Physics 120B: Lecture 8

Odds and Ends

Binary/Hex/ASCII

Memory & Pointers in C

Decibels & dB Scales

Coherent Detection

Binary, Hexadecimal Numbers

- Computers store information in binary
  - 1 or 0, corresponding to V_{CC} and 0 volts, typically
  - the CC subscript originates from “collector” of transistor
- Become familiar with binary counting sequence

binary	decimal	hexadecimal
0000 0000	0	0x00
0000 0001	1	0x01
0000 0010	2	0x02
0000 0011	2+1 = 3	0x03
0000 0100	4	0x04
0000 0101	4+1 = 5	0x05
etc.		
1111 1100	128+64+32+16+8+4 = 252	0xfc
1111 1101	128+64+32+16+8+4+1 = 253	0xfd
1111 1110	128+64+32+16+8+4+2 = 254	0xfe
1111 1111	128+64+32+16+8+4+2+1 = 255	0xff

Binary to Hex: easy!

- Note separation of previous 8-bit (one-byte) numbers into two 4-bit pieces (nibbles)
  - makes expression in hex (base-16; 4-bits) natural

binary	hexadecimal	decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A (lower case fine)	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

ASCII Table in Hex

first hex digit

second hex digit

	0	1	2	3	4	5	6	7
0	NUL ^@ null (\0)	DLE ^P	SP space	0	@	P	`	p
1	SOH ^A start of hdr	DC1 ^Q	!	1	A	Q	a	q
2	STX ^B start text	DC2 ^R	“	2	B	R	b	r
3	ETX ^C end text	DC3 ^S	#	3	C	S	c	s
4	EOT ^D end trans	DC4 ^T	$	4	D	T	d	t
5	ENQ ^E	NAK ^U	%	5	E	U	e	u
6	ACK ^F acknowledge	SYN ^V	&	6	F	V	f	v
7	BEL ^G bell	ETB ^W	‘	7	G	W	g	w
8	BS ^H backspace	CAN ^X	(	8	H	X	h	x
9	HT ^I horiz. tab (\t)	EM ^Y	)	9	I	Y	i	y
A	LF ^J linefeed (\r)	SUB ^Z	*	:	J	Z	j	z
B	VT ^K vertical tab	ESC escape	+	;	K	[	k	{
C	FF ^L form feed	FS	,	<	L	\	l	|
D	CR ^M carriage ret (\n)	GS	–	=	M	]	m	}
E	SO ^N	RS	.	>	N	^	n	~
F	SI ^O	US	/	?	O	_	o	DEL

ASCII in Hex

- Note the patterns and conveniences in the ASCII table
  - 0 thru 9 is hex 0x30 to 0x39 (just add 0x30)
  - A-Z parallels a-z; just add 0x20
    - starts at 0x41 and 0x61, so H is 8th letter, is 0x48, etc.
  - the first 32 characters are control characters, often represented as Ctrl-C, denoted ^C, for instance
    - associated control characters mirror 0x40 to 0x5F
    - put common control characters in red; useful to know in some primitive environments

Two's Complement

- Unsigned are direct binary representation
- Signed integers usually follow "two's complement"

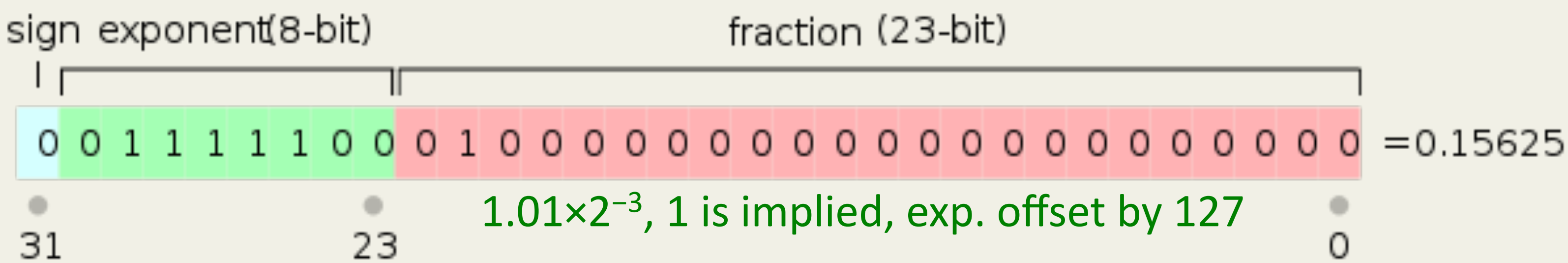
binary	hex	unsigned	2's complement
0000 0000	0x00	0	0
0000 0001	0x01	1	1
0000 0010	0x02	2	2
0111 1111	0x7F	127	127
1000 0000	0x80	128	-128
1000 0001	0x81	129	-127
1111 1110	0xFE	254	-2
1111 1111	0xFF	255	-1

  – rule: to get neg. number, flip all bits and add one
    - example: -2: 0000 0010 → 1111 1101 + 1 = 1111 1110

  – adding pos. & neg. → 0000 0000 (ignore overflow bit)

Floating Point Numbers

- Most standard is IEEE format
  - http://en.wikipedia.org/wiki/IEEE_754-1985



- Three parts: sign, exponent, mantissa
  - single-precision (float) has 32 bits (1, 8, 23, resp.)
    - 7 digits; $10^{\pm 38}$; log(10)/log(2) = 3.32, so $2^{23} \approx 10^7$; ±127/3.32 ≈ 38
  - double precision (double) has 64 bits (1, 11, 52, resp.)
    - 16 digits; $10^{\pm 308}$
- The actual convention is not critical for us to understand, as much as:
  - limitations to finite representation
  - space allocation in memory: just 32 or 64 bits of 1's & 0's

Arrays & Storage in C

- We can hold more than just one value in a variable
  - but the program needs to know how many places to save in memory
- Examples:

```
int i[8], j[8]={0}, k[]={9,8,6,5,4,3,2,1,0};
double x[10], y[10000]={0.0}, z[2]={1.0,3.0};
char name[20], state[]="California";
```

  - we can either say how many elements to allow and leave them unset; say how many elements and initialize all elements to zero; leave out the number of elements and specify explicitly; specify number of elements and contents
  - character arrays are strings
  - strings must end in `'\0'` to signal the end
  - must allow room: `char name[4]="Bob"`
    - fourth element is `'\0'` by default

Indexing Arrays

```
int i,j[8]={0},k[]={2,4,6,8,1,3,5,7};
double x[8]={0.0},y[2]={1.0,3.0},z[8];
char name[20],state[]="California";

for (i=0; i<8; i++)
{
  z[i] = 0.0;
  printf("j[%d] = %d, k[%d] = %d\n",i,j[i],i,k[i]);
}
name[0]='T';
name[1]='o';
name[2]='m';
name[3] = '\0';
printf("%s starts with %c and lives in %s\n",name,name[0],state);
```

- Index array integers, starting with zero
- Sometimes initialize in loop (`z[ ]` above)
- String assignment awkward outside of declaration line
  - `#include <string.h>` provides "useful" string routines
    - done automatically in Arduino, but also String type makes many things easier

Memory Allocation in Arrays

- `state[]="California";` →

each block is 8-bit `char`

C	a	l	i	f	o	r	n	i	a	\0

- `name[11]="Bob";` →

B	o	b	\0							

  - empty spaces at the end could contain any random garbage

- `int i[] = {9,8,7,6,5,4,3,2};` →

9	8	7	6	5	4	3	2

each block is 16 or 32-bit `int`

  - indexing `i[8]` is out of bounds, and will either cause a segmentation fault (if writing), or return garbage (if reading)

Multi-Dimensional Arrays

```
int i,j,arr[2][4];

for (i=0; i<2; i++){
  for (j=0; j<4; j++){
    arr[i][j] = 4+j-2*i;
  }
}
```

i \ j	0	1	2	3
0	4	5	6	7
1	2	3	4	5

in memory space:

4	5	6	7	2	3	4	5

- C is a row-major language: the first index describes which row (not column), and arranged in memory row-by-row
  – memory is, after all, strictly one-dimensional
- Have the option of treating a 2-D array as 1-D
  – `arr[5] = arr[1][1] = 3`
- Can have arrays of 2, 3, 4, … dimensions

Arrays and functions

- How to pass arrays into and out of functions?
- An array in C is actually handled as a "pointer"
  - a pointer is a direction to a place in memory
- A pointer to a variable's address is given by the & symbol
  - you may remember this from `scanf` functions
- For an array, the name is *already* an address
  - because it's a block of memory, the name by itself doesn't contain a unique value
  - instead, the name returns the address of the first element
  - if we have `int arr[i][j];`
    - `arr` and `&arr[0]` and `&arr[0][0]` mean the same thing: the address of the first element
- By passing an address to a function, it can manipulate the contents of memory directly, without having to pass bulky objects back and forth explicitly

Example: 3x3 matrix multiplication

```
void mm3x3(double a[], double b[], double c[])

// Takes two 3x3 matrix pointers, a, b, stored in 1-d arrays nine
// elements long (row major, such that elements 0,1,2 go across a
// row, and 0,3,6 go down a column), and multiplies a*b = c.
{

  double *cptr;    // pointer type variable: * gets at contents
  int i,j;

  cptr = c;        // without *, it's address; point to addr. for c

  for (i=0; i<3; i++){
    for (j=0; j<3; j++){
      *cptr++ = a[3*i]*b[j] + a[3*i+1]*b[j+3] + a[3*i+2]*b[j+6];
      // calc value to stick in current cptr location, then
      // increment the value for cptr to point to next element
    }
  }
}
```

mm3x3, expanded

- The function is basically doing the following:

```
*cptr++ = a[0]*b[0] + a[1]*b[3] + a[2]*b[6];
*cptr++ = a[0]*b[1] + a[1]*b[4] + a[2]*b[7];
*cptr++ = a[0]*b[2] + a[1]*b[5] + a[2]*b[8];

*cptr++ = a[3]*b[0] + a[4]*b[3] + a[5]*b[6];
*cptr++ = a[3]*b[1] + a[4]*b[4] + a[5]*b[7];
*cptr++ = a[3]*b[2] + a[4]*b[5] + a[5]*b[8];

*cptr++ = a[6]*b[0] + a[7]*b[3] + a[8]*b[6];
*cptr++ = a[6]*b[1] + a[7]*b[4] + a[8]*b[7];
*cptr++ = a[6]*b[2] + a[7]*b[5] + a[8]*b[8];
```

  – which you could confirm is the proper set of operations for multiplying out 3×3 matrices

Notes on mm3x3

- The function is constructed to deal with 1-d instead of 2-d arrays
  - 9 elements instead of 3×3
  - it could have been done either way
- There is a pointer, `*cptr` being used
  - by specifying `cptr` as a double pointer, and assigning its address (just `cptr`) to `c`, we can stock the memory by using "pointer math"
  - `cptr` is the address; `*cptr` is the value at that address
  - just like `&x_val` is an address, while `x_val` contains the value
  - `cptr++` bumps the address *by the amount appropriate to that particular data type*, called "pointer math"
  - `*cptr++ = value;` assigns value to `*cptr`, then advances the `cptr` count

Using mm3x3

```
#include <stdio.h>

void mm3x3(double a[], double b[], double c[]);

int main()
{
  double a[]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
  double b[]={1.0, 2.0, 3.0, 4.0, 5.0, 4.0, 3.0, 2.0, 1.0};
  double c[9];

  mm3x3(a,b,c);

  printf("c = %f  %f  %f\n",c[0],c[1],c[2]);
  printf("    %f  %f  %f\n",c[3],c[4],c[5]);
  printf("    %f  %f  %f\n",c[6],c[7],c[8]);

  return 0;
}
```

- passing just the names (addresses) of the arrays
  - filling out `a` and `b`, but just making space for `c`
  - note function declaration before main

Another way to skin the cat

```
double a[3][3]={{1.0, 2.0, 3.0},
                {4.0, 5.0, 6.0},
                {7.0, 8.0, 9.0}};
double b[3][3]={{1.0, 2.0, 3.0},
                {4.0, 5.0, 4.0},
                {3.0, 2.0, 1.0}};
double c[3][3];

mm3x3(a,b,c);
```

- Here, we define the arrays as 2-d, knowing that in memory they will still be 1-d
  - we will get compiler warnings, but the thing will still *work*
  - not a recommended approach, just presented here for educational purposes
  - Note that we could replace `a` with `&a[0][0]` in the function call, and the same for the others, and get no compiler errors

Decibels

- Sound is measured in decibels, or dB
  - as are many radio-frequency (RF) applications
- Logarithmic scale
  - common feature is that every 10 dB is a factor of 10 in power/intensity
  - other handy metrics
    - 3 dB is 2×
    - 7 dB is 5×
    - obviously piling 2× and 5× is 10×, which is 10 dB = 3 dB + 7 dB
  - decibels thus combine like logarithms: addition represents multiplicative factors

Sound Intensity

- Sound requires energy (pushing atoms/molecules through a distance), and therefore a power
- Sound is characterized in decibels (dB), according to:
  - sound level = $10 \times \log(I/I_0)$ = $20 \times \log(P/P_0)$ dB
  - $I_0 = 10^{-12}$ W/m^2 is the threshold power intensity (0 dB)
  - $P_0 = 2 \times 10^{-5}$ N/m^2 is the threshold pressure (0 dB)
    - atmospheric pressure is about 10^5 N/m^2
    - 20 out front accounts for intensity going like P^2
- Examples:
  - 60 dB (conversation) means $\log(I/I_0) = 6$, so $I = 10^{-6}$ W/m^2
    - and $\log(P/P_0) = 3$, so $P = 2 \times 10^{-2}$ N/m^2 = 0.0000002 atmosphere!!
  - 120 dB (pain threshold) means $\log(I/I_0) = 12$, so $I = 1$ W/m^2
    - and $\log(P/P_0) = 6$, so $P = 20$ N/m^2 = 0.0002 atmosphere
  - 10 dB (barely detectable) means $\log(I/I_0) = 1$, so $I = 10^{-11}$ W/m^2
    - and $\log(P/P_0) = 0.5$, so $P \approx 6 \times 10^{-5}$ N/m^2

Sound hitting your eardrum

- Pressure variations displace membrane (eardrum, microphone) which can be used to measure sound
  - my speaking voice is moving your eardrum by a mere 1.5×10^{-4} mm = 150 nm = 1/4 wavelength of visible light!
  - threshold of hearing detects 5×10^{-8} mm motion, one-half the diameter of a single atom!!!
  - pain threshold corresponds to 0.05 mm displacement
- Ear ignores changes slower than 20 Hz
  - so though pressure changes even as you climb stairs, it is too slow to perceive as sound
- Eardrum can't be wiggled faster than about 20 kHz
  - just like trying to wiggle resonant system too fast produces no significant motion

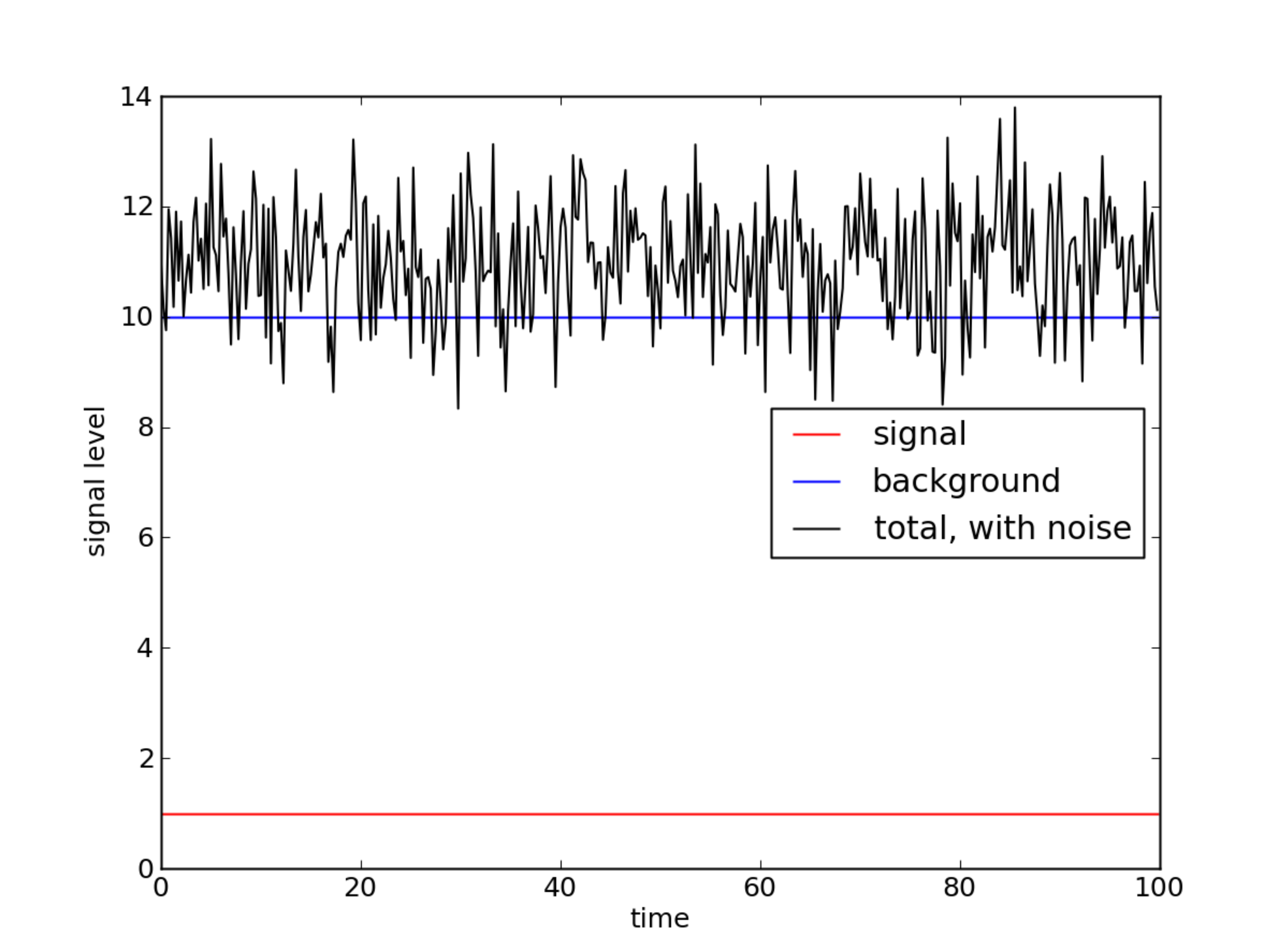
dB Scales

- In the radio-frequency (RF) world, dB is used several ways
  - dB is a relative scale: a ratio: often characterizing a gain or loss
    - +3 dB means a factor of two more
    - −17 dB means a factor of 50 loss, or 2% throughput
  - dBm is an absolute scale, in milliwatts: $10 \times \log(P/1\ \mathrm{mW})$
    - a 23 dBm signal is 200 mW
    - 36 dBm is 4 W (note 6 dB is two 3 dB, each a factor of 2 → 4×)
    - −27 dBm is 2 μW
  - dBc is signal strength relative to the carrier
    - often characterizes distortion from sinusoid
    - −85 dBc means any distortions are almost nine orders-of-magnitude weaker than the main sinusoidal “carrier”

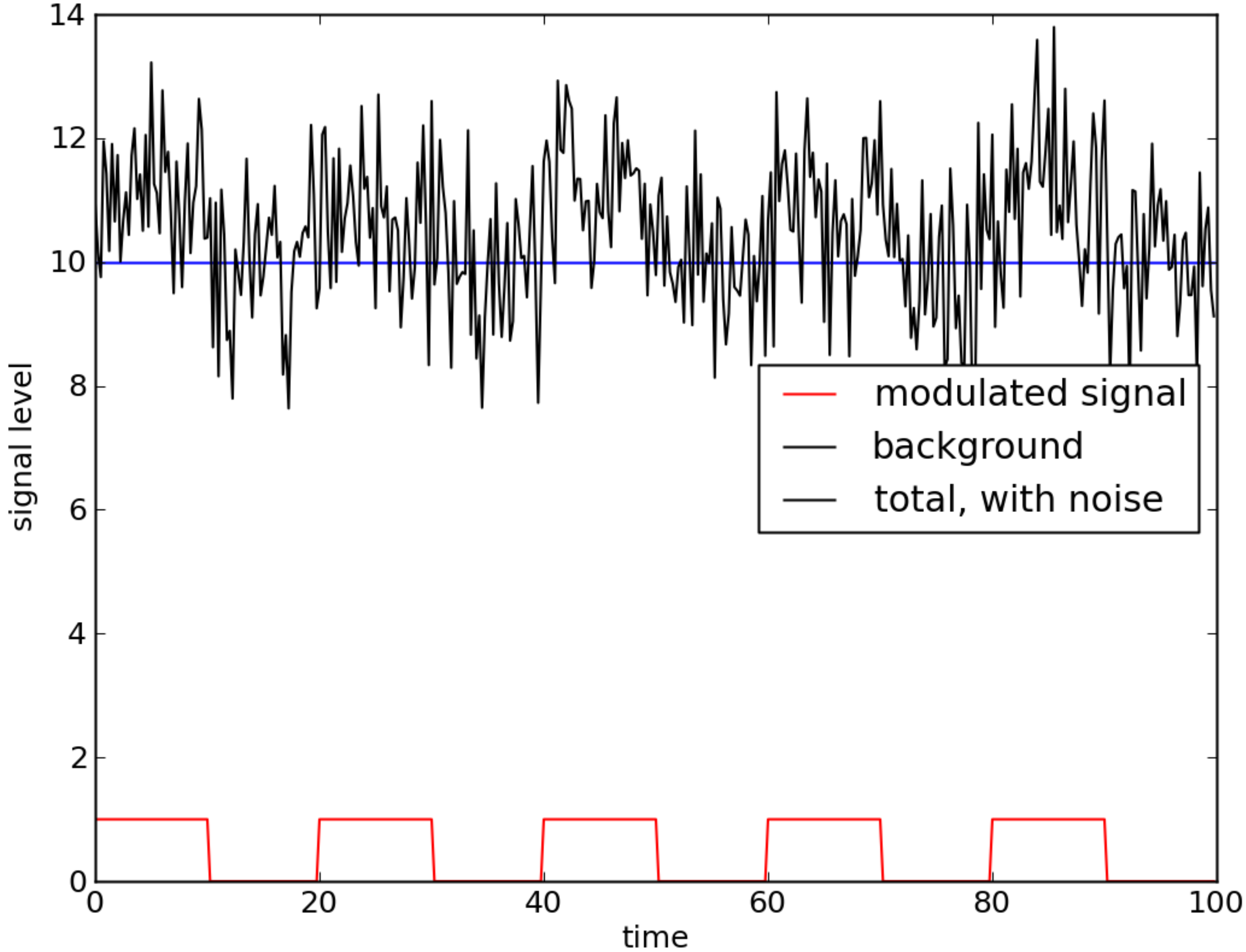
Coherent Detection

- Sometimes fighting to discern signal against background noise
  - photogate in bright setting, for instance
- One approach is *coherent detection*
  - modulate signal at known phase, in ON/OFF pattern at 50% duty cycle
  - accumulate (add) in-phase parts, while subtracting out-of-phase parts
  - have integrator perform accumulation, or try in software
    - but if background is noisy in addition to high, integration better
  - basically background subtraction
  - gain more the greater the number of cycles integrated

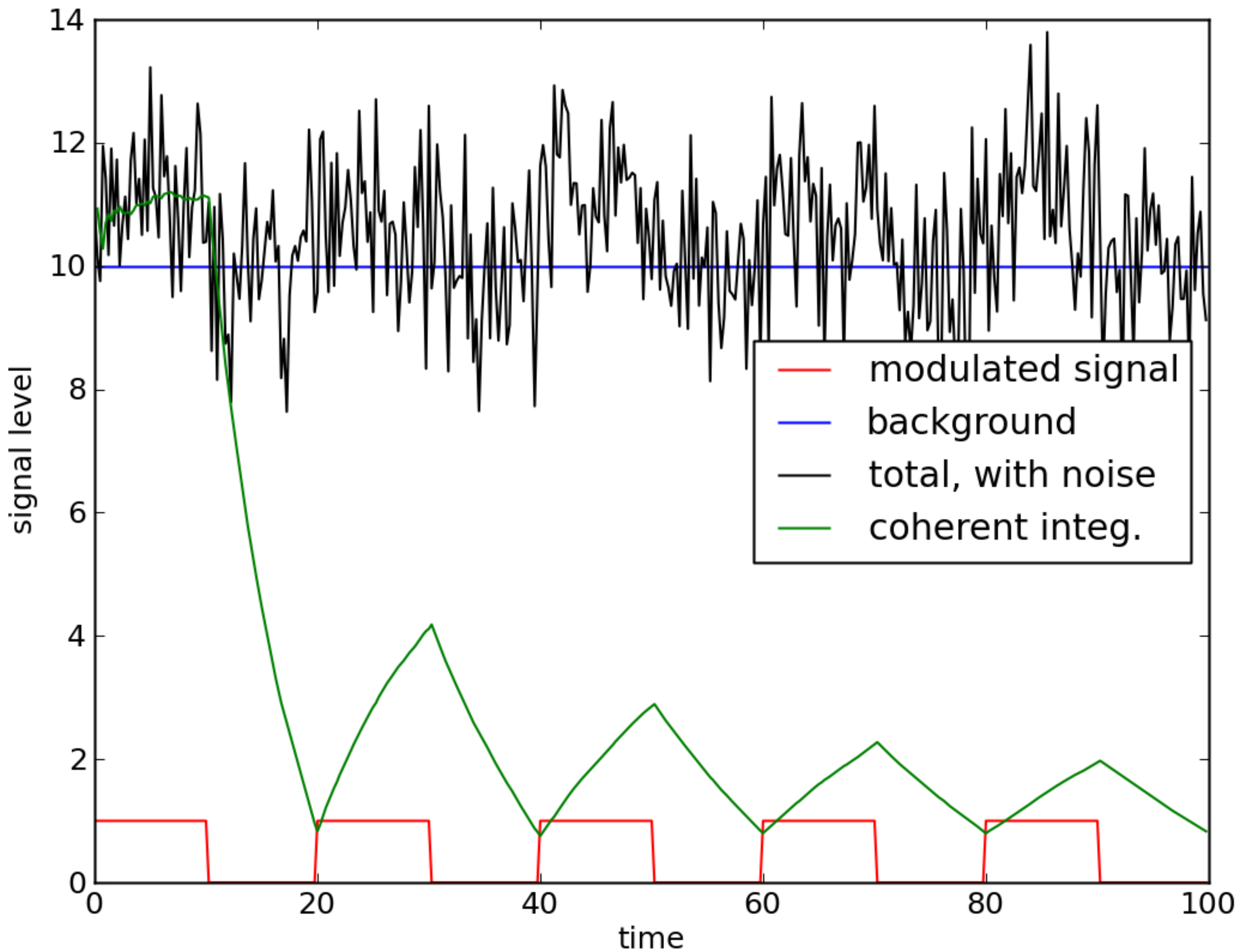
Raw Signal, Background, and Noise



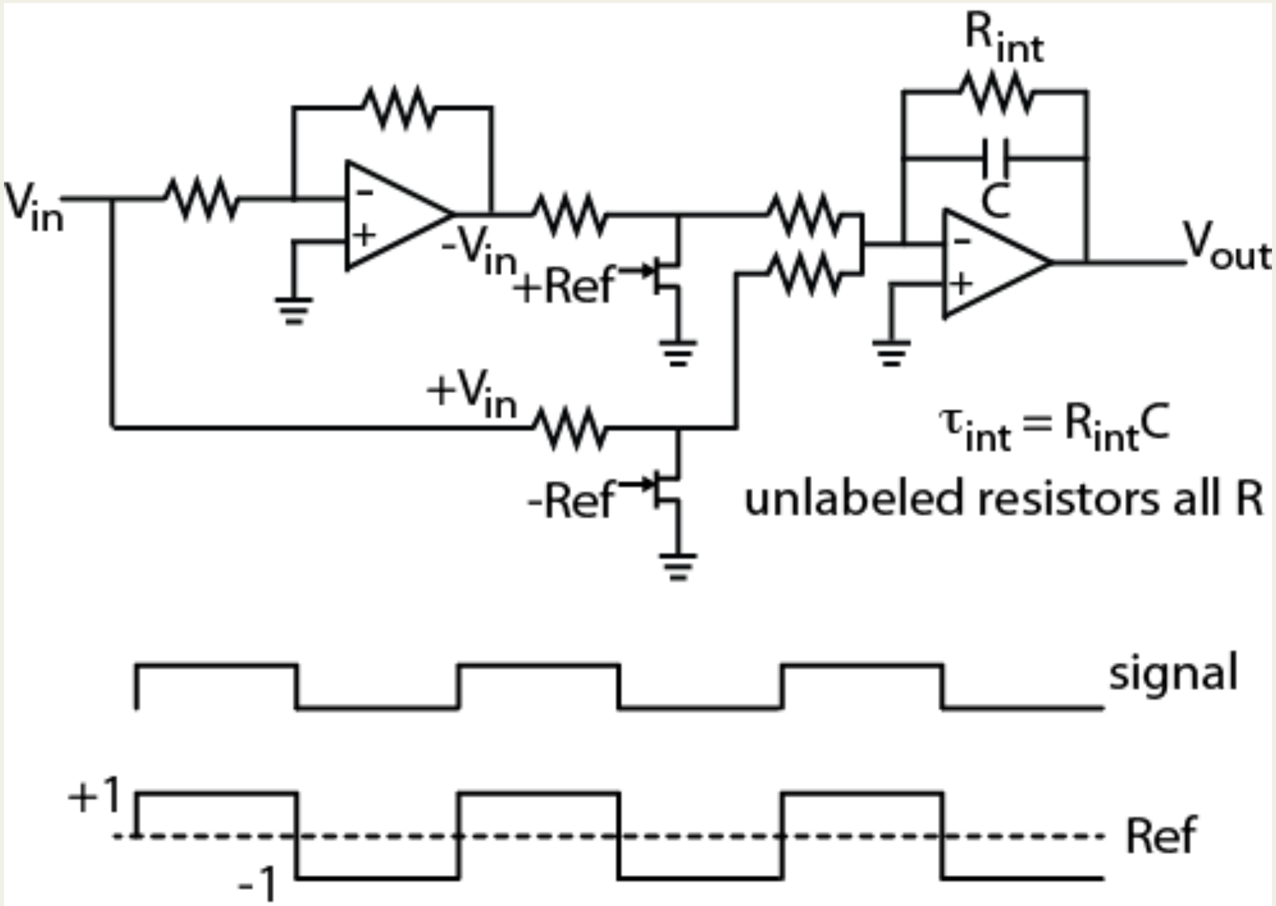
Modulated Signal; still hard to discern



Integration, subtracting “OFF” portions



Expressed in Electronics



first op-amp just inverting; second sums two inputs, only one on at a time
has effect of adding parts when Ref = +1, subtracting where Ref = -1
clears "memory" on timescale of $\tau_{int} = R_{int}C$
could also conceive of performing math in software

Announcements

- Project Proposals due next Friday, Feb 8
- Lab 4 due following Tu/Wed (2/12, 2/13)
- No lecture this Friday (2/1)
- Resume Lecture Monday (2/4)