(t)$$

  - Faster response
  - Smoother response
  - Reaches a steady state

- Disadvantage
  - May never reach desired velocity due to steady state error
  - Can oscillate with poor selection of K_P
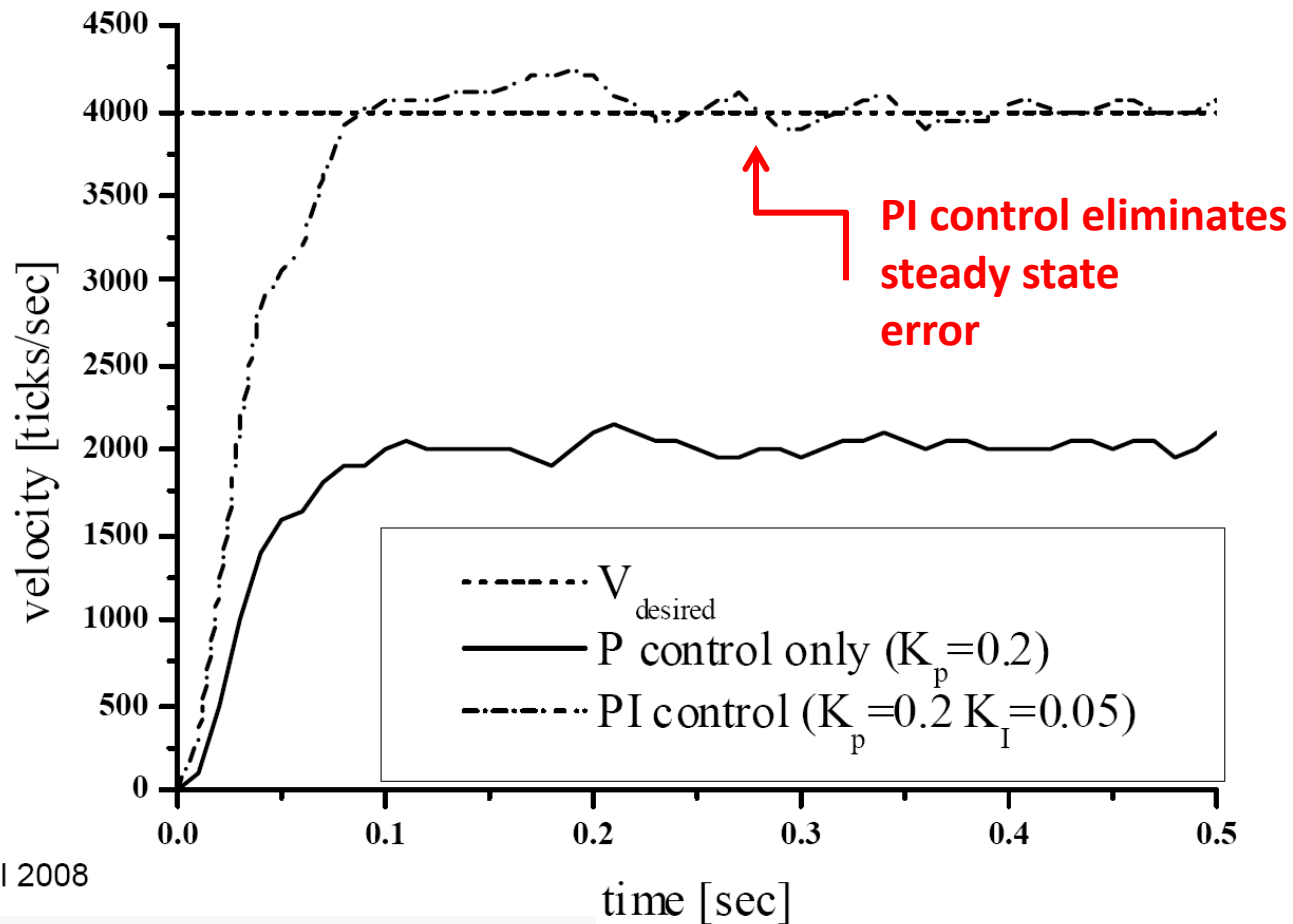
Proportional Controller Response

Bräunl 2008

Proportional Integral Controller

▸ The Integral Controller now adds a component of the Integral of the Error Function.

▸ We now have Hysteresis – the output depends not only on the input but also the previous state of the system.

▸ This is now a discrete function, which is perfect for an iterative approach.

$$R_n = R_{n-1} + K_P \times (e_n - e_{n-1}) + K_I \times (e_n + e_{n-1}) / 2$$

PI Controller Response

PI control eliminates steady state error

Bräunl 2008

velocity [ticks/sec]

time [sec]

$V_{desired}$

P control only (K_p=0.2)

PI control (K_p=0.2 K_I=0.05)

Proportional Integral Derivative Controller
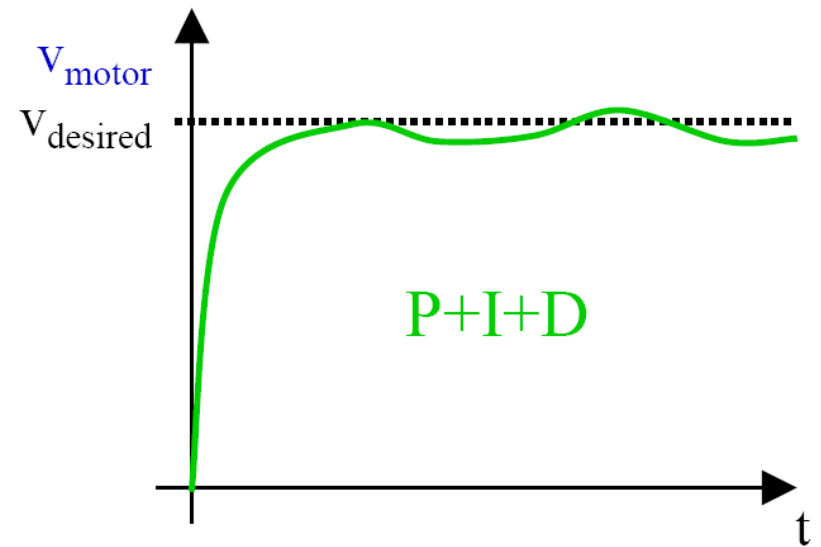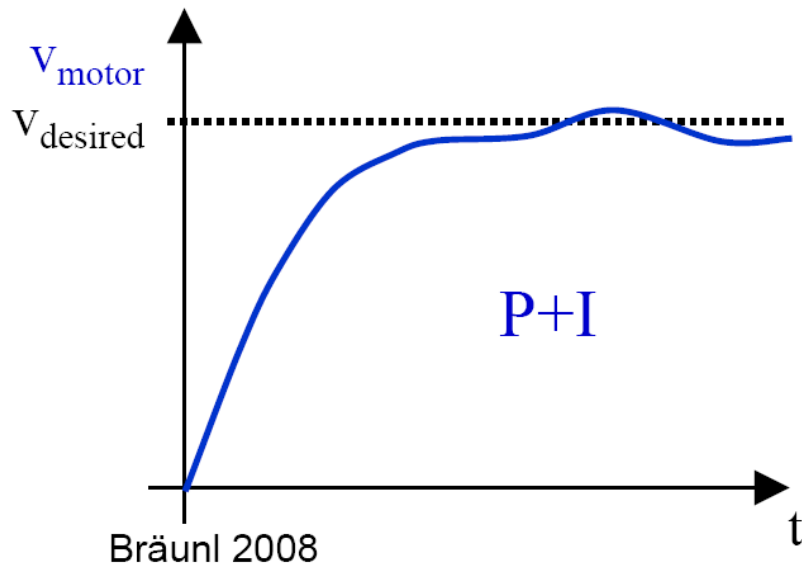
▶ With PI, we now have a decent response with little steady state error. However its response time could be faster and we would like to remove the oscillations in the signal.

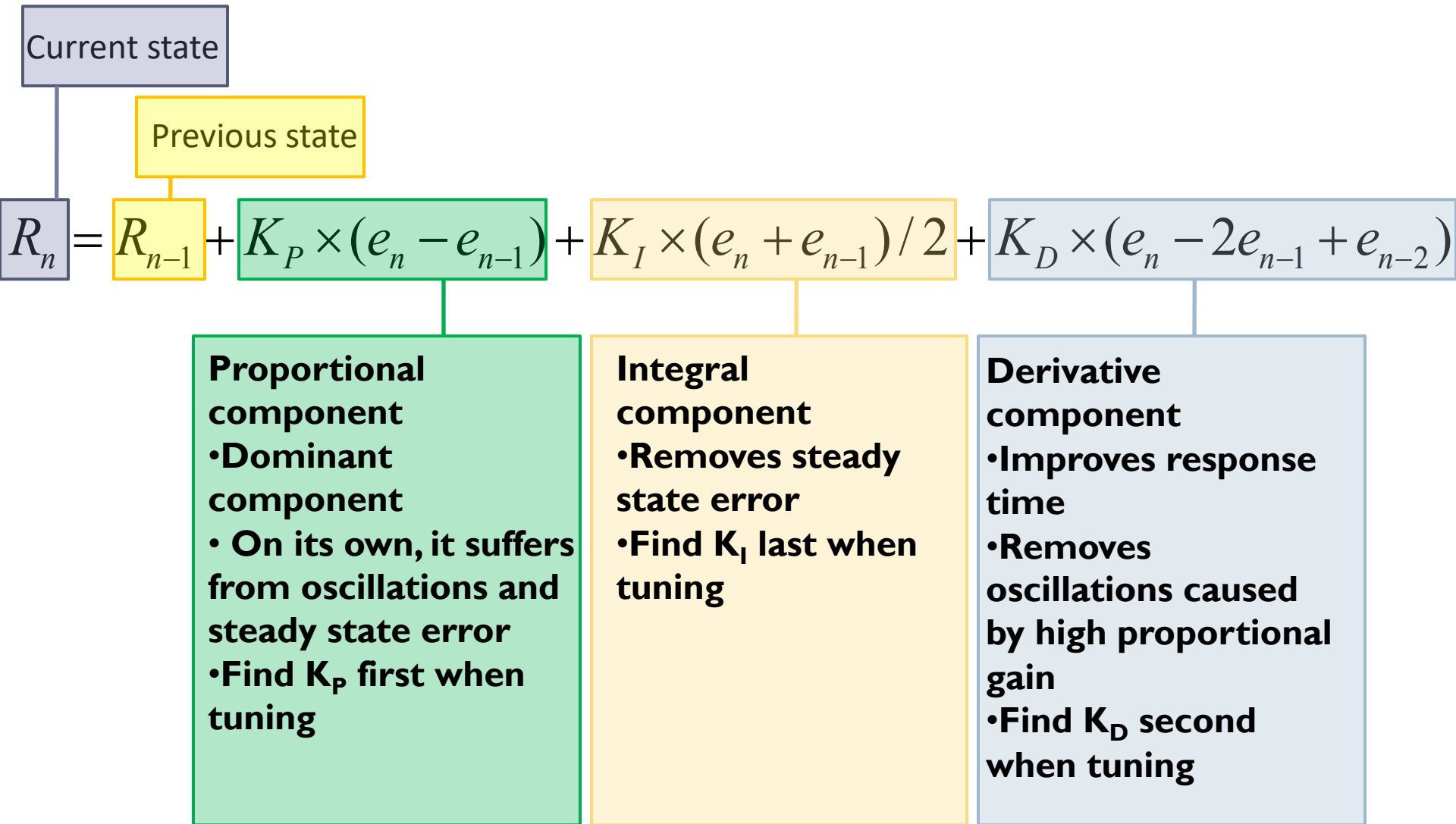▶ To do this we incorporate a Derivative term into our controller.

$$R_n = R_{n-1} + K_P \times (e_n - e_{n-1}) + K_I \times (e_n + e_{n-1})/2 + K_D \times (e_n - 2e_{n-1} + e_{n-2})$$

PID Response

Bräunl 2008

PID Controller

Current state

Previous state

$$R_n = R_{n-1} + K_P \times (e_n - e_{n-1}) + K_I \times (e_n + e_{n-1})/2 + K_D \times (e_n - 2e_{n-1} + e_{n-2})$$

Proportional component
- **Dominant component**
- **On its own, it suffers from oscillations and steady state error**
- **Find K_P first when tuning**

Integral component
- **Removes steady state error**
- **Find K_I last when tuning**

Derivative component
- **Improves response time**
- **Removes oscillations caused by high proportional gain**
- **Find K_D second when tuning**

PID Controller in C

$$R_n = R_{n-1} + K_P \times (e_n - e_{n-1}) + K_I \times (e_n + e_{n-1})/2 + K_D \times (e_n - 2e_{n-1} + e_{n-2})$$

```
int R_new, R_old, e_new, e_old;
float Kp;
…                                        /* initialise value etc */
void Pcont() {
   e_new = errorF();
   R_new = R_old + Kp x (e_new-e_old);      /* proportional component only */
   e_old = e_new;                   /* store new values as old for */
   R_old = R_new;                   /* next iteration */
   return;
}
```

PID Controller in C

$$R_n = R_{n-1} + K_P \times (e_n - e_{n-1}) + K_I \times (e_n + e_{n-1})/2 + K_D \times (e_n - 2e_{n-1} + e_{n-2})$$

```c
int R_new, R_old, e_new, e_old;
float Kp, Ki;                           /* added new control gain */
…
void PIcont() {
    e_new = errorF();
    R_new = R_old + Kp x (e_new-e_old) + Ki x (e_new + e_old)/2;
    e_old = e_new;
    R_old = R_new;
    return;
}
```

PID Controller in C

$$R_n = R_{n-1} + K_P \times (e_n - e_{n-1}) + K_I \times (e_n + e_{n-1})/2 + K_D \times (e_n - 2e_{n-1} + e_{n-2})$$

```
int R_new, R_old, e_new, e_old, e_old2;     /* added new error variable */
float Kp, Ki, Kd;                           /*added Derivative control gain */
…
void PIDcont() {
    e_new = errorF();
    R_new = R_old + Kp x (e_new-e_old) + Ki x (e_new + e_old)/2
            + Kd x (e_new – 2 x e_old + e_old2);
    e_old2 = e_old;          /* be sure to store e_old2 before changing e_old */
    e_old = e_new;
    R_old = R_new;
    return;
}
```

Tuning a PID Controller

‣ Proportional first.
  ‣ Start with Kd, Ki = 0
  ‣ set the desired velocity to reasonable setting
  ‣ start with a low value for Kp and increase until max reached or system response oscillates
  ‣ divide Kp by 2 if it oscillates

‣ Then Derivative.
  ‣ Increase Kd and observe behavior when changing desired speed by about 5%.
  ‣ Choose a value of Kd which gives a fast damped response.

‣ Finally Integral.
  ‣ Slowly increase Ki until oscillation starts.
  ‣ Then divide Ki by 2 or 3.

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely2013)

Programming in C - Variable Scope

- A variable's scope is the range from which it can be accessed.

- Trying to access a variable outside of its scope will usually result in a compiler/preprocesor error.

- Variable names within different scopes may be reused. This means you can have two variables with the same name that are not the same!

- Be careful with the scope of your variables!

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely2013)

Variable Scope

- A LOCAL VARIABLE has BLOCK SCOPE. It is accessible only within the {block} that it, itself is declared in and those nested within the same block.

- e.g.

```
...
int function() {
    int x = 0;      /* the variable x is declared inside function() */
    ...
}
...
/* The variable x will only be accessible inside the {} of the function */
int funtion2() {
    x = 1;
    ...
}
/* function2 does not have access to variable x, so will not compile */
```

- Local variables can also exist in nested blocks such as for, while and if statements.
- You should use local variables wherever appropriate, ie – don't give variables higher scope than they require.

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely2013)

Variable Scope

- GLOBAL VARIABLES have GLOBAL SCOPE (C++) or FILE SCOPE (C). They are accessible throughout a program by any function within it.

- e.g.

```
...
int x;          /* variable x is declared outside of the functions */
int function() {
     x = 0;
     ...
}
int funtion2() {
     x = 1;
     ...
}
/* This time both functions have access to x, so this will work */
```

- A global variable can save you having to pass variables back and forth between functions.
- A global variable may be superseded by a local variable of the same name. This will cause you to have two variables of the same name but the value will depend on the block from which it is called. This can be confusing so is best avoided.
- Try to only use global variables when it is beneficial.

Variable Scope

```c
int a = 1;
// here we have access to a
// here we have no access to b
// here we have no access to c
void function1() {
        int b = 2;
        // here we have access to a
        // here we have access to b
        // here we have no access to c
        int i;
        for(i = 0; i < 10; i++) {
                int c = 3;
                // here we have access to a
                // here we have access to b
                // here we have access to c
                // here we also have access to i
        }
        // here we also have access to i

        ...
}
void function2() {
        int a = 4;
        // here we do not have access to the original a as we have overridden the variable name for this block
        // here we have no access to b
        // here we have no access to c
        // here we have access to a new variable called a, which has no effect on the original a outside this block
        ...
}
```

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely2013)

Variable Scope

- The 'static' keyword is used to maintain a value of a variable, within a function block, over multiple executions even after that block has exited.

- A static variable should be initialised when declared.

- The variable will only be initialised the first time the function is called.

- Example;

```
void counter() {
        static int x = 0;                   // declare and initialise x
        x++;                                // increment x
        LCDprintf("%d\n", x);               // print x on the EyeBot
        return;
}
```

- This function will print a number, incrementing each time it is executed, and maintain the number even after it has been run.

Variable Scope

- There is more to learn about scope, particularly if you advance to C++ or other languages.

- You shouldn't need to know any more than this for this unit but if you are interested in learning you should check it out online.

Timing a Function in C

▸ It is important for your controller to run at a regular frequency for the control to work.

▸ In our library, we have a function OSAttachTimer that will allow us to run our function at a regular frequency.

▸ TIMER **OSAttachTimer**(int scale, void(*)(void) function);

Input: *(scale) prescale value for 1kHz Timer (1 to …)*

(function) function to be called periodically

Output: *(TIMER) handle to reference the timer struct.*

Semantics: Attach irq-routine (void function(void)) to the irq-list.
The scale parameter adjusts the call frequency (1000/scale Hz) of this routine to allow many different applications. ***Note: Execution time of any attached routine (and total time of all attached routines) has to be significantly < 10ms.*** *Otherwise timer interrupts will be missed and motor/sensor-timing gets corrupted.*

▸ Make sure your routine doesn't have an endless loop within it!

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely2013)

Timing a Function in C

```c
int main()
{
   TIMER t1;
   t1 = OSAttachTimer(20, &PIDcont);
   while (KEYRead() != KEY4)          /* check for end key */
   { /* set desired speed with input keys */ }
   OSDetachTimer(t1);
   return 0;
}


void PIDcont() {
   /* implement your controller here */
}
```

Switch statement

- The switch statement is a control structure that can take the place of many if else structures.

- It allows you to "switch" to one of several "cases" depending on a variable value.

```
...
switch (variable) {
  case 0:      x = function1();              /* if variable == 0 */
               break;
  case 1:      x = function2();              /* if variable == 1 */
               break;
  ... etc ...
  default:     x = functionD();              /* if variable is any
                                                other value*/
}
```

Switch statement example

```c
int main() {

    int key = KEYGet();               /* wait for user to press a key */

    switch key {

        case KEY1:drawMIC();

                break;                /* break skips out of the switch block */

        case KEY2:drawPOT1();         /* if you don't use break, the code in the */

                break;                /* following case will execute */

        case KEY3:drawPOT2();

                break;

        case KEY4:return 0;           /* this line is effectively redundant
                                         (covered by the default case)*/

        default:  return 0;           /* return will end the current function */

    }

    return 0;

}
```

Simple Tips for Labs

▸ Check out the RoBIOS function library.

▸ Check your compiler/preprocessor errors

  ▸ They will tell you where your error is and often a useful error message

  ▸ Parse errors are often just typographical errors

▸ Try to keep your structure clear

  ▸ Correct indenting, brackets etc

  ▸ Use a more advanced editor with code recognition (eg Notepad++/Sublime)

▸ Check your variable scope

  ▸ Make sure you don't accidentally redeclare a variable in a nested block scope

▸ Keep your program simple

  ▸ Don't over engineer it but do think about how to reduce your code

  ▸ If you are copying and pasting a piece of code over and over you should be using a function instead

▸ Wear shoes! If you are not wearing shoes you may be ejected from the lab.

More Tips for C Programming

- The following are a few tips based on common errors that have been occurring in the labs.

- End you program gracefully
    - firstly, make sure your program can end – this will save you time on having to reload the program every time you have to reset the EyeBot
    - make sure you release any hardware and timers you've used in your code (eg – motors, quad encoders, servos, cameras, timers etc.)

- Similarly, don't forget to initialise hardware

More Tips for C Programming

- ## Know your data
  - make sure you know the possible range of any data or measurement you are using – if you don't know it, measure it and display it on the screen
  - know the range of valid inputs for functions
  - simply bounding your data is not enough if you don't know the real range – you may need to scale first

- ## Make your code tell you what it's doing
  - use LCDPrintf **a lot** to debug your code – often problems are found just by displaying variables on the display while the code runs or printing messages to let you know when the code reaches certain points

Embedded Systems: Marcus Pham 2018 (Based on Stephen Whitely 2013)

More Tips for C Programming

‣ Check your spelling
  ‣ C is case sensitive, make sure that you spell variables and function names correctly

‣ Avoid implicit declaration
  ‣ a warning about implicit declaration occurs when you use a variable or call a function at a point before you've declared it