Physics PHZ-5156-C, 28 August 2006

PACKET I

This packet, available from Pro Copy or as Postscript files under the directory `/home/5156/packet` on `physics`, contains detailed instructions and background reading supporting the projects planned for the course. Additional information (including project due dates and assignments added or deleted) will be handed out in class.

For most weeks, you will copy some files into your own directory; these will include a `Makefile` and suggested templates for your assignments. At first, the templates will really be mostly-completed programs, where you have only to fill in some details. This is to help those students who may not have a strong prior background in programming. The `Makefile` contains instructions, executed when you type `make`, for compiling and testing your programs.

When you have finished an assignment, typing `make submit` in the assignment directory will "submit" your completed assignment. It does not actually copy anything; rather, it notifies the grader that you have finished and that he may mark your work. When he is ready, the grader will log in as *you* to check your work. Not that anyone was thinking about it, but in case a student booby-trapped her `Makefile` or other files, she would just hurt herself. Do not change your files after typing `make submit`; the command sends the grader a checksum, which would change with any changes you make. Remember to include any of your files necessary for your project in the project directory; in other words, avoid references of the form "`../mydir/myfile`", since they won't work if the grader copies your directory to a temporary location.

This packet includes the following. Scheduling and topics are tentative.

week	title	
		This cover sheet
1.	1.	Histograms and an Introduction to C
		C Style Guide
		Notes for Fortran-90 Users
2.	2.1.	Probability Distributions and Kernel Smoothing
*	2.2.	Digital Filters in the Time Domain
3.	3.	Working in the Frequency Domain
4–5.	4.	The Spike-Sorting Problem in Neurobiology
6–8.	5.	Integrating Regular and Chaotic Classical Orbits *thanks to Don Fredkin*
*	6.	The Pricing of American Put and Call Options
*	7.	Experimental Control and Data Acquisition with Labview
9–11.	8.	Quantum Magnetism
		Supplementary notes on changing bases
12–15.	9.	Classical-Molecular-Dynamics Simulation of Crack Propagation

**not planned for this year*

Please understand if some of the edges are a little rough. We may find errors; run the `errata` program for the latest updates. The schedule also is tentative: if we find it's too fast, we'll drop a topic and slow down. While I think it less likely, if the class finds the pace too slow, we can add some of the optional topics.

Two additional packets contain the CVODE reference manual (useful for the classical-mechanics project) and references on Unix, the `vi` editor, `axis`, `gdb`, `make`, and `awk`.

Policy on cooperation and collaboration. Students are encouraged to discuss their problems, solutions and methods with one another and to assist one another in finding bugs in programs and otherwise taming the computer. Except for the group project (chapter 9), such cooperation should not include exchange of source code or written solutions, either by electronic mail or in printed form. Each student is expected to write his or her own program; submission of joint work is not acceptable (again, except for chapter 9). **Copying of code or code fragments from sources outside the class or from previous years' projects is plaigiarism and may be cause for expulsion from the university.**

Physics PHZ-5156-C, 28 August 2006

Week 1: Histograms and an Introduction to C

Copy the question files and suggested program templates from `/home/5156/assignments/week1`*, for example using the following commands:*

```
cd                                          start at your home directory
cp -r /home/5156/assignments/week1 .        copy the directory to all levels
cd week1                                    begin working in this directory
```

When you have completed everything, the `make` *command should compile and demonstrate your programs. Feel free to modify the* `Makefile` *if you wish. The grading for this week is S/U, with "S" allowed for any substantial effort; we expect that students with more programming experience will help those with less. Type* `make submit` *when you are ready to submit your work.*

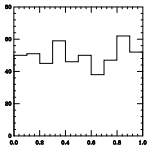
Often one measures something that should have a single value but finds a distribution of values. Standard courses in statistical techniques tell us how to estimate the true underlying value (often the mean is best justified) and variance. This week, however, we'd just like to draw a histogram as an approximation to the shape of the distribution of fluctuations. If we see two separated peaks in the histogram, we might begin to wonder whether in fact the thing we are measuring can take *two* different true values.

The next page of notes gives you your assignment. To help any students with less experience in the C programming language, I've provided partly-completed template files. I shall be available several evenings this week in room 102 to work individually or in a group with students requiring a review.

The rest of the notes are essentially props to help with the C language. These include some comparisons of C to other languages. The comparison of quick-sort routines demonstrates how it is possible to write very concise, easy-to-read code. The following example shows how the opposite is also possible. Some people count the mere possibility of writing obfuscated code against C, but then the same criticism could apply to English or French. Just as everyone writes text differently, so everyone writes C differently. You will be exposed to at least three programming styles. Of these, that in *Numerical Recipes* is probably the worst: the authors are FORTRAN programmers who haven't mastered the C idiom. In all cases, your goal should be code that you or someone else will be able to pick up in five years and still understand completely. Short subroutines and copious comments are the most important keys to achieving this. In this course, we are not going to be concerned with how fast your program runs. Once you start writing programs for research, you will find that typical programs spend nearly all their time in a few very well isolated pieces of code; it therefore makes better sense to worry about optimizing these sections only after everything works than to take ten times as long to write the program by squeezing every possible optimization out of the irrelevant parts. We distinguish, however, algorithms from code optimization: often you will achieve an enormous savings in time (both execution and programming) by picking the right algorithm from the start.

Continuing in the packet, we offer some elementary exercises in preparation for the week's assignment, information on how to compile and debug programs, pictures modeling how C treats pointers and argument-passing internally, a binary-tree sorting program (to demonstrate the use of `structs`), and a list of common programming mistakes. It will help if you bring your copy of these notes to the laboratory so we can refer to them in explaining some point or another.

Week-1 Project



We shall write three programs useful for the manipulation of data. Even before you have written them yourself, you will be able to test working versions. These commands extract columns of data from a file, collect the data into histogram bins, and then convert the histogram bins for graphing by the `axis` program. Using the working versions, first try them out. You will already have copied the `data` file into your directory; it contains three columns of manufactured data. The following command will plot a histogram of the second column:

```
extract 2 < data | histbin | hist2axis | axis | xplot
```

The data in column 2, all of which here happen to lie between 0 and 1, have been grouped into ten bins, with the number in each bin plotted vertically: so the first bin, for data greater than or equal to 0 but less than 0.1, contains 50 data, the second bin, between 0.1 and 0.2, 51 counts, the third 45, *etc.*

Histograms find use wherever measurements include noise or a fundamentally stochastic element. For example, in Chapter 4, we'll use the histogram program written here to analyze data from a cat's brain stem; one could apply the same program to a distribution of scattering angles in particle physics or to the spacings between energy levels of electrons in a ring.

`extract` While in the example above, we wished to extract only a single column, in general, the `extract` program should know how to extract an arbitrary number of columns. For example, to plot column two against column three, we could give the command

```
extract 2 3 < data | sort -n | axis | xplot
```

You will already have copied an outline `extract.c` that includes comments and templates for writing the various functions needed as well as a complete `util.c` containing some miscellaneous utilities needed for the project and a `Makefile`. Once you have completed `extract.c`, the command `make extract` should compile and link the program.

The top of `extract.c` contains a detailed description of what the program should do. Once the program has compiled, you should test it with your own data set or with the `data` file provided, making sure it does what you expect. (The template also suggests an optional feature, the processing of comment lines embedded in the data. I like to use such lines to track when and how data were created and later processed.)

`histbin` This program reads in a single column of numbers, determines the range of the data (the extremal numbers all the data lie between), puts them into bins, and prints out two columns. The first column is the midpoint of a bin, the second the number of counts in that bin. If an argument is given to `histbin` on the command line, it specifies how many equal-width bins `histbin` should create. Otherwise, the program makes ten. I have provided in `histread.c` a `histread()` function to read all the lines of the file into an array; internally, it uses a singly-linked list before converting the list to an array for your use. Again, the top of `histbin.c` describes in greater detail what the program should do.

`hist2axis`

The public-domain `axis` program prints a graph of `x` and `y` values in the first and second columns; in its simplest form, `axis` merely connects the dots with lines and draws a graduated frame. The output of `histbin` above can be piped to axis,

```
extract 2 < data | histbin | axis | xplot
```

but the result doesn't look very good. We prefer to represent the number of counts in each histogram bin as a box of some height rather than as a point in an `x-y` plot. The `hist2axis` program accomplishes this by writing the `x` and `y` values of the box's corners on successive lines, which `axis` then connects with lines.

Comparison of C and FORTRAN
Quick-sort Algorithm

```c
/* qsort.c
 * 7/96  DAR
 * adapted from Kernighan and Ritchie, 2nd ed.
 */

#define MINFORQ 7       /*if fewer than MINFORQ elements, use insertion sort*/

/* swap elements i and j of the array v (must work even if i==j) */
void swap(int *v, int i, int j)
{
        int temp = v[i];
        v[i] = v[j];
        v[j] = temp;
}

/* Insertion sort: good when there aren't too many to sort (n=length).
 * This works by picking the second item from the left and ordering it with
 * respect to just the first two, then picking the third, ordering it with
 * respect to the first three, etc.  Adapted from Numerical Recipes.
 */
void insertionsort(int *vec, int n)
{
        int *vleft, *vright=vec+1, nright=n, nleft=1, ninner, a, b;

        while(--nright > 0)     {
                a = *vright;
                for(ninner=nleft++,vleft=vright++ -1;  ninner--;  vleft--) {
                        if( (b= *vleft) < a) break;     /* found place */
                        vleft[1] = b;                   /* shuffle */
                }
                vleft[1] = a;
        }
}

/* Pick an element from somewhere in the vector (between left and right).
 * At the end of this routine, the element will be moved so that everything
 * to its left is smaller, everything to its right larger.
 */
void qsort(int *vec, int left, int right)
{
        int i, last;

        if((i=right-left+1) < MINFORQ) {        /* more efficient */
                insertionsort(vec+left, i);     /* to call this algor.*/
                return;
        }
        swap(vec, left, (left+right)/2);        /*move partition elem*/
        last = left;                            /* to vec[left] */
        for(i=left+1; i<=right; i++)            /* partition */
                if(vec[i] < vec[left])
                        swap(vec, ++last, i);
        swap(vec, left, last);
        qsort(vec, left, last-1);
        qsort(vec, last+1, right);
}

/* Test it. */
int main()
{
        static int v[]={61,23,7,77,59,50,13,96,12,39,80,67,15,84,60,61,70,65,1};
        int *p=v, n=sizeof(v)/sizeof(v[0]);
        qsort(v, 0, n-1);
        while(n--)
                printf("%d ", *p++);
        puts("");
        return 0;
}
```

Why implicit typing is bad:
`DO 1 I=1.100`

```fortran
C FQSORT.F
C 7/96 DAR
C ADAPTED FROM NUMERICAL RECIPES
C THE ONLY CHANGES I'VE MADE ARE TO ELIMINATE IMPLICIT TYPING, TO NAME
C A FEW OF THE VARIABLES MORE MNEMONICALLY, TO SHORTEN THE PROGRAM BY
C REMOVING THE COMPLICATION THAT MAKES SURE AN ALREADY-SORTED ARRAY ISN'T
C THE WORST CASE, TO CHANGE THE ORDER OF PARAMETERS, AND TO ADD A MAIN
C ROUTINE FOR TESTING

C TEST QSORT
      PROGRAM MAIN
      IMPLICIT NONE
      INTEGER ARRAY, I
      DIMENSION ARRAY(15)
      DATA ARRAY/19,56,11,68,56,23,23,56,66,45,48,52,65,40,77/
      CALL QSORT(ARRAY,15)
      WRITE(6,100) (ARRAY(I), I=1,15)
100   FORMAT(1H , 15I)
      END

C SORT AN ARRAY OF LENGTH N INTO ASCENDING NUMERICAL ORDER
      SUBROUTINE QSORT(ARR,N)
      IMPLICIT NONE
C USE INSERTION SORT ON ANY SUB-SECTION SMALLER THAN QSORTMAX
C THIS WORKS BY PICKING THE SECOND ITEM FROM THE LEFT AND ORDERING IT WITH
C RESPECT TO JUST THE FIRST TWO, THEN PICKING THE THIRD, ORDERING IT WITH
C RESPECT TO THE FIRST THREE, ETC.  ADAPTED FROM NUMERICAL RECIPES.
      INTEGER QSORTMAX, NSTACK
      PARAMETER(QSORTMAX=7, NSTACK=50)
      INTEGER A, ARR, N, ISTACK, JSTACK, LEFT, RIGHT, J, I, IQ
      DIMENSION ARR(N), ISTACK(NSTACK)

      JSTACK=0
      LEFT=1
      RIGHT=N
10    IF(RIGHT-LEFT.LT.QSORTMAX) THEN
        DO 13 J=LEFT+1,RIGHT
          A=ARR(J)
          DO 11 I=J-1,1,-1
            IF(ARR(I).LE.A) GOTO 12
            ARR(I+1)=ARR(I)
11        CONTINUE
          I=0
12        ARR(I+1)=A
13      CONTINUE
        IF(JSTACK.EQ.0) RETURN
        RIGHT=ISTACK(JSTACK)
        LEFT=ISTACK(JSTACK-1)
        JSTACK=JSTACK-2
      ELSE
C QUICK SORT: PICK AN ELEMENT FROM SOMEWHERE IN THE VECTOR (BETWEEN LEFT AND
C RIGHT).  AT THE END OF THIS SECTION, THE ELEMENT WILL BE MOVED SO THAT
C EVERYTHING TO ITS LEFT IS SMALLER, EVERYTHING TO ITS RIGHT LARGER.
        I=LEFT
        J=RIGHT
        IQ=(RIGHT+LEFT)/2
        A=ARR(IQ)
        ARR(IQ)=ARR(LEFT)
20      CONTINUE
21        IF(J.GT.0)THEN
            IF(A.LT.ARR(J))THEN
              J=J-1
              GOTO 21
            ENDIF
          ENDIF
          IF(J.LE.I)THEN
            ARR(I)=A
            GOTO 30
          ENDIF
          ARR(I)=ARR(J)
          I=I+1
22        IF(I.LE.N)THEN
            IF(A.GT.ARR(I))THEN
              I=I+1
              GOTO 22
            ENDIF
          ENDIF
          IF(J.LE.I)THEN
            ARR(J)=A
            I=J
            GOTO 30
          ENDIF
          ARR(J)=ARR(I)
          J=J-1
        GOTO 20
30      JSTACK=JSTACK+2
        IF(JSTACK.GT.NSTACK)PAUSE 'FORTRAN CANNOT HANDLE THIS'
        IF(RIGHT-I.GE.I-LEFT)THEN
          ISTACK(JSTACK)=RIGHT
          ISTACK(JSTACK-1)=I+1
          RIGHT=I-1
        ELSE
          ISTACK(JSTACK)=I-1
          ISTACK(JSTACK-1)=LEFT
          LEFT=I+1
        ENDIF
      ENDIF
      GOTO 10
      END
```

Winner, 1995 International Obfuscated C Contest
Best-Game Category
Don Dodson, Glendale, Arizona, USA

```
#include <time.h>
#include <curses.h>
#define P(A,B,C,D,E) mvaddch(b+A,a+B,(q[y]&C)?D:E);
#define O(A,B,C) case A:if(q[x]&B)C;break;
#define R rand()
#define U 0,1,4,5
#define J(x) (1<<x)
#define V ' '


int r[27]                       ={0,J(0)
,2,1 ,3,J                       (2),5,U,
U,U,U,U},                       u[6]={-1
,7,49,-49,-7,1},q[343],x,y,d,l=342,a,b
,j='#';int main(){srand(time(0));for(x
=0;x<343;x++)q[x]=0;x=R%343;while(l){d
=r[R%27];if(((x%7==(x+u[d])%7)+(x/0x31
==(x+u[d]                       )/0x31)+
(((x/7)%7                       )==(((x+
u[d])/7)%                       7))==J(1
))&&(x+u[d]>=0)&&(x+u[d]<343)){if(!q[x
+u[d]]){q[x]+=J(d);x+=u[d];q[x]+=J(5-d
);l--;}else if(R<R/0x7){do{x=R%0x157;}
while(!q[x]);}}}x=294+R%0x31;initscr()
;noecho(                        );crmode
();clear                        (/*|*/);
refresh(                        );while(
x>0){move(J(0),60);printw("Level %d",(
x/0x31)+J(0)); q[x]|=J(J(3));for(y=(x/
0x31)*0x31;y<(J(0)+x/0x31)*0x31;y++)if
(q[y]&J(J(3))){a=J(0)+(3*((y/7)%7));b=
J(0)+(3*                        (y% 7));
mvaddch(                        J(1)+((y
%7)*3),J                        (1)+(((y



/7)%7)*3                        ),V);P(0
,0,0,0,j                        )P(3,0,0
,0,j)P(0,3,0,0,j)P(3,3,0,0,j)P(0,J(0),
J(0),V,j)P(0,J(1),J(0),V,j)P(J(0),3,J(
1),V,j)P(J(1),3,J(1),V,j)P(J(1),J(0),4
,'U',V)P(J(1),J(1),J(3),'D',V)P(J(0),0
,J(4),V,                        j)P(J(1)
,0,J(4),                        V,j)P(3,
J(0),J(5                        ),V,j)P(
3,J(1),J(                      5),V, j)}
 mvaddch(J                     (1)+((x%7
  )*3),J(1)                   +(((x/7)%
   7)*3),'J            ;refresh()
    ;switch(getchar()){O('k',J(0),
      x--)O('j',J(5),x++)O('l',J
        (1),x+=J(3)-J(0))O('h'
          ,J(4),x-=7)O('u',4


,(x+=49,    clear()))O('d',8,(x-=49,clear
()))case   'q':x=-1;break;}}clear();refresh
(/*v*/);   nocrmode ();echo();endwin();if(!
x)printf    ("You Escaped!\n");exit(0);}
```

Physics Z-5156
August 2006

Some items of C syntax compared to other languages

meaning	C	FORTRAN IV/77	Pascal
end of statement	`;`	end of line	; (only between statements)
line continuation	*not needed*	digit in column 6	*not needed*
comment	`/*...*/`	cols. 72–80, `C` col. 1	`(*...*)`
begin block	`{`	`IF`, `DO`, *etc.*	`begin`
end block	`}`	`ENDIF`, *etc.*	`end`
`i` holds an integer	`int i;`	`INTEGER I`	`var i: integer`
`x` holds a real number	`double x;`	`REAL*8 X`	`var x: double`
`s` holds a 100-byte string	`char s[100];`	`CHARACTER*100 S`	`type str100=array[1..100] of character;` `var s: str100`
implicit typing	generally not allowed	allowed, causes trouble	not allowed
composite object	`struct`	`COMMON` (very limited)	`record`
assignment	`=`	`=`	`:=`
equality test	`==`	`.EQ.`	`=`
inequality test	`!=`	`.NE.`	`<>`
Boolean false	`0`	`.FALSE.`	`false`
Boolean true	non-zero	`.TRUE.`	`true`
Boolean subexpressions	stop when determined	evaluate all	evaluate all
parameter passing	by value or pointer	by reference	by value or reference
function return	`return` *value*	*name*=*value*	*name*=*value*
function returning double	`double` *name*`()`	`FUNCTION` *name* `REAL*8` *name*	`function` *name*`: double`
function returning nothing	`void` *name*`()`	`SUBROUTINE` *name*	`procedure` *name*
local storage	stack, constant, or static	implementation-dependent	stack or constant

examples and exercises

1. Hello

```
/* hello.c
 * 7/96 D. Rabson
 * The canonical "hello, world" program in C.
 */
#include <stdio.h>

int
main()
{
        printf("Hello, world\n");
        return 0;                               /* required by ANSI standard */
}
```

2. Temperature table

```
/* ttable.c
 * 7/96 DAR
 * Print a table of Celsius temperatures and their conversions to Fahrenheit.
 *
 * Exercise: modify the print format so that it comes out prettier.
 */

#include <stdio.h>

int
main()
{
        double celsius;         /* hold the Celsius temperature */

        /* For loop: first part is initialization, second is the
         * condition to be tested before each iteration of the loop,
         * last is an action to take at the end of each iteration.  Any
         * of the parts may be blank (although it's not sensible for the
         * second one to be blank).
         */
        for(celsius=5; celsius<=40; celsius+=2)
                printf("%f %f\n", celsius, 1.8*celsius + 32.);

        return 0;               /* ANSI standard requires this. */
}
```

3. Using an array

Make a lookup table of Farhenheit equivalents for integral Celsius temperatures from 0 to 25, then print it. Here's a start. Be sure to add comments.

```
#define LARGEST 25                          /* largest C temperature */
int
main()
{
        double fahrenheit[LARGEST+1];           /* room for 0 to LARGEST */
        void initialize_table(double [], int);          /* below */
        void print_table(double *, int);                /* below */

        initialize_table(fahrenheit, LARGEST+1);        /* initialize */
        print_table(fahrenheit, LARGEST+1);             /* print it out */
        return 0;
}
```

4. Fahrenheit–Celsius

The program is correct as printed here, but you will be given a version with two bugs slipped in. Use a debugger to find them.

```
/* fahr2cels.c
 * 7/96 DAR
 * convert Fahrenheit to Celsius degrees
 *                      C = (5/9)(F - 32)
 *
 * usage:
 *              fahr2cels < input > output
 *                      converts each line of input from Fahrenheit
 *                      to Celsius, ignoring blank lines. Other lines that
 *                      cannot be converted into numbers will be interpreted
 *                      as zero.
 *              fahr2cels number number number ...
 *                      converts each argument from Fahrenheit to Celsius
 *
 * I've written this to demonstrate as many as possible of the items in
 * the syntax-comparison chart: it could be much shorter.
 */

#include <stdlib.h>                     /* include standard declarations */
#include <stdio.h>                      /* standard I/O declarations */
#include <ctype.h>                      /* includes "isspace()" */

/* preprocessor directives: these take effect when the compiler reads
 * the source code.
 */
#define SCALE   (5./9.)                 /* note: no semicolons */
#define OFFSET  (32)

/* Define a preprocessor "function" -- beware side effects! */
#define MAX(a,b)        ( (a)>(b) ? (a) : (b) )

/* Statics: these are accessible by any function in this file. */
static double absolute0 = -273.15;      /* absolute zero in degrees Celsius */

/* The main program: if called with any arguments, it tries to interpret
 * each in turn as a Fahrenheit degree, to be converted to Celsius.  If
 * there are no arguments, the program reads lines of input.
 *
 * Note that argv[0] is the name by which the program was called.
 * Note that we've not declared a type for main().  The type of a
 * function (but not a variable) defaults to "int."
 */
main(int argc, char **argv)
{
        static void print_conversion(double);   /* a function defined below */
        char buf[1024];         /* room to hold a line */
        static char *find_first_nonspace(char *);       /* a function below */
        char *line;                     /* a line we got from fgets() */

        if(argc>1)      {               /* braces optional: block 1 statement */
                while(--argc)           /* ditto -- left out here */
                        /* atof() is declared in stdlib.h */
                        print_conversion(atof(*++argv));
        } else {
                /* fgets() is decalared in stdio.h */
```

```
                while((line=find_first_nonspace(fgets(buf,sizeof(buf)-1,stdin)))
                   != 0 ) {             /* the !=0 is optional */
                        if(!line[0])
                                continue;       /* blank line */
                        print_conversion(atof(line));
                }
        }
        return 0;                       /* ANSI C requires return value */
}

/* Print the conversion of a double from Fahrenheit to Celsius. */
/* "static" means that it can be called only from this file. */
static void
print_conversion(double f)
{
        /* This is a somewhat silly way to initialize the scale factor
         * of 5/9, but I do it this way for demonstration purposes.
         */
        static double scale = 0;        /* 0 before this is ever run */
        double c;                       /* hold a degrees-Celsius temperature */

        if(!scale)                      /* first time run only */
                scale = SCALE;

        /* I don't combine these two lines into one, because the MAX()
         * preprocessor macro evaluates one of its arguments twice.
         */
        c = scale*(f-OFFSET);
        printf("%f\n", MAX(c,absolute0) );      /* print Celsius conversion */
}

/* Find and return a pointer to the first non-space in a string. */
static char *
find_first_nonspace(char *s)
{
        /* I've deliberately written this less than optimally in order to
         * demonstrate some pieces of syntax.
         */
        while(s && *s && isspace(*s) && s++)
                ;
        return s;
}
```

Mechanics of compiling and debugging a C program

simple program (be sure not to overwrite the .c file.)

```
cc -g -o program program.c
```
(-g allows debugging)

```
program
```
to run

linking two files

```
cc -g -c prog1.c
cc -g -c prog2.c
cc -g -o program prog1.o prog2.o -lm
```

simple Makefile for the previous example (tabs, not spaces, mandatory)

```
shell% cat Makefile
CFLAGS=-g
LDLAGS=-g
LIBS=-lm                                    # -lm to link to math library
program:        prog1.o prog2.o
                cc $(LDFLAGS) -o program prog1.o prog2.o $(LIBS)
shell%
shell% make
```

information on linking C to FORTRAN

see the file `/home/5156/doc/example.c-prog`

optimization (faster code)

replace `-g` (debugging) with `-O`

alternative compiler

`gcc` (actually, on Linux, `cc` *is* `gcc`)

alternative compiler

`pgcc` (commercial; sometimes produces faster code)

A typical compilation actually goes through several steps. For instance, when I say `cc -c program.c`, the file `program.c` is first passed through the C preprocessor, which expands the `#` directives. It then goes through one or more passes of the main compiler, including optional optimization (`-O`), to create a temporary assembly-code file. The assembly code can be saved in a `.s` file with the `-S` switch. The assembler, `as`, is then called to create an object (`.o`) file, which is finally linked together by `ld` with one or more startup modules and numerous default and user-specified run-time and static libraries (see the `ld` manual page for the `-l` and `-L` flags) to create an executable file.

Debugging *"Don't make mistakes; they're a waste of time."*

`TRACER` Scattering `printf` calls throughout a program is often the easiest debugging method. My files `/home/5156/rabsonlib/src/debug.h` and `/home/5156/rabsonlib/src/debug.c` show an unnecessarily fancy way of implementing tracers.

`dbx` Standard Unix debugger: available on most platforms but a little non-Unix-like in its syntax. Not available on Linux.

`gdb` Similar to `dbx`, different syntax. Use the `help` comamnd for help. See also my initialization file, `/home/5156/rabsonlib/.gdbinit`.

`ddd` Graphical front end to gdb. This works fairly well and is self-explanatory.

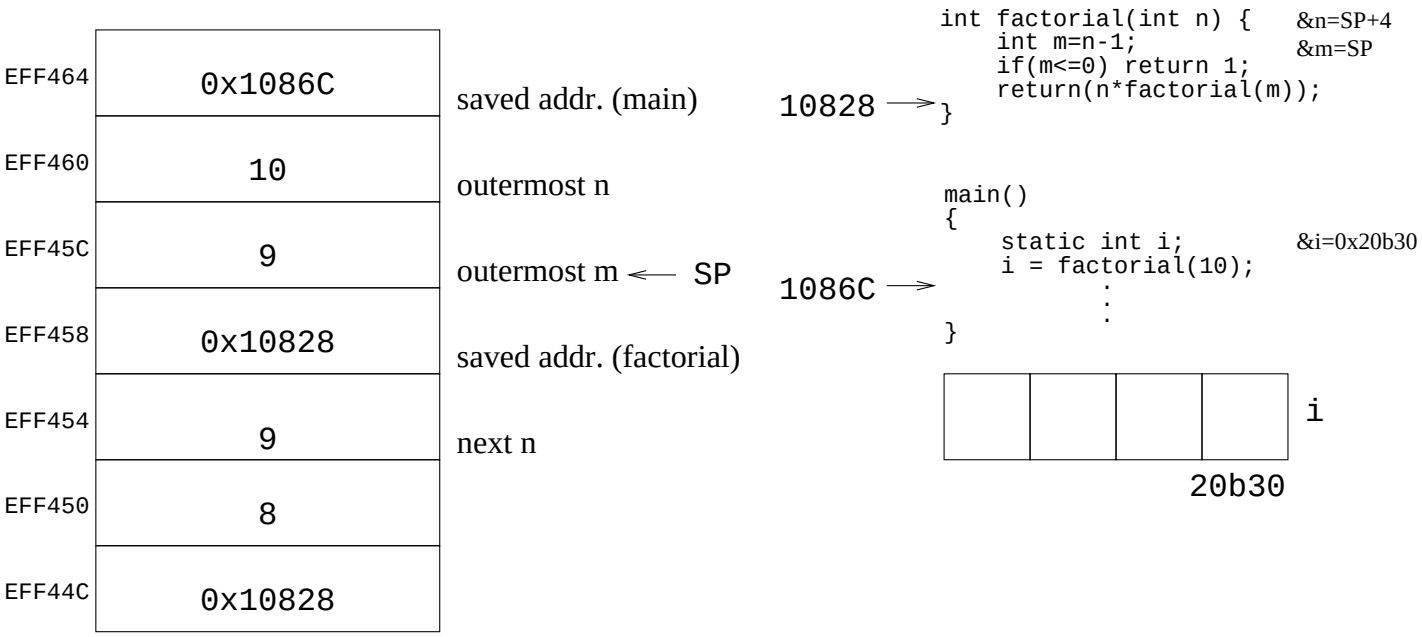
`xxgdb` Graphical front end to gdb; not as fancy as ddd, but works without the Motif library.

`pgdbg` This graphical debugger comes with the Portland-group compilers and may be less buggy than `ddd` and `xxgdb`. Use the `-text` flag, as the graphical interface appears to be broken.

`adb` assembly-level debugger for Real Programmers. It even has an option for examining and modifying variables in a running kernel. *(Not available on Linux.)*

`gprof` A profiler, not a debugger. It is rarely good practice to write every routine to be as fast as possible. Rather, you will find that most numerical programs spend 90% of their time executing 1% of the code. Compile with `-pg`, then run `gprof` to find the 1% of code whose efficiency matters.

The stack (schematic: implementation-dependent)

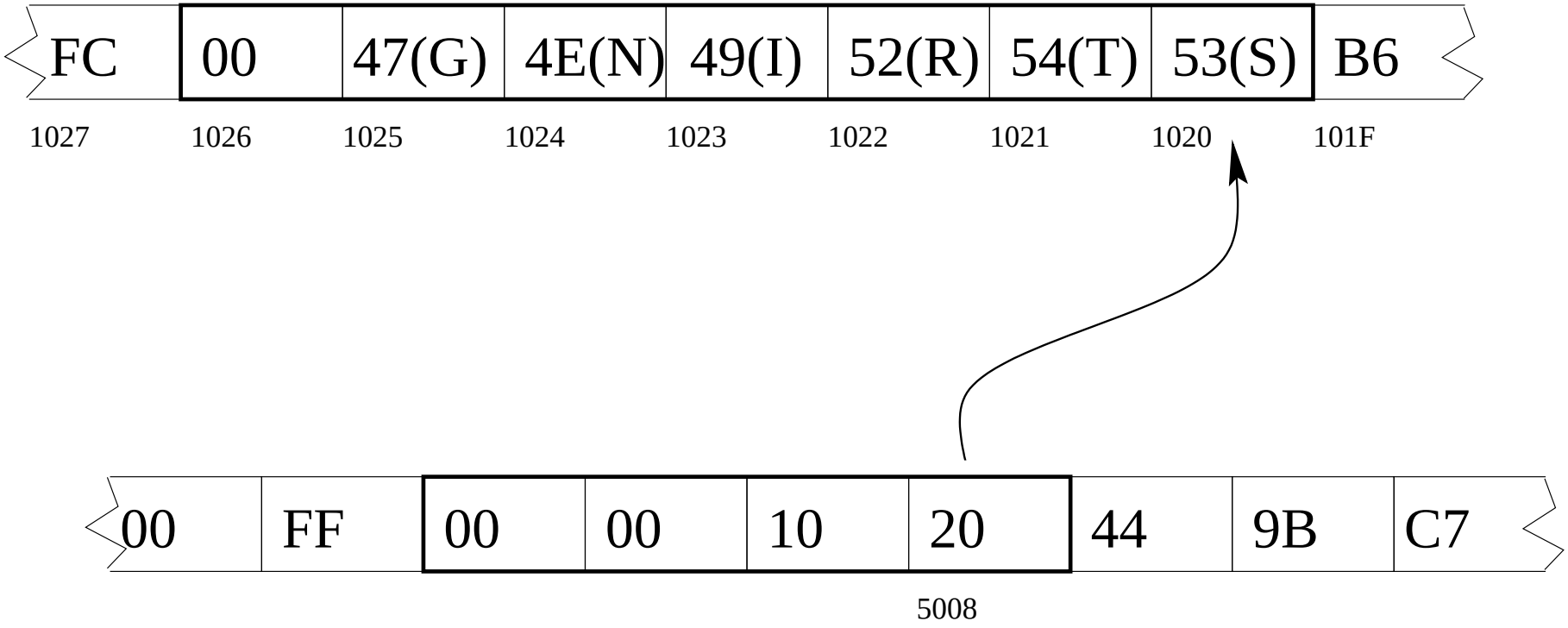


This picture is oversimplified. Automatic variables will often be put into registers and saved on the stack along with return addresses. Also many machines have both a stack and a frame pointer to reduce the overhead of each subroutine. The *model*, however, is right for the C language, which does not depend on hardware.

storage (definitions boxed; declarations underlined)

	storage	**initialization**	**scope**
`int lineno=0;`	static: address fixed	compilation time	global (if declared elsewhere)
`static char v[]="1.2";`	static	compilation time	this `.c` file
`extern double d;`	static	another file	this `.c` file and the other
`static char *g(char *);`		see below	this `.c` file
`void *my_own_malloc(unsigned int);`		another file	this file and wherever declared
`void func(void)`			global
`{`			
`char *s=v;`	automatic (stack)	each invocation	func()
`static int count=0;`	static	compilation *only*	func()
`extern int glob_foo;`	static	another file	func() and elsewhere
`s = g(++s);`			
`printf("%s.%d\n", s, ++count);`			
`}`			
`static char * g(char *x)`	`x` automatic		`g`: this file; `x`: this func. (`g`)
`{`			
`char *y=malloc(strlen(x));`	`y`: automatic `y[0]`, *etc.*: malloc	each invocation hasn't happened yet	this function global (if `y` passed)
`y[0]='\0';`		y[0] here	global
⋮			
`return y;`			
`}`			

Pointers and arrays Physics Z5156




```
static char arr[] = "STRING";
char *p = arr;                  /* same as &arr[0] */
printf("p=%x=%s\n",p,p);    /* p=1020=STRING */
printf("&p=%x\n",&p);       /* &p=5008 */
printf("&arr[0]=%x\n",&arr[0]);       /* 1020 */
printf("arr[2]=%c\n", arr[2]);        /* R */
printf("p[2]=%c\n", p[2]);            /* R */
printf("%d,%d\n",sizeof(p),sizeof(arr)); /* 4,7 */
```

Some mistakes in C

```
        /* Get a string from standard input, truncating at the newline.
         * Use this instead of gets() because gets() can overflow.
         * THIS CODE IS WRONG!!!!!
         */
        char *
        safe_gets()
        {
                char buffer[512];
                char *rvalue=fgets(buffer, sizeof(buffer), stdin);
                return rvalue;
        }
```

```
/* The string sent to this routine should be of the form <integer> <double>.
 * The caller passes variables i and x to stuff with these two numbers.
 * We return 0 on success, -1 on trouble.
 * THIS CODE IS WRONG!!!!!
 */
int
get_int_and_double(char *input, int i, double x)
{
        if(sscanf(input, "%d %lf", i, x)==2)
                return 0;
        else
                return -1;                      /* failure */
}
```

```
/* Return 1 if the three integers are all equal, 0 otherwise.
 * THIS CODE IS WRONG!!!!!
 */
int
equal3(a, b, c)                 /* old-style function: this isn't the bug. */
int a, b, c;                    /* this line optional in old style */
{
        if( (a=b) && (b=c) )
                return 1;
        else
                return 0;
}
```

```
                /* Increment the integer pointed to by ip.
                 * THIS CODE IS WRONG!!!!!
                 */
                void
                incr(int *ip)
                {
                        *ip++;
                }
```

Structures
A binary tree

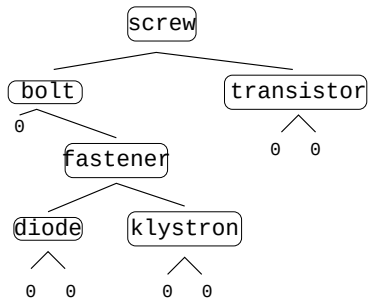
```
/* bintree.c
 * 7/96 DAR
 * Sort input lines alphabetically w/ binary tree.
 *
 * One of the chief advantages of a binary tree
 * (although this program does not exploit it)
 * is the ability to mix additions and deletions
 * yet with the tree completely sorted at all
 * times.  For large applications, such a tree
 * should use balancing techniques, which I have
 * not implemented here.
 */

#include <stdlib.h>
#include <stdio.h>
#define ALLOC(x) ((x *)malloc(sizeof(x)))

/* The binary-tree node */
struct node {
        char *string;
        struct node *left;
        struct node *right;
};
/* copy a string into dynamic storage */
char *copy(char *s) {
        char *rvalue = malloc(strlen(s)+1);
        strcpy(rvalue, s);
        return(rvalue);
}
/*Copy string into dynamic store & return node.*/
struct node *newnode(char *string) {
        struct node *rvalue = ALLOC(struct node);
        rvalue->string = copy(string);
        rvalue->right = rvalue->left = 0;
        return rvalue;
}
/* Truncate a string @ first newline, if any. */
char *stripnl(char *s) {
        char c, *rvalue=s;      /* original s */
        if(!s) return 0;
        while(c= *s)    {
                if(c=='\n') {
                        *s = 0;
                        return rvalue;
                }
                s++;
        }
        return rvalue;
}
/* Find where a string belongs in a binary tree:
 * return pointer to stuff.
 */
struct node **
locate(char *s, struct node **rootptr) {
        struct node *n= *rootptr, **r;
/* if tree is empty, stuff root ptr */
        if(!*rootptr) return rootptr;
        while(n) /* traverse tree */
                n = *(r=(strcmp(n->string,s)>0
                    ?&n->left :&n->right));
        return r;
}
```

Input:

```
screw
bolt
fastener
diode
transistor
klystron
```




```
/* Add a string to the binary tree. */
void add(char *s, struct node **rootptr) {
        /* make node & find where to put it */
        *(locate(s,rootptr)) = newnode(s);
}
/* Print the binary tree, recursively. */
void out(struct node *node) {
        if(!node) return;           /* no more */
        out(node->left);            /* before this */
        printf("%s\n", node->string);/*print this node*/
        out(node->right);           /* after this */
}
/*Main routine: take input until done, then print tree*/
main() {
        struct node *root=0;        /* the root */
        char buf[256];              /* room for input*/
        while(fgets(buf,sizeof(buf)-1,stdin))/*read*/
                add(stripnl(buf), &root);   /*&add*/
        out(root);                  /* print tree */
        return 0;
}
```

I wrote this style guide in 1997 while serving as director of an academic computing facility; it prescribes a common style for all the systems programming in that facility.

C Style Manual

10/97 D. A. Rabson (edited 8/06)

According to the parody *Real Programmers Don't use PASCAL*, perhaps the earliest and best known piece of computer literature, "the determined Real Programmer can write FORTRAN programs in any language." Unfortunately, many do. There are three related distinguishing features of FORTRAN programs written in C:

1. Their subroutines go on for pages and pages;
2. Coming back after a year, the author can't understand any of the code;
3. The author's colleagues can't understand it, period.

I have come back to C programs ten or more years after writing them and understood every detail of the code immediately. Most of these guidelines aim to help others do the same. Others acknowledge that we are working together in a group with high student-programmer turnover and so enforce my own style – or idiosyncrasies – so that all the code in the facility can be modified in a consistent manner.

Directory:

Every directory should have a README, explaining the program in it, and a Makefile. The README will usually epitomize each file or subdirectory in the directory.

File header:

Every file should begin with a header describing the file's purpose, interfaces (if any), who wrote it, and when. The description can be abbreviated, as a README or a manual page in the same directory will usually contain more details. Example:

```
/* help.c
 * 8/96 D. Rabson
 * simple utility to peruse a help tree
 *
 * usage: help [topic]
 *
 * Interactive program to let the user read topics
 * in the help tree: see manual page, help.1.
 */

#include <stdlib.h>             /* include standard prototypes */
#include <stdio.h>
```

If the file serves as a library, the header should contain calling descriptions of all externally-callable subroutines contained in it.

Almost all C files should include the standard library headers, as shown above. The manual page for a library function will indicate whether additional header files are needed; for example, index(3) calls for `#include <string.h>`.

Subroutines (functions):

Subroutines must be readable. Achieve readability with short, well-commented subroutines, each of which does one simple-to-explain thing. Split subroutines up as necessary.

A subroutine should generally contain no more than 24 lines, so it will fit in a single small window.

Subroutine (function) comments:

Every subroutine should begin with a header describing the purpose, calling sequence, and return value of the subroutine. If the subroutine is called by some other subroutine or more, name the callers. Example:

```
/* called by go() below.  If s (the path a user specified
 * for a file, not a directory) includes a /, call directory().
 * The calling routine (go) has just printed the requested
 * file; this has the effect of changing to the directory
```

```
 * in which the file resides just before go() issues an
 * ls(".") directive.
 */
static void
path_modify(char *s)
{
```

Subroutine (function) headers:

Subroutine names must be flush against the left of lines so that `/^name/` will work in regular-expression editors. Any declaration keywords precede the function name on the previous line, as in the "path_modify" example above. Subroutines needed only in the current file should always be declared static. ANSI style (as above) is preferable to KR (*e.g.*, `path_modify(s) char *s;`) because many compilers that can detect errors using ANSI prototypes will ignore them in KR style.

Indentation:

Each new level (e.g., subroutine, block statement, struct) should indent one further tab. Tabs should be used, not spaces (Emacs users may need to invoke a command like "`<ESC> x fundamental`" to prevent the editor from imposing its own style.). Indentation must be consistent. Example:

```
int
foo(int n)
{
        int i;                  /* loop variable */
        double f=1.0;           /* calc number of flobs */

        /* go through n steps of the algorithm */
        for(i=n; i--; ) {
                f *= bar(i, n); /* renormalize by boozles */
                while(f < i)    /* step 2 of algorithm (see top) */
                        f = f*(f-1.) + 1.;
        }
        return (int) g(f);      /* return number of flobs per boozle */
}
```

If there are too many levels of indentation, it is time to split the subroutine into several subroutines, not to indent less each level.

Declarations:

In most cases, comment each declaration, so the reader knows what the variable does.

Loops:

Block loops should be structured like the "for" loop in the example above, with an open brace on the same line as the "for" and a close brace on a line by itself with the same indentation as "for."

Single-line loops should be indented like the "while" loop in the example above.

Line wrapping:

If a line is too large to fit in 80 columns, continue it on the next line with two to four leading spaces in a way that makes clear the continuation. Examples:

```
1.      for(i=n,j=0;  quumquaat(i,j,f)<qmax
           ;i--,j++)    {


2.      printf("%s %d 0x%x\n"
           ,"now is the time", i+foobar(x)
           ,dt);
```

Strings may need to be wrapped to the left margin, as in

```
                                printf(
"Now is the time for all good men to come to the aid of their country.\n\
The quick brown fox jumped over the lazy dogs.\n");
```

but it is better to avoid this when possible.

Floating-point numbers:

Use doubles in preference to floats except where storage size is important (e.g., in a large matrix or raw binary output). The speed improvement from single precision (float) is small (around 20%) and realizable only with special compilation flags: naive substitution of floats for doubles will actually slow a program down on modern hardware.

Line buffer overflow:

Make buffer overflow on input impossible under all circumstances.

Use fgets() with a buffer, never gets(). The former will truncate too long a line at the buffer size, where the latter will overflow:

```
char buf[BUFSIZ];

fgets(buf, sizeof(buf), stdin);
```

Note the use of sizeof instead of a magic number or even BUFSIZ. Be careful, however, not to use sizeof() when the buffer is passed as a character pointer (*e.g.*, char *buf). In this case, pass the size in a variable and use that size.

Where a line size cannot be predicted and could be arbitrarily long, write a short subroutine using getchar(3), malloc(3), and realloc(3) to read a line of input, *e.g.*, agets(3) in rabsonlib.

Line-oriented input:

Use fgets(3) and sscanf(3) in preference to scanf(3). The latter will hang on a line with too few fields, whereas with the former, one can verify the expected number of fields:

```
char buf[BUFSIZ];
int i, lineno=1;
double x;

while(fgets(buf, sizeof(buf), stdin))    {
        if(sscanf(buf, "%d %lf", &i, &x)!=2)
                f_err("bad input, line %d\n", lineno);
        .
        .
        .
```

More on buffer overflow:

To avoid buffer overflow with format %s on sscanf(3), either allocate enough room for each scanned string variable to hold the entire buffer, or else use a splitting subroutine instead of sscanf(3) for the parsing. See split(3) in rabsonlib.

Decision trees:

For two or three choices, it is acceptable, if the resulting subroutine fits in 24 lines, to use if ... else ... else ... endif. However, if the number of choices becomes too large to see all at once, substitute switch ... case ... case if doing so pares the number of lines to 24. This is more readable as well as more easily optimized into a lookup table by the compiler.

For more than half a dozen choices, it may be better to use a lookup table. Where performance is important, the lookup table should be hashed or in alphabetical order (to enable a binary search). Example of a simple lookup table:

```
/* list of options */
struct choice {
        char *command;          /* name of command */
        int (*f)(int);          /* call this func. on command*/
} table[] = {
        { "exit", f_exit },
        { "print", f_print },
        { "shakespeare", f_compose_sonnet },
        { 0, 0 }
};
```

An auxiliary subroutine finds a match for "command" and calls the corresponding "`f_`" function:

```
/* Find and execute the f_*() subroutine named by "command" with argument i.
 * This simple iterative search could easily be replaced with a binary search.
 * Return the return value of the f_*() function, or return -1 on no match.
 */
static int
do_command(char *command, int i, struct choice *tab)
{
        int (*g)(int);                          /* g is a pointer to a func. */

        while(g = tab->f)                               /* assign g */
                if(!strcmp(command, tab++->command))    /* compare & advance */
                        return (*g)(i);         /* match: run the function */
        return -1;                              /* no match */
}
```

Error checking:

Check for errors and unexpected input on any step that is likely to fail. If possible, recover. If not, print a meaningful message and die. I have subroutines `f_err(...)` and related that combine the functionality of fprintf(3), perror(3), and exit(3).

Physics PHZ-5156-C, fall semester 2006

Some important extensions of Fortran-90 over Fortran-77
IGNORE THIS IF YOU DON'T PLAN TO USE FORTRAN

The goals in scientific programming are first to get correct answers and second to write code that is modular, clear, and can be modified later. These goals can be achieved in any of a variety of languages, and students may use any language for which we have a compiler on `physics`. While I shall give examples in the `C` programming language, some students may prefer to use `C++` or `Fortran-90`.[1]

Some older variants of `Fortran`, such as `Fortran-IV` and `Fortran-77`, were popular in scientific programming before 1980. Indeed, we'll make use of numerical libraries, such as `LAPACK`, that have been implemented in `Fortran-77`. While parts of the projects can be successfully completed in `Fortran-77`, the language lacks several essential capabilities. Students who decide to use `Fortran-77` will need either to write parts of their code in a language like `C` that has these features and then link the object files together[2] or else use `Fortran-90`. `Fortran-90` claims all of `Fortran-77` as a (small) subset, so it should not be too painful to upgrade. To invoke a `Fortran-90` compiler on `physics`, use the command `pgf90`. The most common arguments are the same as for the Fortran-77 compiler, `f77`, or the `C` compiler, `cc`.

`Fortran-90` is a very large language, and I cannot hope even to summarize its extensions over `Fortran-77`. However, there are three features that will be enormously useful in this class.

1. In all operating systems used in scientific research (mostly Unix, but also VMS and DOS), the primary means by which the program receives input from the user or from other programs is through the command line. For example, to tell `myprog` that it should use a 23×23 matrix, I might invoke it as

   ```
   prompt% myprog 23
   ```

   Amazingly, no version of the `Fortran` standard contains any provision for reading the number 23 from command line. There appears to be a *de-facto* standard using function `IARGC` and subroutine `GETARG`, illustrated below (Figure 1). It worked on two different compilers that I tried, but there's no guarantee that it will work on all compilers. Figure 2 shows how to convert a numerical command-line argument (such as "23") from a string to a number.

2. Most scientific programs have arrays (one or two-dimensional) of a size that depends on some parameter in the problem at hand, such as the number of sites in a simulation. This parameter is variable and might not be known until the program is running. Old-fashioned `Fortran-77` programers were forced either to recompile their programs for each new set of parameters (which is time consuming) or to fix the maximum dimension each array is ever "likely" to achieve. Either technique is prone to error. The `ALLOCATE` statement in `Fortran-90` lets the program allocate memory as it is needed. Figure 2 illustrates dynamic memory allocation.

---

[1] `Fortran-90` and `C++` are examples of "object-oriented" languages, which provide certain advantages over `C` and `Fortran-77`, such as operator overloading. They also have enormously more syntactic elements. In my experience as a computer-center manager, I found that each programmer writing in `C++` knew a different subset of the language, and that computer-science students who had learned only `C++` tended to misuse language features, resulting in subroutines that went on for pages, could not be read or modified by anyone else, and were in no sense modular. This is what I call "object-disoriented programming" and is part of the reason I do not recommend `C++` or `Fortran-90` for new programmers.

[2] On `physics`, give the command "`help linking_languages`" for information on linking `C` and `Fortran`.

```
physics.cas.usf.edu 1% cat myprog.f90
! myprog.f90
! DAR 8/04
! Demonstrate command-line reading in Fortran.
! This program echoes each command-line argument on a separate line.
! usage: myprog [arguments]
!
PROGRAM MAIN
IMPLICIT NONE
CHARACTER(15) :: BUF            ! truncate at 15 characters
INTEGER :: I, J, IARGC
I = IARGC()                     ! get the number of arguments
DO J=1,I
        CALL GETARG(J, BUF)     ! get the j'th argument,
        WRITE(6, *) BUF         ! and write it out
END DO
END
physics.cas.usf.edu 2%
physics.cas.usf.edu 2% pgf90 -o myprog myprog.f90
physics.cas.usf.edu 3%
physics.cas.usf.edu 3% myprog one two
 one
 two
physics.cas.usf.edu 4% myprog
physics.cas.usf.edu 5%
```

Figure 1. Example for numbered paragraph 1.

3. Old versions of `Fortran` provided a limited number of scalar data types: `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, and `CHARACTER`. Sometimes, it is useful to package different types together; it is almost essential in order to implement data structures such as linked lists. As a simple example, one might have a database of employee records, where each employee had a name, a salary, and the year of hire. One would like a single variable type with three fields: `CHARACTER` for the name, `REAL` for the salary, and `INTEGER` for the year of hire. The old `COMMON`-block mechanism was woefully inadequate, since only one package of each "type" could exist. The `TYPE` keyword of `Fortran-90` (analogous to `struct` in `C`) allows the user to create a new data type. Figure 3 shows a Fortran-90 code that does the same thing as the `bintree.c` example in the Week-1 packet. Bintree.f90 also demonstrates a skeletal implementation of variable-length strings.

In writing in Fortran (whether Fortran-90 or Fortran-77), the programmer should avoid bad habits common to archaic Fortran-IV/66 codes. These include lack of comments (use the `!` feature, which allows comments on the same line as code), subroutines that go on forever (try to limit each subroutine to 24 lines, so it will fit in one standard `xterm` window), and the use of magic numbers. Above all, declare every variable explicitly, and begin every procedure with the line

`IMPLICIT NONE`

(See the lower-left box on page 3 of the week-1 pcaket.)

The examples in this brochure are available on `physics` in the directory

`/home/5156/examples/fortran90`

```
physics.cas.usf.edu 1% cat allocdemo.f90
! allocdemo.f90
! DAR 8/04
!
! Demonstrate memory allocation in Fortran-90.
!
! Allocate, fill, and print out a one-dimensional array of
! arbitrary length specified on the command line.
!
! usage: allocdemo n
! where n is the length

PROGRAM MAIN
IMPLICIT NONE
CHARACTER(15) :: BUF
INTEGER, POINTER, DIMENSION(:) :: A     ! pointer to 1-dim array
INTEGER :: N, IARGC                     ! N=the length
IF(IARGC().LT.1) GOTO 1                 ! no argument: error
CALL GETARG(1,BUF)                      ! get the argument
READ(BUF,*) N                           ! convert string->number
IF(N.LT.1) THEN                         ! N must be >= 1
1       STOP 'SYNTAX: ALLOCDEMO N'      ! die
ENDIF
ALLOCATE(A(N))                          ! allocate N integers
CALL DOFILL(A,N)                        ! fill it
CALL DOPRINT(A,N)                       ! print it out
DEALLOCATE(A)                           ! done with it
END

! Fill an array with integers from 1 to its length
SUBROUTINE DOFILL(P,N)
IMPLICIT NONE
INTEGER :: N, P(N), I
DO I=1,N
        P(I) = I                        ! fill it
END DO
END

! Print out an array of length N
SUBROUTINE DOPRINT(P,N)
IMPLICIT NONE
INTEGER :: N, P(N), I
DO I=1,N
        WRITE(6,*) P(I)
END DO
END
physics.cas.usf.edu 2%
physics.cas.usf.edu 2% pgf90 -o allocdemo allocdemo.f90
physics.cas.usf.edu 3%
physics.cas.usf.edu 3%
physics.cas.usf.edu 3% allocdemo 3
            1
            2
            3
physics.cas.usf.edu 4%
physics.cas.usf.edu 4% allocdemo
SYNTAX: ALLOCDEMO N
```

Figure 2. Example for numbered paragraph 2.

```
! bintree.f90
! DAR 8/04, rev 8/06
! Sort input lines alphabetically w/ binary tree.
!
! This does the same thing as my C-language bintree.c but is not an exact
! translation.  (It is possible but very ugly to implement pointers
! to pointers in Fortran-90).

! This code is prefaced with two modules: the first defines type(string)
! and a couple of interfaces to it, the second type(node) and some
! interfaces.

! In addition to the binary-tree data structure, this example illustrates
! input/output, modules, operator overloading, a variable-length
! string class, and pointers.
! IMPORTANT NOTE TO C PROGRAMMERS: pointers in Fortran-90 are not
! the same as pointers in C.

! Order of modules: it's not supposed to matter, but it appears to
! help if the string module comes before the node module (which needs
! the string module).

!-----------------------------------------------------------------------
! This implements a minimal "string" class for variable-length strings
! (which, incredibly, are not supported directly in Fortran 90).
! All it knows how to do is to create a string from a CHARACTER*(*)
! and to fill a CHARACTER*(*) from a string.
! In the latter case, there is no protection against buffer overflow.
! Usage:        type(string) s
!               character*(*) c
!               s = c                           ! copy string from character
!               c = s                           ! copy character from string
! The first assignment copies from the character*(*) c to the string s,
! ignoring any trailing blanks in c.  If s has already been written,
! its old contents will be deallocated first.  The second assignment
! copies from the string s to the character*(*) c.
!
module string_module
        type string
                character, pointer :: s(:)              ! pointer to string
                integer :: len                          ! length
        end type string
        interface assignment(=)                         ! conversion copies
                module procedure stringchar             ! string <= char
                module procedure charstring             ! char <= string
        end interface
contains
        ! copy source of type character*(*) to destination of type(string)
        subroutine stringchar(dst, src)
                IMPLICIT NONE
                type(string), intent(out) :: dst
                character(*), intent(in) :: src
                integer :: i
                if(associated(dst%s)) deallocate(dst%s) ! erase old string
                dst%len = len(trim(src))                ! how long a string?
                allocate(dst%s(dst%len))                ! allocate room
                do i=1,dst%len                          ! copy the string
                        dst%s(i)=src(i:i)
                end do
        end subroutine stringchar
        ! copy source of type(string) to a character*(*)
        ! ASSUME the latter is big enough (segmentation fault if not)
        subroutine charstring(dst, src)
                IMPLICIT NONE
                type(string), intent(in) :: src
                character*(*), intent(out) :: dst
                integer :: i
                do i=1,src%len
                        dst(i:i)=src%s(i)
                end do
        end subroutine charstring
end module string_module
!-----------------------------------------------------------------------
! The binary-tree node
! Interfaces:
!       call newnode(node, c)
!               where c is a character*(*): initialize a new node
!       call placenode(s, root)
!               root is a pointer to the root (or nil), s a character array
!               Find the correct place to insert a new node, and do so.
!
! A full implementation would also have a freenode() interface.
module node_module
        use string_module                               ! see above
        type node
                type(string) st
                type(node), pointer :: left, right      ! children of this node
        end type node
contains
        ! Allocate a new node: dst is a type(node); src is a character*(*)
        subroutine newnode(dst, src)
                IMPLICIT NONE
                type(node), intent(out) :: dst
                character(*), intent(in) :: src
                dst%st = src                            ! see string_module
                nullify(dst%left, dst%right)            ! no children yet
        end subroutine newnode

        ! Note: it is essential for out, placenode, placeonde1, and placenode2
        ! to be in the module, since according to documentation, the pointer
        ! attribute applied to a dummy argument doesn't work unless
        ! there is an explicit interface.

        !Find the right place for a character*(*), and put a node there.
        ! s is the character*(*), root a pointer to the root node(or a null ptr)
        !The root pointer will be modified if it is originally nullified
        subroutine placenode(s, root)
                IMPLICIT NONE
                character*(*), intent(in) :: s
                type(node), pointer :: root

                if(associated(root)) then               ! root already present?
                        call placenode1(s, root)                ! hard case
                else
                        call placenode2(s, root)                ! easy case
                end if
        end subroutine placenode

        ! placenode() calls this when root points to nothing
        subroutine placenode2(s, root)
                IMPLICIT NONE
                character*(*), intent(in) :: s
                type(node), pointer :: root

                allocate(root)                          ! allocate a node
                call newnode(root, s)                   ! fill it
        end subroutine placenode2

        !placenode() calls this when root exists -- call sequence as placenode's
        ! Three hacks: (1) instead of writing comparison interfaces in the
        ! string module between strings and character*(*)'s, I convert
        ! the string back to a character*(*). (2) Pointers to pointers
        ! are really messy in Fortran-90, so I use 'flag' to keep track
        ! of whether the most recent traversal was to the left or to
        ! the right. (3) strings longer than 256 won't be compared correctly
        subroutine placenode1(s, root)
                IMPLICIT NONE
                character*(*), intent(in) :: s
                type(node), pointer :: root, ptr, prevptr
                integer, parameter :: bufsiz=256        ! for buf
                character(bufsiz) :: buf                ! for string compar.
                integer flag                            ! hack: 0=left, 1=right

                ptr => root                     ! ptr points to root node
                do while(associated(ptr))       ! find place by traversing tree
                        prevptr => ptr          ! save old pointer
                        buf = ptr%st            ! hack for string comparison
                        if(s < buf) then
                                ptr => ptr%left
                                flag = 0        ! hack: lack of C's **
                        else
                                ptr => ptr%right
                                flag = 1        ! hack
                        endif
                end do
                allocate(ptr)                   ! new node
                call newnode(ptr, s)            ! initialize it
                if(flag.eq.0) then              ! is it left child?
                        prevptr%left => ptr     ! set the left field
                else                            ! or right?
                        prevptr%right => ptr    ! set the right field
                end if
        end subroutine placenode1


        ! Print out the binary tree (recursive)
        recursive subroutine out(n)
                IMPLICIT NONE
                type(node), pointer :: n

                if(.not.associated(n)) return           ! no more
                call out(n%left)                        ! before this
                write(6,*) n%st%s                       ! print this node
                call out(n%right)                       ! after this
        end subroutine out

end module node_module
!-----------------------------------------------------------------------

! Main program: read lines in to binary tree, then print them out.
program main
        use string_module
        use node_module
        IMPLICIT NONE
        integer, parameter :: bufsiz=256        ! for reading one line
        character(bufsiz) :: buf, format        ! one line of input; format
        type(node), pointer :: rootptr          ! point to root of tree

        ! Default format won't read a whole line: must construct format string
        write(format, *) '(A', bufsiz, ')'      ! for read(5,*) below
        nullify(rootptr)                        ! at first, no tree
        do
                read(5,format,end=1)  buf       ! read line; go to 1 on EOF
                call placenode(buf, rootptr)    ! place a new node in tree
        end do
1       call out(rootptr)                       ! output
end
```

Physics Z-5156, fall 2006

Chapter 2, Part I: probability distributions, kernel smoothing (nonparametric estimation)
short assignemnt

Copy the question files and suggested program templates from `/home/5156/assignments/kernel`*, for example using the following commands:*

```
cd                                          start at your home directory
cp -r /home/5156/assignments/kernel .       copy the directory to all levels
cd kernel                                   begin working in this directory
```

When you have completed everything, the `make` *command should compile and demonstrate your programs. Feel free to modify the* `Makefile` *if you wish. When you are done, issue the* `make submit` *command to tell the instructor you have finished. Do not change anything in the directory after running* `make submit`.

Last week we wrote some utilities for making and displaying histograms of data. A histogram might be useful in a stochastic measurement as a rough estimate of the probability distribution, *i.e.*, the probability of getting a measurement between x_0 and $x_0 + \Delta$ (bin 0), the probability between $x_0 + \Delta$ and $x_0 + 2\Delta$ (bin 1), *etc.* The histogram can be thought of as a rather blocky, stepwise continuous, approximation to the true probability distribution. As an alternative, we propose now to interpolate a continuous estimate of that distribution.

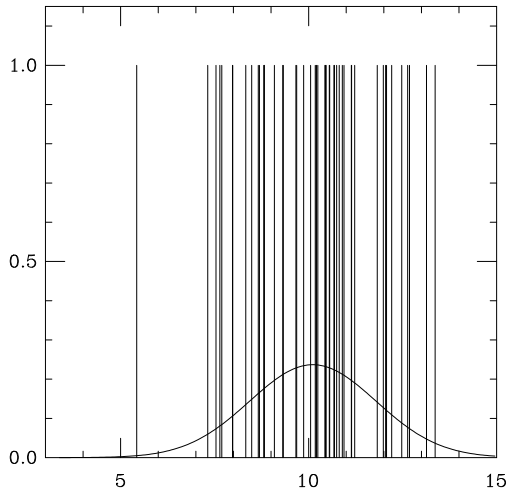


Fig. 1

Figure 1 plots as vertical bars the 50 (artificial) measurements stored in the file `/home/5156/data/stochastic.dat`. If we believe there is one "true" value for the measured quantity, we could characterize the data by their mean, standard deviation, skewness, and (in principle) kurtosis and further cumulants (see *Numerical Recipes*, second C edition, chapter 14). If we're sure that the random fluctuations about the "true" value should be Gaussian, the mean μ and standard deviation σ define the probability distribution completely. The solid bell-shaped curve is the normalized Gaussian $\frac{1}{\sigma\sqrt{2\pi}} \exp(-(x-\mu)^2/2\sigma^2)$ using the best inferred values for μ and σ.

1. We have supplied a two-line `awk` script, `mean`, that calculates the mean of a list of numbers. Copy it into `dist`, and modify the script to calculate both the mean and the standard deviation of the data in `/home/5156/data/stochastic.dat` as given by the square root of *Numerical Recipes* equation 14.1.7. For the purposes of this exercise, you needn't implement the two-pass algorithm, equation 14.1.8, unless you wish: the roundoff won't be very severe. **When you have completed this, the command `make 1` should run `dist` on the given file, printing out the mean and standard deviation.** The modified `awk` script should still be only two lines long, so this is a quick exercise.

Armed with an estimate of the probability distribution of your real data, you could create further stochastic data following the same distribution. You might test the hypothesis that the original data were Gaussian distributed by comparing experimentally-observed behavior with that of computer-generated numbers. *Numerical Recipes* devotes a chapter to random-number generation. We may have an opportunity to use it later in the course for Monte-Carlo integration.

We have so far imagined that the quantity underlying our data is a single Gaussian. What, however, if there are two peaks, or three, or if it's really a Lorentzian? We can fit each hypothesis, and *Numerical Recipes* suggests a number of tests and algorithms. The general idea is to make a model with some number of parameters and then find the set of parameters (not necessarily unique) that best fits the data using some criterion such as the least-square deviation. For the sum of two Gaussians you would have five parameters: the mean and standard deviation of each of the Gaussians and the ratio of their weights.

Unless you have a very large set of data, this can be a dangerous game. With n data points, you can invent a model with n parameters that fits the data exactly, but that doesn't mean that the model is right. I don't know where the adage originated, but it has been observed that "*with enough parameters, you can fit an elephant.*"

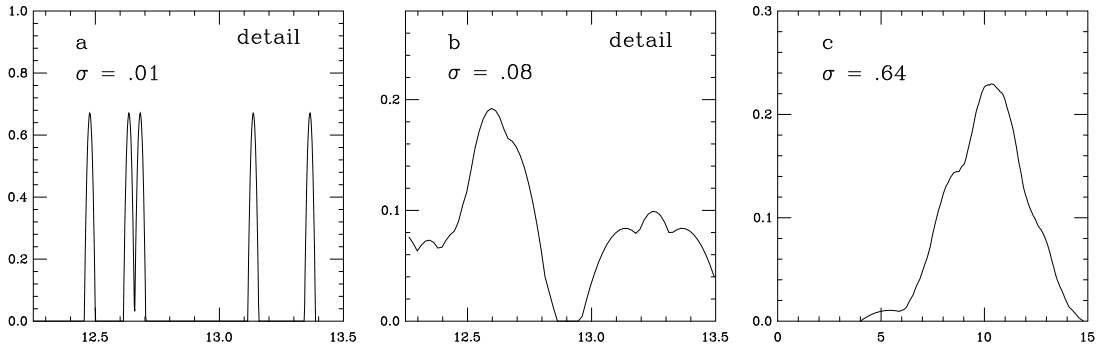


Fig. 2

Sometimes a *non-parametric* estimate of a probability distrubtion is appropriate. If all we want is to display the bars of figure 1 as a smooth function without imposing our own prejudice as to its form, we can replace each of the spikes with a smooth function of limited extent, then add all the smooth functions. Figure 2 (a and b) shows a blowup of part of the data from figure 1 with a narrower and a wider function; figure 2c shows the whole distribution for a function wider still. The non-parametric smoothing shows some shoulders that may or may not be real.[1]

[1] In fact, the data were generated from a Gaussian distribution with a mean of 10 and a standard deviation of 2, so the shoulders aren't real. The estimated mean of 10.1121 falls well inside one "standard deviation of the mean" from the underlying "true" value.

In `kernel.c` we use an inverted parabola cut off where it hits the x axis (see figure 2a), although you may instead choose a Gaussian with cutoffs or any similar curve. The choice and width of the smoothing function can be thought of as parameters, but they do not bias the *form* of our estimated probability distribution. Of course, as the smoothing function gets broader, it widens the estimate. As *Numerical Recipes* comments in introducing section 14.8, one should generally use the original data, rather than a smoothed curve, in fitting a parametric model.

I have found kernel smoothing a useful complement to histograms in exploring the level-spacing statistics of electrons in a wire;[2] I have also used it as a low-pass filter to reduce the spikiness of noisy data.

---

2. The program `kernel.c` smooths an unordered set of data points, such as in `/home/5156/data/stochastic.dat`, writing the `axis`-style two-column files used to create figure 2. It takes at least one command-line argument, the standard deviation of the smoothing function; an additional optional command-line argument tells how many evenly-spaced ordinates to print (the default is 250). For example, figure 2a can be viewed with the command

```
kernel .01 4000 < /home/5156/data/stochastic.dat | axis -x 12.25 13.5 | xplot
```

Your assignment is to fill in the missing parts of `kernel.c` (although as always you are free to rewrite it in the language and style of your choice). **When you are done, the command** `make 2` **defined in the** `Makefile` **will put the program through some tests.**

---

notes on implementing `kernel.c`

Comments in and accompanying the source file describe the algorithms in greater detail. We again call the `histread()` function included last week to read the input into an array. Next—we defer to the next paragraph an optimization—for each of the evenly-spaced x values (outer loop) between the largest and smallest, we consider (inner loop) each input point x_i in the array, adding to the output value at x the value of the smoothing function K evaluated at $x - x_i$.

The function $K(x)$ is zero for $|x| \geq a$, so only points x_i lying within a distance a of x contribute at x. To avoid having to test every input point at every x point, we sort the input array into ascending numerical order: we suggest using the system-supplied `qsort()` function to sort the array in place.[3] This lets us make binary searches of the input array to find the first and last point that will need to be evaluated. The way most people find the right page in a dictionary is by a modified binary search. *Numerical Recipes* discusses the algorithm on page 117 (second C edition); you needn't implement the fancier `hunt()` idea on the following pages. To find an item in a sorted list of length N by binary search takes on average a time that grows with N only as $\log_2 N$, whereas to find it the straightforward way, going through the array until it's found, the average time is $N/2$. If you had a million things, and each step took one second, it would take you almost six days to find your item in an unsorted array, but if the array had already been sorted, a binary search would average only 20 seconds. We still have to sort the array, a step whose time scales as $N \log N$, but we do that only once. If the number of x steps in the outer loop is comparable to N, we win very big with the binary search.

---

[2] See for example Figure 1 in Phys. Rev. B **69**, 054403 (2004).

[3] "In place" means without the need for a second array. If you would like to understand the algorithm used internally by `qsort()`, it is explained in section 4.10 of Kernighan and Ritchie (2nd ed) and section 8.2 of *Numerical Recipes* (2nd C ed). Compare Kernighan's and Ritchie's elegant recursive code to the cumbersome, difficult-to-read subroutine in *Numerical Recipes*. This is proof, if any were needed, that Press *et al.* never really learned the C language, instead just translating literally from the original FORTRAN. Their English descriptions of algorithms, however, are good.

Physics Z-5156, Fall 2006

Chapter 2, Part II: Digital Filters in the Time Domain

Copy the question files and suggested program templates from `/home/5156/public/assignments/filter`*, for example using the following commands:*

```
cd                                              start at your home directory
cp -r /home/5156/public/assignments/filter .    copy the directory to all levels
cd filter                                       begin working in this directory
```

You should respond to short-answer questions in the files e.g., `q1`*, provided. When you have completed everything, the* `make` *command should compile and demonstrate your programs. Feel free to modify the* `Makefile` *if you wish. When you are done, issue the* `make submit` *command to tell the instructor you have finished. Do not change anything in the directory after running* `make submit`*.*

Suppose instead of an unordered set of measurements whose underlying distribution we wish to estimate we have time-dependent data, measured for simplicity at equally-spaced times (a "time series," although in some circumstances the independent variable t may actually be a position, inverse magnetic field, *etc.*):

$$x_0 = x(t_0 = 0),\ x_1 = x(t_1 = \Delta),\ x_2 = x(t_2 = 2\Delta), \ldots, x_N = x(t_N = N\Delta) \quad . \tag{1}$$

Often, we are interested in periodic (sine-wave) or nearly periodic phenomena in the data: absorption or emission peaks, de Haas-van Alphen oscillations in a metal, seismographs. In a week or two, we'll examine neural data from Dr. Kendall Morris's laboratory in the Department of Physiology and Biophysics. Along with such phenomena, unfortunately, the data always include noise. Take a look at the sinusoidal signal with simulated noise in `/home/5156/public/data/noisy.dat`:

```
axis < /home/5156/public/data/noisy.dat | xplot
```

Slow drifts, changing on the order of minutes or hours, from a variety of sources such as temperature differences and instabilities in an amplifier may dominate the signal, while radio-frequency interference may result in rapid random oscillations at frequencies well above that of the signal. Let us concentrate on the latter first. The insertion of a capacitor provides a simple filter for blocking high-frequency noise:

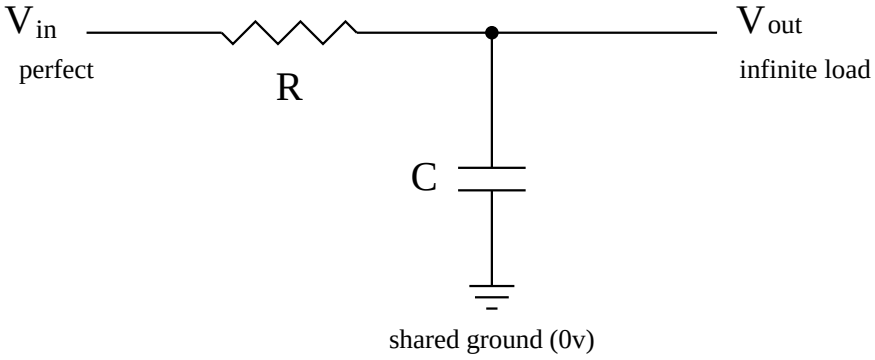


Fig. 1

Recall a first course in waves or electronics, where you found that this filter responds to the input $x(t)$ with a time delay, putting out

$$y(t) = \frac{1}{\tau}\int_{-\infty}^{t} x(t')e^{(t'-t)/\tau}dt' \quad , \tag{2}$$

where the time constant $\tau = RC$. In the particular case of a step function, $x(t) = 0$ for $t < 0$ and $x(t) = A$ for $t \geq 0$, this gives the familiar decay formula

$$y(t) = A(1 - e^{-t/\tau}) \quad \text{for } t \geq 0 \ . \tag{3}$$

In general, if you have the opportunity to use an analog filter for high-frequency noise, take it: as we shall see, its digital counterpart does nothing for frequencies higher than half the sampling rate. Nonetheless, we may have the discretely sampled data x_i and wish to filter them. Our experience with kernel estimation suggests averaging over several recent points to eliminate the rapid variations of high-frequency noise:

$$\text{output} \quad y_i = \sum_{j=0}^{J-1} c_j x_{i-j} \quad . \tag{4}$$

To preserve overall signal strength, ensure that the sum of the coefficients c_j is 1. Note that the output y_i at time i depends only on the current and previous values of the input x: of course, if we want output in real time, we shall have access only to past, not future, data.

Short-answer question 1 (*answer this in the file* `q1`*).* For a finite J, why can we not make the response of this filter arbitrarily close to that of figure 1 and equation (2)? Explain in words, but you may find it helpful, before answering the question, to try a simple case, such as $J = 3$, $c_j = 1/3$, $j = 0 \ldots 2$, by hand on the step function.

This difficulty suggests a solution. In addition to a history of inputs, we can include some feedback from previous *outputs*:

$$y_i = \sum_{j=0}^{J-1} c_j x_{i-j} + \sum_{k=1}^{K} d_k y_{i-k} \quad . \tag{5}$$

More parameters give us greater flexibility but introduce a potential difficulty:

Short-answer question 2 (*answer this in the file* `q2`*).* Why is the choice $c_0 = 1$, $d_1 = 2$, all other coefficients zero, a poor one? Consider for example an input that consists of a small blip at time t_0 and which is zero everywhere else.

Now would be an appropriate time to read *Numerical Recipes* (second C edition) section 13.5 with an eye toward the practical details; while we will fill in some of the theoretical background in the lectures, this is a "laboratory" course meant to give you a hands-on feeling for how to use results. For now, just take equation (13.5.2) as a "recipe" for how a digital infinite-impulse-response filter treats different frequencies: if the modulus $|\mathcal{H}(f)|$ is small, it means that the filter attenuates the frequency f. The phase (or argument) of the complex number $\mathcal{H}(f)$ measures how far the signal is shifted in time.[1]

Step 1: `tdf` Write a digital time-domain filter implementing equation (5). We have provided a suggested template of the main program in `tdf.c` with auxiliary files `share.h`, `share.c`, and `doublecomplex.h`. The command `make tdf` defined in the `Makefile` provided will compile and link these files together into a command called `tdf`. Comments at the beginning of `tdf.c` describe how the program works. At every step you need to keep track of the previous J inputs and the previous K outputs (assume inputs with negative index are all zero). I found this most conveniently implemented with a structure containing an array of some length (J or K) and an index (or pointer)

[1] Appendix A reviews complex numbers.

telling where to put the next element. Functions are in charge of adding a new element and taking the dot product of the array with a vector of coefficients, taking care to wrap around to the zeroth element when the array is about to overflow. Only these functions, and not the functions that call them, need pay attention to where the current index sits. Because to the outside world the whole process is seamless, without indication of how it's implemented internally, I call this kind of structure a circular array.

Actually, there's a cleverer way, although the space savings is negligible: instead of storing the previous J inputs and previous K outputs, save only L numbers in an array `save[]`, where L is the larger of J and K. At each step, move all the elements of an array one step forward (closer to 0). Increment `save[0]` by c_0 times the current input x; it will be your new output. Increment `save[1]` by $c_1x + d_1y$, `save[2]` by $c_2x + d_2y$, *etc.* By the time the points work their way up to index 0, they have everything you need. Feel free to use either this method, the straightforward one with two "circular" arrays, or some combination.

The "shared" files take care of argument parsing, which will be nearly the same for this program and for `tdf-analyze` below. Invoking `tdf`, the user provides the coefficients c_j, preceded by a `-f` for "finite," and d_k, preceded by `-i`, on the command line. The input is a single column of numbers representing the time series x_i, corresponding to times 0, 1, 2, *etc.* (If your input has multiple columns, you can use the `extract` program you wrote in the first week or the filter command `awk '{print $n}'` to extract just the x_i.) Output are the values y_i from (5). It is not necessary to flush the stored values at the end; usually the larger of J and K is still very much smaller than the number of data points, and we don't worry about edge effects.

Short-answer question 3 (*answer this in the file* `q3`*).* If J and K are fixed and there are N data points, what is the order of the computation? What if J and K are set to fixed fractions (*e.g.*, 1%) of N? *The order of a computation tells how the number of machine operations to be performed scales with an increasing size of the input, N. For example, $\mathcal{O}(N^3)$ means that a problem with $N = 2N_0$ takes eight times as long as one with $N = N_0$. The order ignores the prefactor describing how long a single operation takes: making an algorithm twice or even a hundred times as fast doesn't change its order. For large problems, a reduction in the order of a problem can be infinitely more important than any speedup in the prefactor, as we shall see next week with the fast Fourier transform.*

Step 2. To test your program initially, generate (with `awk`, an editor, or a C program) a file `square.dat` of data containing a square pulse

$$x_i = \begin{cases} 0 & 0 \le i < 50 \\ 1.0 & 50 \le i < 100 \\ 0 & 100 \le i < 150 \end{cases} \tag{6}$$

The command `make squaretest1` defined by the `Makefile` provided will run a few simple tests for finite and recursive filters. Verify that they do what you expect. (Implementation note: the `-a` flag in `axis -a` tells `axis` to provide abscissas 0, 1, 2, ... automatically.)

Step 3. Equation (13.5.2) in *Numerical Recipes* gives the frequency response of the filter (5). Use the program `tdf-analyze` (provided) to evaluate (13.5.2). You may use the template we have provided in `tdf-analyze.c`. We've also provided `mathematica`, `maple`, and `matlab` versions. The same parsing routines are used as for `tdf`, and we have provided complex arithmetic functions in `doublecomplex.h`. We have included a script `zaxis` that is useful for looking at the output of `tdf-analyze`.

Skim sections 12.0 and 12.1 of *Numerical Recipes* to understand why why `tdf-analyze` plots only frequencies between 0 and $1/(2\Delta)$.

Short-answer question 4 (*answer this in the file* `q4`*).* By hand or with a program, plot a pure sine wave $\sin(2\pi t/T)$ of period T. Let $T = (8/5)\Delta$, so that you are sampling (every Δ) fewer times than twice per period. You needn't turn in your graph. What is the apparent and wrong period you see in your discrete data? Do you now understand why `tdf-analyze` looks at frequencies only up to $1/2\Delta$?

Let's design a digital filter to mimic the analog from figure 1. The impedance of the resistor is the real number R (measured in Ohms), while the impedance of the capacitor, $-\frac{i}{2\pi f C}$, depends on the frequency f and is purely imaginary. (Don't worry if you don't follow this: it makes sense, though, that the real part of its impedance should vanish, since a good capacitor doesn't dissipate any heat: it just pushes the voltage and current out of synchrony.)

The output voltage is the impedance of the capacitor times the current going through it, while the current going through both the resistor and the capacitor (we assume the load is very large, so that none leaks through the output terminal) is the input voltage divided by the sum of the resistance and capacitive impedances. A little algebra (you're not required to do this) establishes that

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{1}{2\pi i \tau f + 1} \quad , \tag{7}$$

where again $\tau = RC$. This ratio is the frequency response $\mathcal{H}(f)$ for the analog filter.

Can a digital filter with just two parameters, c_0 and d_1, give a comparable response? For reasons already covered, we assign no meaning to what happens for $f > 1/2$ (in units of $1/\Delta$), and probably expect best results for $f \ll 1/2$. Making this approximation and setting all other c's and d's to zero, we truncate a Taylor expansion in f for (13.5.2):

$$\mathcal{H}(f) = \frac{c_0}{1 - d_1(1 - 2\pi i f \Delta)} \quad . \tag{8}$$

Step 4. Comparing equations (8) and (7), determine the coefficients c_0 and d_1 in terms of the dimensionless parameter $\tilde{\tau} = RC/\Delta$, the time constant in units of the sampling interval. This is just a little bit of algebra; see an instructor for help if needed. Modify (or rewrite using any tool you prefer) the awk script `rc`, which takes a single argument specifying the time constant $\tilde{\tau}$ and prints out command-line flags suitable for `tdf` and `tdf-analyze`. Here's how to use the output of `rc`, with a time constant of 4Δ, as the command line of `tdf`:

```
tdf `rc 4` < input | axis -a | xplot
```

Note the backquotes (*accents graves*) rather than ordinary forward quotes. The command `make squaretest2` defined in the `Makefile` will run `tdf` with a few sample time constants.

Step 5. Using the tools of your choice (for example `tdf-analyze`, `mathematica`, `awk`), write a script `compare` taking no arguments which, when run, plots a comparison of the desired RC filter frequency response (just the modulus) with that of the digital filter. Use labels and dotted lines or colors so we can tell which is which. If you decide to use `axis`, the documentation given by

```
xdvi /home/5156/public/doc/axisdoc.dvi
```

should help.

Short-answer question 5 (*answer this in the file* `q5`*).* For a time constant $\tilde{\tau} = 4$, compare numerically the observed decay of a sharp edge in your digital filter with equation (3). Based on this comparison and step 5 above, how well would you say the two-parameter filter fits the capacitive circuit? Now try (mathematically or by trial and error) to make a two-parameter finite-impulse-response filter (all the d_k's zero, two of the c_j's non-zero) that mimics the analog circuit. Can you do as well?

Finally, consider the four-parameter bilinear recursive bandpass filter given by equation (13. 5.13): this smoothly cuts off frequencies below Arctan $a/\pi\Delta$ as well as those above Arctan $b/\pi\Delta$.

In preparation for next week's assignment, you may find it helpful to write a short script like `rc` but with arguments for the lower and upper cutoff frequencies. The directory `/home/5156/public/data` contains several data sets whose behavior you may wish to examine with your filter. You needn't turn in any work related to this.

Appendix A. Complex Numbers

Recall that for a complex number $z = re^{i\theta} = r(\cos\theta + i\sin\theta) = a + ib$, mod$z = |z| = r = \sqrt{z^* z} = \sqrt{a^2 + b^2}$ and $\arg z = \theta = \text{Arctan}\, b/a$, where z^* denotes the complex conjugate, calculated by replacing $i = \sqrt{-1}$ with $-i$ everywhere it appears in z. Complex conjugation preserves the modulus but multiplies the argument by -1.

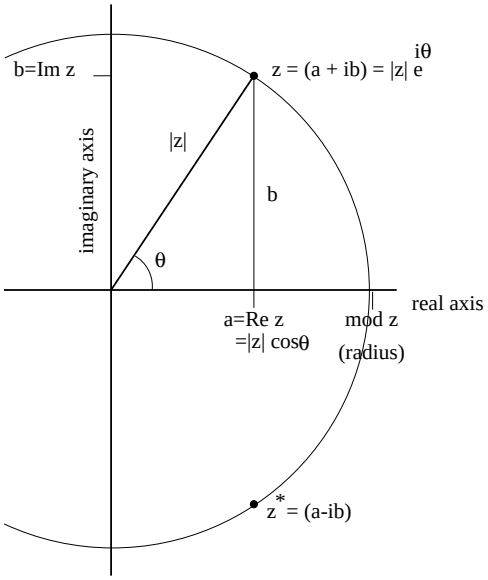


Fig. 2

The real number a is called the real part (Re z) of z, b the imaginary part (Im z). As suggested by the Arctan above, one may form a picture of a complex number in the a-b plane, with θ the angle going counterclockwise from the real (a) axis to the vector from the origin to z.

We shall have occasion to use de Moivre's theorem, which you may easily establish by Taylor expanding both sides of the following expression and showing that they are equal, term by term in θ:

$$e^{i\theta} = \cos\theta + i\sin\theta \quad . \tag{9}$$

This equation lets us draw the trigonometric picture in figure 2. It also lets us interpret multiplying z_1 by z_2 as multiplying the two moduli and then rotating z_1's argument by z_2's. As an example, the complex number $1 = 1 + 0i$ is a point on the real axis. If we multiply it by z in the figure, we get a point a radius $|z|$ from the origin whose vector makes an angle θ with respect to the positive real axis (in other words, just z).

The text refers to "poles." These are very handy in complex analysis, but all we need to know is the definition. A function $f(z)$ is said to have a simple pole at z_0 if $f(z_0)$ blows up but $(z - z_0)f(z_0)$ does not. For example,

$$f(z) = \frac{1}{z^2 - a^2} \tag{10}$$

has simple poles at $z = \pm a$.

Complex numbers come into physics in a variety of ways. For linear electrical circuits, we use them for convenience. Consider a capacitor with an alternating current,

$$I = I_0 \cos(2\pi f t) \tag{11}$$

going through it. We know that the voltage across will lag the current by 90°,

$$V = \frac{1}{2\pi f C} I_0 \cos(2\pi f t - \pi/2) = \frac{1}{2\pi f C} I_0 \sin(2\pi f t) \quad . \tag{12}$$

We can, although we don't have to, represent the current and voltage as the real parts of complex currents and voltages:

$$\begin{aligned} I(t) &= \mathrm{Re}(\tilde{I} e^{2\pi i f t}) \\ V(t) &= \mathrm{Re}(\tilde{V} e^{2\pi i f t}) \\ \tilde{I} &= I_0 \\ \tilde{V} &= -\frac{i}{2\pi f C} I_0 \quad . \end{aligned} \tag{13}$$

Here is the beauty of complex numbers. The ratio V/I of the voltage across a resistor to the current through it is just a number, the resistance. The ratio V/I for a capacitor, however, has a tangent in the numerator and a frequency in the denominator. The tangent itself depends on frequency. However, the ratio $Z = \tilde{V}/\tilde{I} = -i/2\pi f C$ is just a negative imaginary number, called the complex impedance. Inductors have positive imaginary impedance. You can verify that any alternating-current (A.C.) circuit whose only components are resistors, capacitors, and inductors acts like a direct-current (D.C.) circuit with resistors with real resistance and capacitors and inductors with their imaginary resistances. In the D.C. circuit, the currents and voltages $\tilde{I}$ and $\tilde{V}$ may be complex but remain constant in time. To get the true currents and voltages, we multiply by $e^{2\pi i f t}$, then take the real part.

We can represent electrical circuits and mechanical oscillators with complex numbers so long as they are linear (in voltages, currents, or displacements). The prescription breaks down if we need any non-linear quantities, for example the square of the voltage. We believe that quantum mechanics is fundamentally a linear theory at the microscopic level: furthermore, the requirement of conservation of probability *requires* that a quantum wavefunction should be a complex quantity. In quantum mechanics, complex numbers are not just a convenience: the world, apparently, really is that way.

One typically represents a complex number $a + ib$ on a computer by storing its real and imaginary parts contiguously, although other conventions are sometimes useful. The C language does not have a built-in complex data type, but it is fairly easy to implement a complex `struct`, as discussed by *Numerical Recipes* (second C edition) in sections 1.2 (pages 23–24) and 5.4 and in Appendix B. Unfortunately, C provides no way to tell binary operators like `+` and `*` what to do with complex numbers on either side or both. The `C++` language does, and some of you may decide eventually to learn `C++`.

Physics Z-5156, 2006

Chapter 3: Working in the Frequency Domain — the fast Fourier transform

Copy the question files and suggested program templates from `/home/5156/public/assignments/fft`

When you have completed everything, the `make` *command should compile and demonstrate your programs. Feel free to modify the* `Makefile` *if you wish. When you are done, issue the* `make submit` *command to tell the instructor you have finished. Do not change anything in the directory after running* `make submit`.

Last week we designed a filter to smooth away rapid oscillations in data and showed that the resulting program was equivalent to a low-pass filter: it let low frequencies through and blocked high frequencies (the rapid oscillations). In fact, any signal can be described as a sum of components with different frequencies, each carrying an appropriate coefficient. If $h(t)$ represents the signal as a function of time, we write (see Numerical Recipes 12.0.1, 2nd ed.)

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt \\ h(t) &= \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} df \quad . \end{aligned} \tag{12.0.1}$$

The two descriptions of the data, $h(t)$ in the time domain and $H(f)$ in the frequency domain, can be interconverted and so are evidently equivalent.[1] Section 12.0 reviews the Fourier transform (12.0.1) and two of its theoretically important applications, correlation and convolution. We shall have occasion later, in the context of a quantum spin chain, to look at the autocorrelation of a signal, $G(\xi) = \int g^*(x+\xi) g(x) dx$, which measures the degree of spatial order in g. The convolution will come up this week.

Section 12.1 describes how to approximate (12.0.1) on a digital computer, which handles neither truly continuous data sets nor infinite ones. Note that the finite discrete Fourier transform is not quite the same thing as the infinite Fourier *series* you may have studied in other contexts.

The implementation of the transform

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N} \tag{12.1.7}$$

would appear rather slow. Here it is in pseudo-code:

```
for(n=0; n<N; n++) {                          /* N outer iterations */
        H[n] = 0;
        for(k=0; k<N; k++)                    /* N inner iterations */
                H[n] += h[k] exp(2π i k n /N);
}
```

[1] We follow *Numerical Recipes* in expressing frequencies in Hertz (conventionally f) rather than in radians per second (ω). Each convention has advantages; f is probably simpler for numerical work. Incidentally, the factor $\exp(\pm 2\pi i f t)$ in equation (12.0.1) generalizes the notion of a kernel, although unlike our smoothing kernels it is infinite in extent.

We'll call this algorithm the *slow Fourier transform*. The time needed to calculate all the coefficients scales as N^2. If N is a million, this is serious. Section 12.1 describes how to compute the same thing without any approximations in a time that scales only as $N \log N$. As a bonus, the fast Fourier transform works entirely in place: at the end of the day, `h` has been replaced by `H` without the need for any additional storage. While I will not ask you to implement your own fast-Fourier-transform subroutine, you should at least skim this section just to get an idea of how it works its magic. Typical of code in *Numerical Recipes*, the subroutine `four1()` starts its arrays at 1 instead of at 0, which is particularly perverse in the case of the Fourier transform, since it then represents H_0 by `H[1]`.[2]

The first half of section 12.3 shows how to perform two Fourier transforms of real (not complex) data simultaneously. You may then skip to read sections 13.4, 13.1, and 13.3.

---

1. Is it really all that fast?

This exercise is just a warmup; don't spend too much time on it. I have provided you with a public-domain fast-Fourier-transform subroutine `fft_()` in the file `singleton8.f`. Don't worry that it's written in Ancient Greek; you can treat it as a black box.[3] The `Makefile` includes all the necessary libraries, and I have provided a complete program, `test-fft`, that shows how to call it from C. Note that while arrays are passed exactly as to a C subroutine, FORTRAN always expects to receive pointers to scalar variables such as `int`s, never the values themselves. For our current purposes, the parameters `NTOT`, `N`, and `NSPAN` should all be pointers to a single variable that tells how long the arrays `A` and `B` are. Although `fft_()` knows how to deal with some N other than powers of 2, we shall not take advantage of this.[4] Whereas `four1()` alternates the real and imaginary parts of the complex signal h in a single array, `fft_()` uses two separate arrays. Both storage schemes have merit. Having the two parts separate provides a small advantage when transforming two real arrays simultaneously.

Short programming and essay. The file `test-sft.c` is almost identical to `test-fft.c` except that it calls a subroutine `sft()` in `sft.c`. Write a slow Fourier transform, then verify that `test-sft` and `test-fft` give identical numbers (for which the `diff(1)` utility may come in handy). You needn't be very fancy. For your own peace of mind, verify too that `test-fft < data | test-fft -r` reproduces the original data. Both test programs write timing information to `stderr`, which is *not* normally redirected to a file when you use `> output` but rather displayed on the screen. The command `rand` n will write n pseudo-random numbers between 0 and 1. Use these as input for the two test programs and create `axis` input files plotting n (powers of two) versus execution time.[5] You needn't collect detailed statistics: the difference between the two algorithms will not be subtle. Save the input files with the names `q1slow.axis` and `q1fast.axis`, or if you prefer, change the `Makefile`'s treatment of the command `make 1`. Comment in one paragraph in the file `q1` on what happens in each case when the input length doubles.

[2] I would add that `four1()` should have more comments, explaining the trigonometric recurrence (for example they calculate $\cos\theta - 1$ as $-2\sin^2(\theta/2)$ because the former expression loses precision for small θ) and the various loops. Their variables should either have more expressive names than "m," or else their purpose should be noted.

[3] Singleton is relatively compact for an optimized implementation of the fast Fourier transform, but modern versions incorporating various tricks run faster. Probably the best current code is called `fftw` for "Fastest Fourier Transform in the West." Where `singleton8` takes up around 500 lines of code, the `fftw` package uses around 33,000. In later projects, you may link to `fftw`.

[4] Notice the error message it prints out if N has too many distinct prime factors or if one of the prime factors is too big. FORTRAN has no facility like C's `malloc()` for allocating memory as needed, so it runs out. No problem arises, however, with powers of 2.

[5] The test program counts CPU time in some fraction of a second only for the Fourier subrou-

2. Power spectrum

The frequency (or in the spatial case wavenumber) at which something happens (*e.g.*, absorption, emission, reflection) often has physical meaning, and experimenters work very hard to discover it. Intrinsically sharp (δ-function) peaks can be broadened by thermal effects, phase jitter, instrumental resolution, and, as discussed in *Numerical Recipes* section 13.4, the mere fact of digital sampling (measuring the data only at particular times rather than continuously). Electrical and other sources of noise further mask the signal of interest, and often a strong, unintersting peak can make it hard to find a nearby weak one.

Depending on time constraints, I may supply the `gendata` *program.* Before analyzing real data, let's generate some. That way we have control of the noise. The partly-completed `gendata` program plots 4000 points from three superimposed "signal" sine waves, one two-hundred-kilo gorilla signal that we don't care about, and Gaussian white noise. Chapter seven of *Numerical Recipes* covers the generation of "random" (meaning apparently uncorrelated) numbers, but for the limited purpose here, the system-supplied `random(3)` (not `rand(3)`) subroutine does an adequate job of generating uniform deviates. The file `gaussd.c` converts the uniform deviates into Gaussian following equation (7.2.10). A table in `gendata.c` provides a fixed list of frequencies and amplitudes. After making the `gendata` program, create a data file with the random seed 42 using the command

`gendata 42 > gd.42` .

You can examine the data visually by

`axis -x` *startx endx* `< gd.42| xplot` ,

but it's likely you'll be able to make better sense of them in the frequency domain.

Numerical Recipes supplies a spectrum-analysis subroutine at the end of section 13.4, but I'd like you to implement your own using the `fft_()` subroutine in `singleton8.f`. (Since Singleton uses two separate arrays, you'll not be able to copy *Numerical Recipes* blindly.) Comments at the beginning of `power.c` describe how it works, and I have filled in some pieces, although as always you are free to do things differently. Note the way our `pick()` subroutine finds the user-selected window function from the table `wintab[]` and how `apply_windows()` multiplies two arrays by the selected function. The typedef `ftype` makes it easier to declare a variable that points to a window function; the statements

```
typedef double (*ftype)(int, double);   /* defines ftype */
double win_Welch(int, double);          /* declares win_Welch */
ftype window = win_Welch;               /* assigns a value to window */

x = (*window)(arg1, arg2);              /* calls win_Welch() */
```

are equivalent to

`x = win_Welch(arg1, arg2);` .

The `power` program (at least as we've implemented it) expects only a single column of numbers, not the two that `gendata` puts out. Use your `extract` command or a pipeline with `awk '{print $2}'` to pick out the relevant data. `Power` takes two optional arguments, which if both are present can come in either order. One is the M parameter discussed in the text. It should be a power of two; the data will be transformed in half-overlapping overlapping sequences of length $2M$. If you omit M, all the data will be transformed at once. The other parameter is one of the names `square`, `Bartlett`, `Welch`, and `Hann` to specify the window function. The default is `square`.

Write the `power` program, using our template if you wish. Do not just copy `spctrm()` from *Numerical Recipes*, although you may consult it to understand the algorithm. (Note that k in `spctrm()` is half the K in the text.)

tines; you could use the shell's `time` feature instead, but that would include time spent on reading and writing as well.

The output of `power` will need to be cut off if you want to see anything other than the monster peak near frequency 0.4; I've included a little shell script called `process` as a model. By trial and error, you should be able to come up with a combination of M and window function that lets you resolve all three of the signal peaks (it doesn't take much fiddling to see the big one). Of course, here you have the advantage of knowing in advance where they're supposed to be: a real experiment doesn't afford such luxury. You will probably note that the signal peaks aren't so easy to separate from noise. The text describes statistical techniques for testing whether a feature is real or not; you needn't use them here, but your pictures should convince you that they could be important in research.

Write a short (two- or three-paragraph) report on your power spectrum in the file `q2`. Describe the results of your hunt for the best window and frequency resolution (as specified by M). Estimate, roughly, the areas under the three peaks (if you can find them). First subtract the baseline in the vicinity of the peak (you can determine the noise level by hand or with your `tdf` program). Then determine (approximately) the full width of the peak at half the maximum signal level. Assuming a Gaussian, this width is about 1.18 times twice the standard deviation, giving an area $\approx 1.06Aw$, where A is the maximum amplitude and w is the full width at half amplitude. (You don't have to work out the math.) How well do the ratios of areas agree with the squares of the amplitudes given in `table[]` in `gendata.c`?

3. Correcting for a known instrumental resolution function

I will announce in class whether there is time to attempt this part. In our previous example, we found plenty of broadening just from leakage; we countered it by increasing the frequency resolution. A spectrometer, however, will always register a smooth (less than perfectly sharp) signal, no matter how carefully we sample its output. In effect, it acts like one of the kernel-smoothing filters we made last week. To put it another way, the signal we see is the convolution of the true signal $u(t)$ and an instrumental resolution kernel, $r(t)$:

$$s(t) = \int_{-\infty}^{\infty} r(t-\tau)u(\tau)d\tau \quad . \tag{13.3.1}$$

All the nastiness of part 2 comes on top of this. Section 13.3 describes how to design an optimal filter to apply to the deconvolution algorithm of section 13.1.

The file `/home/5156/public/data/spectrum.dat` contains real experimental data (I will define its nature further in class). One peak is well isolated, and its shape therefore is exactly the instrumental resolution function $r(t)$. Two other peaks are too close to resolve reliably. Modifying the programs you have written or modifying `convlv()` in *Numerical Recipes*, try to deconvolve the two peaks. Store the resulting spectrum (in `axis`-input format) in the file `deconv.dat`.

Physics Z-5156, Fall 2006

Chapter 4: The Spike-Sorting Problem in Neurobiology

Copy the question files and suggested program templates from `/home/5156/assignments/-neuro`*, for example using the following commands:*

```
cd                                      start at your home directory
cp -r /home/5156/assignments/neuro .    copy the directory to all levels
cd neuro                                begin working in this directory
ln -s /home/5156/data/b01c1b.bin .      create a symbolic link to the raw data
```

For this project, I would like a short report with appropriate accompanying figures, graphs, and tables. The main point of the project is to answer as well as you can two questions, as explained in more detail below: (1) how many distinct neurons has the probe picked up; and (2) for each such neuron, what is the sequence of firing times? Once you have come to your conclusions, I would like you to compare your results to those of one or more classmates. You should comment on any differences. I don't expect certainty and will not be grading on correctness; there may not be time in two weeks to accomplish a detailed statistical analysis establishing that your result is the unique correct answer. Spike sorting remains an open problem in statistical biology and the subject of much current research.

Please write in plain text (a file you make with `vi` or `emacs`), `troff`, TEX, or Latex. Please do not use HTML, SGML, or Microsoft "quoted printable" formats, as they are difficult to read. I cannot read Microsoft Word at all, but Postscript generated by Word is acceptable. When you submit your results, please make clear with a `README` file which files in the directory contain your report and programs. When you are done, issue the `make submit` command to tell me you have finished. Do not change anything in the directory after running `make submit`.

Some biological and electrochemical background

Dr. Kendall Morris, USF Department of Physiology and Biophysics, is interested in understanding the mechanism by which the vertebrate brain stem controls respiration (breathing).[1] Understanding this relatively simple part of the brain is a first step to interpreting the complex spatio-temporal patterns characteristic of higher functions.[2]

Using the cat as a model, Dr. Morris implants 72 electrodes in the brain stem. Since the electrodes are extracellular, each picks up signals from several neurons at once. The very first task is to disentangle the signals, replacing the time series of voltage measurements on each electrode with one time series for each neuron. The next step would be to look for statistically significant correlations among neurons and between neurons and the respiratory cycle. The emerging patterns can then be compared to physical theories of collective behavior, such as neural networks (which are related to spin glasses) and dynamical systems. These models come from the field of statistical mechanics, and many of the researchers in the field have backgrounds in physics and statistics.

A long axon connects a neuron's cell body to a distant synapse. Experiments in the first half of the twentieth century using intracellular probes in the giant-squid axon established a simple picture of how neurons conduct electrical signals.[3] Normally, active molecular pumps maintain a

[1] K.F. Morris, R. Shannon, B.G. Lindsey, "Changes in cat medullary neurone firing rates and synchrony following induction of respiratory long-term facilitation," *J. Physiol.* **532** (2001) 483.

[2] For an example of such patterns, see J. Prechtl *et al.*, "Visual stimuli induce waves of electrical activity in turtle cortex," *Proc. Nat. Acad. Sci. USA* **94** (1997) 7621.

[3] See, *e.g.*, Nicholls, Martin, Wallace, *From Neuron to Brain*, 3rd ed., Sinauer, 1992.

deficit of Na^+ inside the cell relative to the exterior, resulting in a net negative potential inside of approximately 80mV. The cell membrane can be modeled as a capacitive-resistive-battery network, supporting the propagation of a depolarizing pulse. As such a pulse approaches a segment of membrane, reducing the negative polarization, a threshold is reached at which sodium channels open, briefly resulting in a large net positive charge inside the axon. Other molecular mechanisms (Ca^{++}, K^+, organic ions, and Cl^- all play important roles) then quickly repolarize the segment, sometimes overshooting the rest potential, and usually a refractory period follows during which the membrane slowly recovers its resting state and cannot fire. The traveling pulse is called an *action potential*, and normally (absent experimental probes such as drugs and applied voltages) is an all-or-nothing, invariant response triggered wherever a depolarization exceeds some threshold.

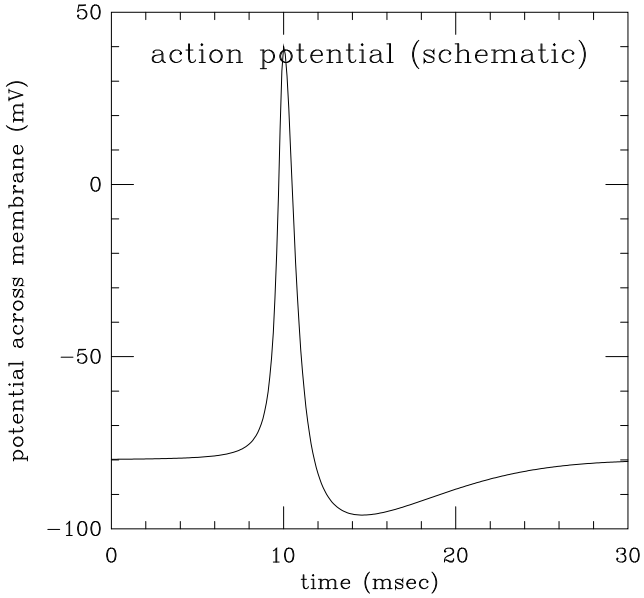


Figure 1. Schematic representation of a typical action potential (adapted from Nichols, Martin, Wallace, *op. cit.*).

We take one important feature from this model: for a given type of neuron, every spike is approximately the same (in duration, amplitude, and shape) as every other spike, so the only way for a neuron to distinguish the strength of a stimulus is to fire at a greater or lesser rate. (This may not be strictly true of certain types of motor bursters; I do not know if we have them in our data set. Dispersion may also change the shape with distance, but this does not alter the conclusion: information in a single neuron can normally be encoded only in the frequency of action potentials.)

Unfortunately, few axons are so large as those of the giant squid or take so well to having glass tubes poked in them. The signals in Dr. Morris's data, taken outside the cells, in the largely conducting saline intracellular fluid, are thus strongly attenuated and possibly dispersed echoes of the intracellular spike. The researcher needs to amplify the signals significantly. (The difficulty of these experiments is another reason that so many of the people doing them have a physics background.) If we assume that every spike is intrinsically the same as every other, we can interpret different measured amplitudes and shapes in the data stream as originating from different neurons different distances from the electrode. This provides the basis for spike sorting.

The data

The file `b01c1b.bin` holds a short (one-minute) excerpt from one electrode. The data were sampled at a rate of 24 KHz from an A.C.-coupled analog amplifier, meaning that what we see is effectively the derivative of the voltage. The amplifier stage uses an analog notch filter to cut

out 60-Hz noise from the electronics, but persistent problems with ground loops (we are, after all, trying to measure voltage differences inside a saline solution) have left sharp components at all odd harmonics of 60 Hz. Further noise may come from the amplifier and from neurons too distant to resolve. There is also a strong electro-cardiogram at around 4 Hz.

When I received the file `b01c1b.bin`, I first had to determine the format. Commands like `od -f b01c1b.bin | more` established the format was unlikely to be floating point, so I tried `od -x` and `od -t xL`, the former looking at two-byte integer shorts, the latter at four-byte longs, in hexadecimal. Interpreted as shorts, the data seemed to change slowly (note that negative numbers roll over backward, putting the first hexadecimal digit in the range `8`–`F`, so that -2 for example would be `FFFE`).[4] The best news was that there didn't appear to be a header, a block of information preceding the data that might be of unknown length and undocumented format.

I then had to decide whether to work with the data in binary or ASCII form. The former is much more efficient, but the latter lets one see what one's doing. In either case, our graphing programs eventually require ASCII.

Once you have written the little conversion program—you might call it `b2a`, for binary-to-ASCII—you should start looking at the data. The time series runs to 1.4 million words (shorts), which is not so large that we can't look at it all at once:

```
b2a < *bin | axis -a | xplot
```

The `-a` switch tells `axis` automatically to supply abscissas to the single column of ordinates in the output of `b2a`. I then recommend looking at more manageable pieces of the data; this is where it's useful to make a working copy of the ASCII representation:

```
% mkdir /scratch/yourname                                  your space on scratch disk
% b2a < *bin > /scratch/yourname/data.ascii                write big file on scratch disk
% ln -s !$ .                                               make symbolic link here
% head -2200 < *ascii | tail -300 | axis -a | xplot       look at points 1901–2200
```

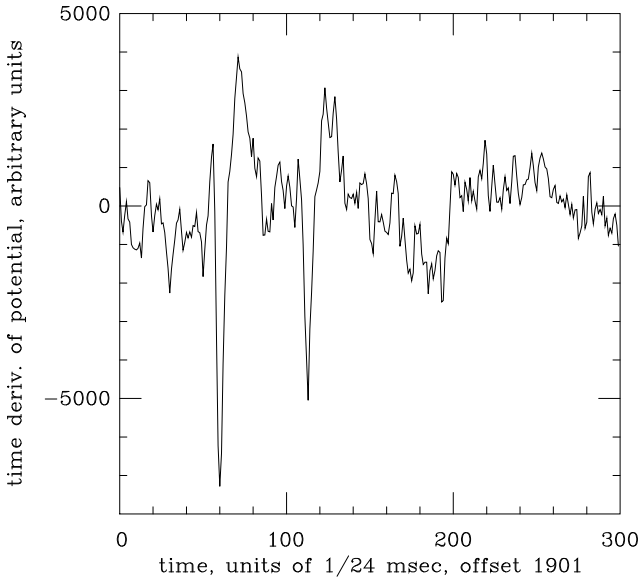


Figure 2. An excerpt from the raw data (`head -2200 < *ascii | tail -300 | axis -a ...`)

[4] I started looking at these numbers on a Sun, where the byte order is reversed. With `od -x`, it was evident that the least significant byte (two hexidecimal digits) was changing far more slowly than the most significant. I then had to write a little ten-line program to reverse the byte order. Sun Sparcs are called "big-endian," because the most significant byte comes first, while Intels are "little-endian." Compare the output of `od -tx2` with that of `od -tx1`.

The segment of this particular example contains two obvious spikes and a representative sample of noise. The lower amplitude of the second spike and the short interval (2.5 msec) after the previous both suggest, but do not prove, that the spikes came from different neurons. Alternatively, we could be looking at a burster, or the differences could be due simply to noise. We could try to rule out the latter possibility on statistical grounds.

Filtering

One kind of filtering we should do before anything else: subtract the mean value from the data. Because the amplifier is AC coupled, this mean value should be small.

We'd eventually like to look at autocorrelations of the subseries for each neuron we find; see *Numerical Recipes*, chapter 13. If we did so right now, we'd find a series of sharp spikes, sometimes a good omen, but not here: they're at the odd harmonics greater than the fundamental of 60 Hz line noise and completely drown out anything of possible interest.

Short-answer question 1 (*answer this in the file* `q1`*).* I inferred a fundamental somewhat different from 60 Hz, which could be due to fluctuation in the electric utility or to a small difference from 24 KHz in the digitization rate. Using your power-spectrum software from last week, estimate the number that should be used instead of 60 Hz. Quote an appropriate number of significant figures. I found the peaks with a little gawk script, then (by hand, but I could have used a script) interpolated the true peak positions using the highest point and one to either side.

I wrote a notch filter that cuts out a triangular piece of frequency space centered at all the odd multiples of "60" Hz, making the base of the triangular notch a command-line argument. The procedure is simple: FFT the data, multiply by a function $H(f)$ that is 1.0 except at the triangular notches, then FFT back. I also implemented overlapping segments of variable length, as discussed last week in the context of spectral estimation. When you implement this, pick the notches relatively narrow, or you will change the heights and shapes of your spikes. (To make sure you're not doing so, plot selections from the raw and filtered data together). Since spikes are quite narrow in time, they are very spread out in frequency, meaning that they're unusually sensitive to any kind of Fourier filtering.

Getting rid of the harmonic noise is a boon for the autocorrelation, but it seems not to do much for figure 2 and may not be necessary for spike sorting. I'll defer further consideration of filtering.

Triggering

To sort spikes, we first must find them. Using the software you wrote in the first week, make a histogram of the measurements (or of their absolute values; either is informative). You could start with the raw data (after subtracting the mean), with the harmonic-notch-filtered data, or with some other filtered set (such as the wavelet-filtered data I describe later). If your filtering has not damaged the signal, it shouldn't make much of a difference (but I could be wrong here).

There are obviously fewer measurements the further we go from zero. We might expect pure noise to have a Gaussian distribution; any systematic deviation in the tails indicates a signal we have some hope of finding. To estimate whether or not the histogram looks Gaussian, I instructed `axis` to print on a logarithmic scale; noise would then be an inverted parabola. Instead, I found figure 3:

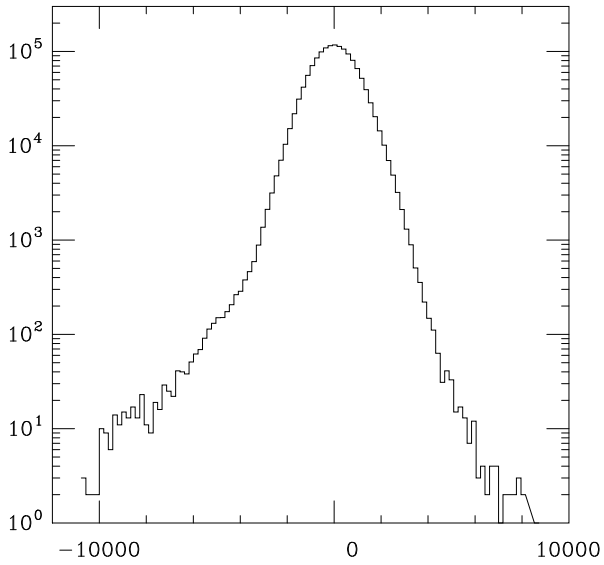


Figure 3. Output of `histbin 200 < *ascii | hist2axis | gawk '$2>0' | axis -y l ...`

The tails, particularly on the negative end, are unmistakable. So we look for all signals more negative than some multiple α of the standard deviation (which we compute using last week's `dist` script). The one-line `gawk` program "`gawk '$1<-`*cutoff*`{print NR}'`" accomplishes the selection. (`NR` in `gawk` is the record, *i.e.*, line number. Counting starts at 1, unlike in C.) You can set α graphically from figure 3 or by asking how many events you would *expect* that many standard deviations out, assuming it were all noise.[5] Pick an α such that the number expected is enormously fewer than the number observed: then almost all the observed peaks are probably real. One needs to do a little better than the `gawk` line above, since it will trigger several times for each negative spike: I wrote a short little `gawk` script that does essentially the same thing but considers only local minima.

In fact, we should be able to extract spikes even if they're in the noise, because we can select not just a single time but the coincidence, say, of a positive peak followed by a negative followed by a positive, all in a time typical of a spike (see figure 2). I leave this for you to pursue.

Once I've triggered, I need to be able to pull out each spike. I did this in two steps: first, I used the `gawk` script described above to select the times (NR's) at which there is a local minimum more negative than the cutoff, $-\alpha$ times the standard deviation, putting the results in a file. Then I wrote a C program to go through that file and extract the context of each spike, which I defined (heuristically) as 50 points, starting 19 time steps before the trigger. I wrote the program in C rather than `gawk` because it's a little tricky to handle overlapping snippets. The program writes each 50-point snippet in a separate file sitting in a directory. I provide a "multiplot" shell script that lets me superimpose an arbitrary number of snippets, the result of which appears in figure 4. There's a single off-scale peak appearing twice (because it's in overlapping snippets), which I've suppressed.

[5] Useful references: Taylor, *An Introduction to Error Analysis*, University Science Books, 1982 (the train-wreck book); Keeping, *Introduction to Statistical Inference*, van Nostrand, 1962; rpt. Dover, 1995. The standard math library, `-lm`, contains an `erf()` function; several libraries also contain `erf_()` functions. Since calling conventions may differ, be careful of the order in which you link the libraries.

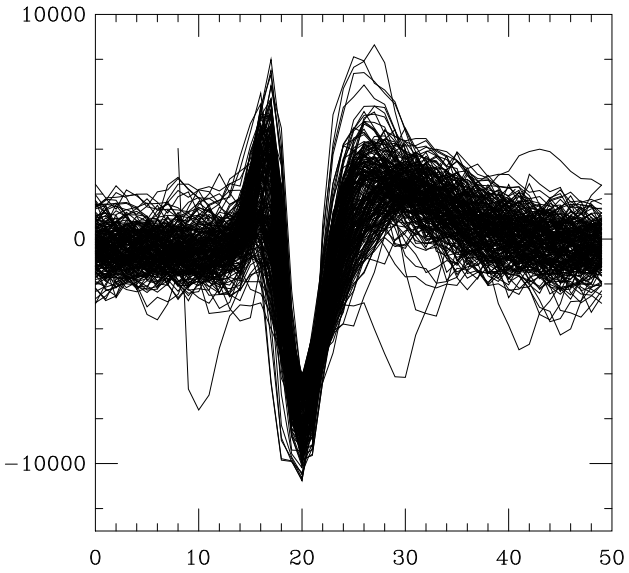


Figure 4. Many superimposed spike candidates.

I may have been too conservative in my α: all the triggers appear to have found spikes, which means I've probably missed some.

Sorting

The next step is to find some set of criteria on the basis of which to sort. One approach (employed by some commercial spike-sorting programs) is to pick eight or so heuristic parameters, such as depth of the negative peak and heights of the flanking positives. Each spike is then reduced to a single point in an eight-dimensional space. The hope is to find clusters in this space. For visualization purposes, it's best to take a two-dimensional projection of the space. The simplest such projections are along pairs of axes; for example, a program might simply plot each spike as a dot whose x component would give the minimum voltage and whose y component would give the time lag to the peak on the right. More generally, however, the two axes could be linear combinations of the original coordinates. After storing the eight (or however many) parameters in an $N \times 8$ matrix, N the number of spikes, practitioners use a *principal-component analysis* to determine the directions in the multidimensional space (eight-dimensional in the example above) that account for as much as possible of the variation among spikes.[6]

Wavelet transforms

Instead of heuristic measures or principal components based on statistics, I chose two wavelet components of the spikes. A wavelet transform, like the Fourier transform, describes a signal in a different form: in both cases, if I start out with a time series of N points, the transform also consists of N points. (In the Fourier case, these may be complex, but for the wavelets, they're real.) In both cases, I can transform back to recover my original set of N points. Consider

[6] H. Hotelling, *J. Educ. Psych.* **24** (1933) 417. Didactic treatments may be found in textbooks on multivariate statistics or on clustering, as in M.R. Anderberg, *Cluster Analysis for Applications*, Wiley, 1973 §5.1.3.3. The idea is simply that covariance-matrix eigenvectors with successively smaller eigenvalues correspond to uncorrelated linear combinations of the original variables with successively smaller variances. This relates to singular-value decomposition, for which see *Numerical Recipes*, chapters 2 and 15.

the original data and their discrete Fourier transform. Each Fourier component represents the complex amplitude of waviness at a given frequency; any Fourier component transformed back by itself is a sine wave, infinite in extent but extremely localized in frequency. Conversely, the original representation of the data can be thought of as a set of amplitudes multiplying delta functions, which are extremely localized. The wavelet transform lies in between, somewhat localized in space and somewhat localized in frequency, but not extremely localized in either sense. I refer you to *Numerical Recipes* for an introduction.[7]

The wavelet algorithm is fractal, picking out important features at different time scales. Roughly speaking, where a given Fourier component tells me how much sine-wave of a given frequency there is in the signal, a given wavelet component tells me how much of a hump there is of a given width starting in a given sector of my data. From this intuitive description, it ought to be a good way of sorting out spikes.

To sort spikes, I used the Daubechies-4 wavelets described in *Numerical Recipes*, adapting their `wt1` and `daub4` routines by re-writing them in C (instead of Fortran-translated-badly-to-C). For each 50-point snippet, the wavelet transform returned 50 wavelet components. I wrote a few scripts to determine for each snippet which five components were most important (largest). I then chose the two indices that occurred most often in the top five lists, plotting these two wavelet components against each other. This yielded figure 5a. While not in any way optimized, the obvious clustering suggests that I've succeeded in sorting two neural signals.

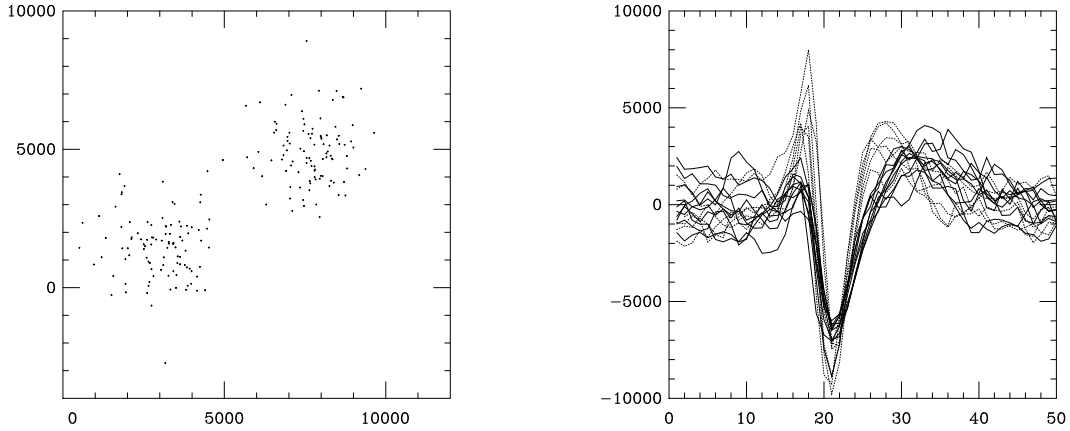


Figure 5. (a) A scatter plot of the two principal Daubechies-4 components for 208 identified spikes; (b) a random selection from the two clusters, with those from the lower left plotted in a solid line and those from the upper right dotted.

Note in figure 5b several features (overall depth, a gap, location of right shoulder) that distinguish the two identified clusters. If we had more time for this project, I'd recommend doing statistical analyses to ensure that the apparent clustering in figure 5 could not be accounted for by sampling error (discrete measurement times) or by some other accident.[8]

Shortly after the first version of this course was offered, Letelier and Weber found that a Daubechies-8 transform did a better job of spike sorting than traditional principal-component methods.[9]

[7] For a thorough reference, see Meyer, *Ondelettes*, transl. Ryan, *Wavelets: algorithms and applications*, SIAM, 1993.

[8] The CDAT-16 digitizer employs 64-times oversampling, significantly reducing the effect of sampling error or aliasing.

[9] J.C. Letelier and P.P. Weber, "Spike sorting based on discrete wavelet transform coefficients," *J. Neurosci. Meth.* **101** (2000), 93.

Digression on wavelet filtering

To get a feeling for how much better suited to neural spikes wavelets are compared to Fourier components, I applied the Daubechies-4 wavelet transform to the entire data set, then threw away (set to 0) the 90% of wavelet components that were smallest before transforming back. Noise was visibly reduced *without affecting the lineshapes of the spikes at all.* I certainly would not want to try the same thing with a Fourier transform: all I'd have left would be noise. I did compare the Daubechies filtering to a Fourier high-pass filter that cut out everything below 500 Hz and let everything above 1 KHz pass. As already noted, localized spikes have significant weight at *every* frequency, so the high-pass filter severely degraded them.

Autocorrelation

Now that you've identified the neurons in the time series, you can start looking for temporal patterns. For example, is one neuron firing in a particularly ordered way? Correlation functions—see *Numerical Recipes* for definitions—also provide a check on the sorting procedure, since one expects a neuron to be somewhat better correlated with itself at a different time than with its neighbors (although neighbors are probably also correlated). You could compare the autocorrelation functions of the different sequences you've identified to the autocorrelation of the whole data set, to correlations between neurons, and to autocorrelations for parts of the data (say, the first and second halves). You can now replace the actual waveforms with Kronecker delta functions at their trigger points (*i.e.*, 1 "at" each spike, 0 everywhere else).

Conclusions

I've indicated several places where I could have done a more thorough job; in particular, it is likely that I've missed (failed to trigger on) some spikes, and there may very well be more than the two neurons I've identified (not counting a solitary outlier). I've suggested more ideas than anyone will have time to explore in two weeks. I hope each student will pick a different set; it will be interesting to compare results.

Physics Z-5156, 2006

Chapter 5: Integrating Regular and Chaotic Classical Orbits[1]

We shall apply computational methods to a simple but interesting problem in classical mechanics: the relative motion of two atoms interacting via the Lennard-Jones 6-12 potential. The potential energy is

$$V(r) = V_0 \left[\left(\frac{\sigma}{r}\right)^{12} - 2\left(\frac{\sigma}{r}\right)^{6} \right] \quad , \tag{1}$$

where r is the distance between the two atoms. You should use the graphic tools at your disposal to draw a graph of $V(r)$ *vs.* r. The potential energy has a minimum at $r = \sigma$, and V_0 is the potential energy at the minimum relative to the potential energy as the atoms separate completely ($r \to \infty$). For $r \gg \sigma$, there is an attractive force $\propto r^{-7}$ between the atoms. This is the van der Waals attraction, which originates in the electrical polarizability of the atoms. For $r \ll \sigma$, the atoms strongly repel one another. The precise form of the repulsive force, $\propto r^{-13}$, is chosen for convenience, but the existence of some "hard core" repulsion is understood in terms of the Pauli exclusion principle. In mechanics we learn that we can separate the motion of the center of mass of a pair of particles from the relative motion. We are interested in the relative motion of masses m_1 and m_2, which is the same as the motion of one particle whose mass, $m = m_1 m_2/(m_1 + m_2)$, is the "reduced mass" of the pair:

$$m\frac{d^2\mathbf{r}}{dt^2} = -\,\boldsymbol{\nabla} V \quad , \tag{2}$$

where $\mathbf{r}$ is the vector from one atom to the other and, for us, V is given by (1). We also learn that energy is conserved, so there is a constant E for which

$$\frac{m|\dot{\mathbf{r}}|^2}{2} + V(r) = E \quad , \tag{3}$$

and, since the force is central, angular momentum is conserved, so there is a constant vector $\mathbf{L}$ for which

$$m\mathbf{r} \times \dot{\mathbf{r}} = \mathbf{L} \quad . \tag{4}$$

Let's simplify the equations by appropriate choice of units and coordinates: we choose the unit of *length* so that $\sigma = 1$. We choose the unit of *mass* so that $m = 1$. We choose the unit of *time* so that $V_0/m\sigma^2 = 1$. We choose the coordinate system so that the direction of $\mathbf{L}$ is the z axis. The motion is then confined to the x-y plane. The usual discussion in a mechanics course uses polar coordinates r, ϕ. With our conventions, a little algebra establishes

$$\frac{\dot{r}^2}{2} + \frac{r^2\dot{\phi}^2}{2} + V(r) = E \quad , \tag{5}$$

and

$$r^2\dot{\phi} = L \tag{6}$$

with

$$V(r) = r^{-12} - 2r^{-6} \quad . \tag{7}$$

[1] Dr. Donald Fredkin wrote the original version of this chapter for an undergraduate version of this course, which we taught together at the University of California, San Diego, in 1996. I adapt it here, with some small changes and additions, by his permission.

We rewrite (6) as

$$\dot{\phi} = \frac{L}{r^2} \tag{8}$$

and substitute (8) into (5) to obtain

$$\frac{\dot{r}^2}{2} + V_{\text{eff}}(r) = E \quad , \tag{9}$$

with

$$V_{\text{eff}}(r) \;=\; V(r) + \frac{L^2}{2r^2} \;=\; r^{-12} - 2r^{-6} + \frac{L^2}{2r^2} \quad . \tag{10}$$

Equation (9) is energy conservation for one-dimensional motion in the "effective potential" $V_{\text{eff}}(r)$ given by (10). We start with this one-dimensional radial problem.

1. Turning points: finding zeroes of a function

For sufficiently small angular momentum L, there is a range of energy E such that there are exactly two values of r for which $V_{\text{eff}}(r) = E$, and therefore, by (9) $\dot{r} = 0$. These special values, which we denote r_p (for periapsis, the closest approach) and r_a (for apoapsis, the furthest distance apart), with $r_p < r_a$, are called turning points. Our first task is to write a function `turn` to compute the turning points. The prototype for `turn` is

```
typedef struct pair {
        double a, b;
} pair;

/* Return the turning points for the radial motion with energy E and angular
 * momentum L in a Lennard-Jones potential.  If the combination (E, L) is not
 * possible, return a negative value for a.  Otherwise, 0 < a < b.
 */
pair turn(double E, double L);
```

The function `turn` has to find zeroes of $E - V_{\text{eff}}(r)$. There is one rule to be remembered about one-dimensional zero finding: to find a zero of $F(x)$, first find a bracketing interval $a < x < b$ such that $F(a)$ and $F(b)$ have opposite signs, and then keep making the interval shorter while maintaining the bracketing property. Once a bracketing interval is established, there are excellent general-purpose algorithms for refining the bracketing interval.

Here is one strategy for finding the turning points: there is a critical value L_c of the angular momentum such that no bounded orbits exist for $L \geq L_c$ (see the figure 1). You should compute the exact value of L_c for which there is a horizontal inflection point and the radial distance r_c at which that inflection point occurs. Now, if $L < L_c$, we can find a minimum of $V_{\text{eff}}(r)$ at $r_{\min}$ and a maximum of $V_{\text{eff}}(r)$ at $r_{\max}$ (see the figure for $L = 1.1$). These extrema, $r_{\min}$ and $r_{\max}$, can be determined by applying a standard zero-finding function, which you will write, to $V'_{\text{eff}}(r)$ as soon as bracketing intervals are available. For $r_{\min}$, a bracketing interval is given by $r_1 < r_{\min} < r_c$ for any $r_1 < r_c$ at which V_{eff} has a negative slope; such an r_1 can always be found by starting with the trial value $r_1 = r_c$ and successively halving r_1. Similarly, for $r_{\max}$, a bracketing interval is given by $r_c < r_{\max} < r_2$, for $r_2 > r_c$ and $V'_{\text{eff}}(r_2) < 0$; such an r_2 can always be found by starting with the trial value $r_2 = r_c$ and successively doubling r_2. Now a bounded orbit exists if and only if $V_{\text{eff}}(r_{\min}) < E < V_{\text{eff}}(r_{\max})$. Further, $r_{\min} < r_a < r_{\max}$ brackets the apoapsis r_a, and a bracketing interval $r_0 < r_p < r_{\min}$ can be readily found for the periapsis; we leave it to you to determine a suitable r_0. Now write the function `turn`. Test it by linking it to the given main program.

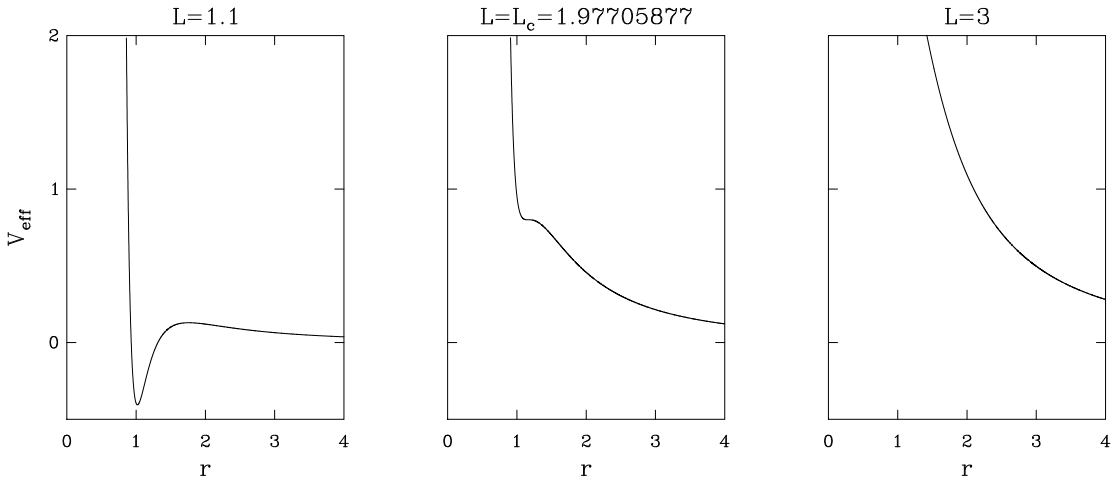


Figure 1. The one-dimensional effective potential (10) for angular momentum less than, equal to, and greater than the critical value. Note that the latter two have no potential well in which the system can orbit. For $L < L_c$, V_{eff} has a minimum at $r_{\min}$ and a local maximum at $r_{\max}$; if the total energy E satisfies $V_{\text{eff}}(r_{\min}) < E < V_{\text{eff}}(r_{\max})$, then the orbital distance r will vary between periapsis r_p and apoapsis r_a.

```
/* tstturn.c (header comments elided) */
#include <stdio.h>
#include <stdlib.h>
#include "turn.h"
int
main(int argc, char **argv)
{
        double E, L;            /* energy and angular momentum */
        pair tp;                /* turning points */

        if (argc != 3) {
                fprintf(stderr, "usage: %s E L\n", argv[0]);
                return 1;
        }
        E = atof(argv[1]);
        L = atof(argv[2]);
        tp = turn(E, L);
        if (tp.a > 0)
                printf("periapsis = %g\napoapsis = %g\n", tp.a, tp.b);
        else
                printf("no bounded orbit exists\n");
        return 0;
}
```

2. Radial period: numerical integration

From (9) we have

$$\dot{r} = \pm\sqrt{2(E - V_{\text{eff}}(r))} \quad , \tag{11}$$

or

$$dt = \pm \frac{dr}{\sqrt{2(E - V_{\text{eff}}(r))}} \quad , \tag{12}$$

where the sign is + when r is increasing and − when it is decreasing. The time to complete one cycle of the radial motion, with r varying from r_p to r_a and back to r_p, is, therefore,

$$T_r = \sqrt{2} \int_{r_p}^{r_a} \frac{dr}{\sqrt{E - V_{\text{eff}}(r)}} \quad . \tag{13}$$

Our next task is to write a function `period` to compute T_r. The prototype for `period` is

```
/* Return the period for the radial motion with energy E and angular momentum
 * L in a Lennard-Jones potential.  If the combination (E, L) is not possible,
 * return a negative value.
 */
double period(double E, double L);
```

The function `period` should use `turn` to find r_p and r_a and then evaluate the integral (13) numerically to some reasonable precision. There are many ways to evaluate integrals numerically, and it does not usually matter much which method is chosen unless the program is going to evaluate thousands of integrals. However, there is a special problem here: the integrand diverges at the end points, although the integral does exist. Numerical integration methods that evaluate the integrand at the end points, or too close to them, are likely to fail. (Worse, the program might try to evaluate a square root of a negative number, because r_p and r_a are known only approximately.) Part of the solution is to use a so-called "open" formula, which does not evaluate the integrand at the end points. This, alone, does not sufficiently avoid the singularity at the end points, but there is a nice variation of Gaussian integration called the Gauss-Chebyshev formula, which computes

$$\int_{-1}^{1} \frac{F(x)\,dx}{\sqrt{1 - x^2}} \;\approx\; C_n \sum_{k=0}^{n-1} F(\xi_k) \tag{14}$$

for suitable coefficient C_n and evaluation points $\xi_0 \ldots \xi_{n-1}$, which can be found in reference books or computed.[2] Let us rewrite (13) in the form

$$T_r = \int_{r_p}^{r_a} \frac{F(r)\,dr}{\sqrt{(r - r_p)(r_a - r)}} \tag{15}$$

with

$$F(r) = \frac{\sqrt{2(r - r_p)(r_a - r)}}{\sqrt{E - V_{\text{eff}}(r)}} \quad . \tag{16}$$

Note that $F(r)$ is regular at the end points. To apply (14) to (15), we make the change of integration variable

$$r \;=\; r_p + \frac{r_a - r_p}{2}(\rho + 1) \;=\; \frac{r_a + r_p}{2} + \frac{r_a - r_p}{2}\rho \quad . \tag{17}$$

Then (15) becomes

$$T_r = \int_{-1}^{1} \frac{F\left(\frac{r_a + r_p}{2} + \frac{r_a - r_p}{2}\rho\right)\,d\rho}{\sqrt{1 - \rho^2}} \quad , \tag{18}$$

[2] See, for example, *Numerical Recipes* §4.5.

which is ready for approximation using (14).

Now write `period`. Link it to the `tstperiod` driver given below, and test it; the driver lets you experiment with the quadrature order n in (14) from the command line. You can compute $\lim_{E\to V_{\rm eff}(r_{\rm min})} T_r$ exactly, providing one check on your function `period`.

```
/* tstperiod.c */
#include <stdio.h>
#include <stdlib.h>
#include "period.h"

int main(int argc, char **argv)
{
        double E, L, Tr;        /* energy, ang mom, radial period */
        int n=10;               /* default quadrature order */

        if (argc < 3 || argc>4) {       /* very simple argument parsing */
                fprintf(stderr, "usage: %s E L [n]\n", argv[0]);
                return 1;
        }
        E = atof(argv[1]);
        L = atof(argv[2]);
        if(argc==4)
                n = atoi(argv[3]);
        if ( (Tr = period(E, L, n)) < 0)        {
                fprintf(stderr, "no bound orbit exists\n");
                exit(1);
        }
        printf("radial period = %g\n", Tr);
        return 0;
}
```

3. Orbits: numerical solution of ordinary differential equations

The direct approach to the determination and study of orbits is numerical solution of (2). Most numerical methods for solution of ordinary differential equations are designed for systems of first-order equations, so we rewrite the second-order (2) as

$$\begin{aligned} \frac{d\mathbf{r}}{dt} &= \mathbf{v} \\ \frac{d\mathbf{v}}{dt} &= -\,\boldsymbol{\nabla} V \quad . \end{aligned} \tag{19}$$

Defining the four-dimensional vector u by

$$\begin{aligned} u_1 &= x \\ u_2 &= y \\ u_3 &= v_x \\ u_4 &= v_y \end{aligned} \tag{20}$$

gives (19) the practical form

$$\begin{aligned} \dot{u}_1 &= u_3 \\ \dot{u}_2 &= u_4 \\ \dot{u}_3 &= -\partial V/\partial u_1 \\ \dot{u}_4 &= -\partial V/\partial u_2 \end{aligned} \tag{21}$$

For the potential given in (1), equation (21) takes the definitive form

$$\begin{aligned} \dot{u}_1 &= u_3 \\ \dot{u}_2 &= u_4 \\ \dot{u}_3 &= f u_1 \\ \dot{u}_4 &= f u_2 \end{aligned} \tag{22}$$

with the definitions

$$\begin{aligned} f &= 12(r_{\rm sq}^{-7} - r_{\rm sq}^{-4}) \\ r_{\rm sq} &= u_1^2 + u_2^2 \quad . \end{aligned} \tag{23}$$

Now write a program called `orbit` to compute N points on an orbit, starting at $u = u(0)$ when $t = 0$ and ending at $t = T$. Use whatever method you like,[3] but write the program so that you can invoke it with the command

```
orbit x0 y0 vx0 vy0 T N
```

The program should write one line to standard output for each of the N times, starting at $t = 0$ and ending with $t = T$. Each line should contain seven numbers, separated by white space: t, x, y, vx, vy, E, and L, where E and L are the energy and angular momentum. Although these two quantities are conserved in nature, they might not be when you run your program because of numerical errors (and, possibly, because of programming errors). You will have to decide on appropriate error tolerances. If E and L are not constant to within the intended accuracy of your solution, something is wrong. Plot some orbits found with `orbit`. Be sure the scales on the x and y axes are the same, and plot y *vs.* x. A good set of initial conditions for getting started is $x = 1.1$, $y = 0$, $v_x = 0$, $v_y = 0.1$. What other plots are interesting? This is the point at which you should make all your programming pay off in insight. Use your programs. Look at x as a function of t, and make a spectral analysis using the fast Fourier transform. Recall the considerations that enter such an analysis: T determines the frequency resolution, N/T (the sampling rate) must be high enough to avoid aliasing, and windowing may be necessary to reduce leakage (but then resolution suffers). Make a semilogarithmic plot (the y-axis should be logarithmic) of the spectrum. What features do you see? Can you explain these features qualitatively? Can you explain them quantitatively using your earlier programs? Make a linear plot to see why we recommend a semilogarithmic plot.

4. Orbits: chaotic motion

Orbits in the Lennard-Jones potential (1) and the corresponding spectra are easy to understand, because the dynamics is "separable" in polar coordinates. Now let's consider motion in the noncentral potential

$$V(x,y) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + x^2 y - \frac{1}{3}y^3 \quad , \tag{24}$$

[3] We recommend using a serious multistep ODE solver. Since this problem is not "stiff" (see *Numerical Recipes* §16.6), an Adams method is appropriate. There are several good solvers available from Netlib and elsewhere. Most are written in Fortran, which is a nuisance. CVODE is written in C and pretty much represents the current state of the art.

which was introduced by Hénon and Heiles to model aspects of dynamics in globular clusters.[4] The equations of motion are

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \dot{v}_x &= -x - 2xy \\ \dot{v}_y &= -y - x^2 + y^2 \quad . \end{aligned} \tag{25}$$

Make a copy of `orbit.c`, calling the copy `hh.c`. Edit `hh.c` so that it solves (25). Note that angular momentum is no longer conserved, so there is no point printing it on each line. Plot some orbits found with `hh`. In particular, look at the two sets of initial conditions

$$\begin{array}{llll} x = 0.1 & y = 0 & v_x = 0 & v_y = 0.1 \\ x = 0.8 & y = -0.5 & v_x = 0 & v_y = 0 \end{array} \tag{26}$$

Plot the spectrum of $x(t)$ for each orbit that you study. See if you can correlate the character of the spectra with the appearance of the orbits. You should also look at plots of x and y *vs.* t. Which of these views of an orbit are most useful? Using appropriate plotting software,[5] you might examine three-dimensional plots of $(x(t), y(t), t)$.

Plotting a complete phase-space trajectory, $(x(t), y(t), \dot{x}(t), \dot{y}(t), t)$, would require a 5-D graphics program. As in the paragraph above, the four-dimensional phase space can be projected on a plane; the resulting pictures tend to look like flattened-out versions of higher-dimensional paths (because, of course, they are). A particularly useful way of looking at trajectories is through *Poincaré sections*: Hénon and Heiles plot $(y, \dot{y})$ only at those times when the trajectory passes through the $x = 0$ plane in the positive direction ($\dot{x} > 0$). The very simplest kind of orbit will always pass through the $x = 0$ plane at a single, zero-dimensional, point. For different initial conditions, the trajectory will hit the $x = 0$ plane at a different place each time it passes ((a) in Figure 2); as more an more points are added, the section converges to a one-dimensional line, (b). One-dimensional sections may be multiply connected, as in (c). For certain ranges of energy and initial conditions, however, much more complicated sections emerge. In some cases, the points may densely fill a two-dimensional area. However, it is far more interesting when the pattern has a fractal *Hausdorff dimension* intermediate between 1 and 2; this may be the case in (d). Fractal dimension is one hallmark of chaos; we will discuss in class what it means for a set to have a fractional dimension and how to calculate it.

The usual definition of chaos is a sensitive dependence on initial conditions: that is, two trajectories that start out with arbitrarily close initial conditions diverge exponentially in time, at least for short enough times. Giordano (*Computational Physics*, Prentice-Hall, 1997—we use this textbook in PHY 4151) examines the motion of the forced, damped pendulum,

$$\ddot{\theta} = -\omega_0^2 \sin\theta - q\dot{\theta} + f_0 \sin(\omega_D t) \quad ,$$

where θ is the angular coordinate, q a frictional coefficient, f_0 the coefficient of the driving perturbation at frequency ω_D, and ω_0 the natural frequency in the absence of the driving and frictional terms. This is an even simpler model than the Hénon-Heiles potential we're investigating and one in which sensitive dependence on initial conditions is manifest. For all the other parameters fixed, the system goes through a transition as a function of driving amplitude, f_0. If f_0 is zero or small enough, friction (which of course is absent in Hénon-Heiles, a Hamiltonian system) will eventually cause the pendulum to come to rest at the bottom ($\theta = 0$), regardless of initial conditions. Figure

[4] M. Hénon and C. Heiles, *Astron. J.* **69** (1964) 73.

[5] *E.g.*, the supplied `plot3d` program or `gsplot` in Octave.

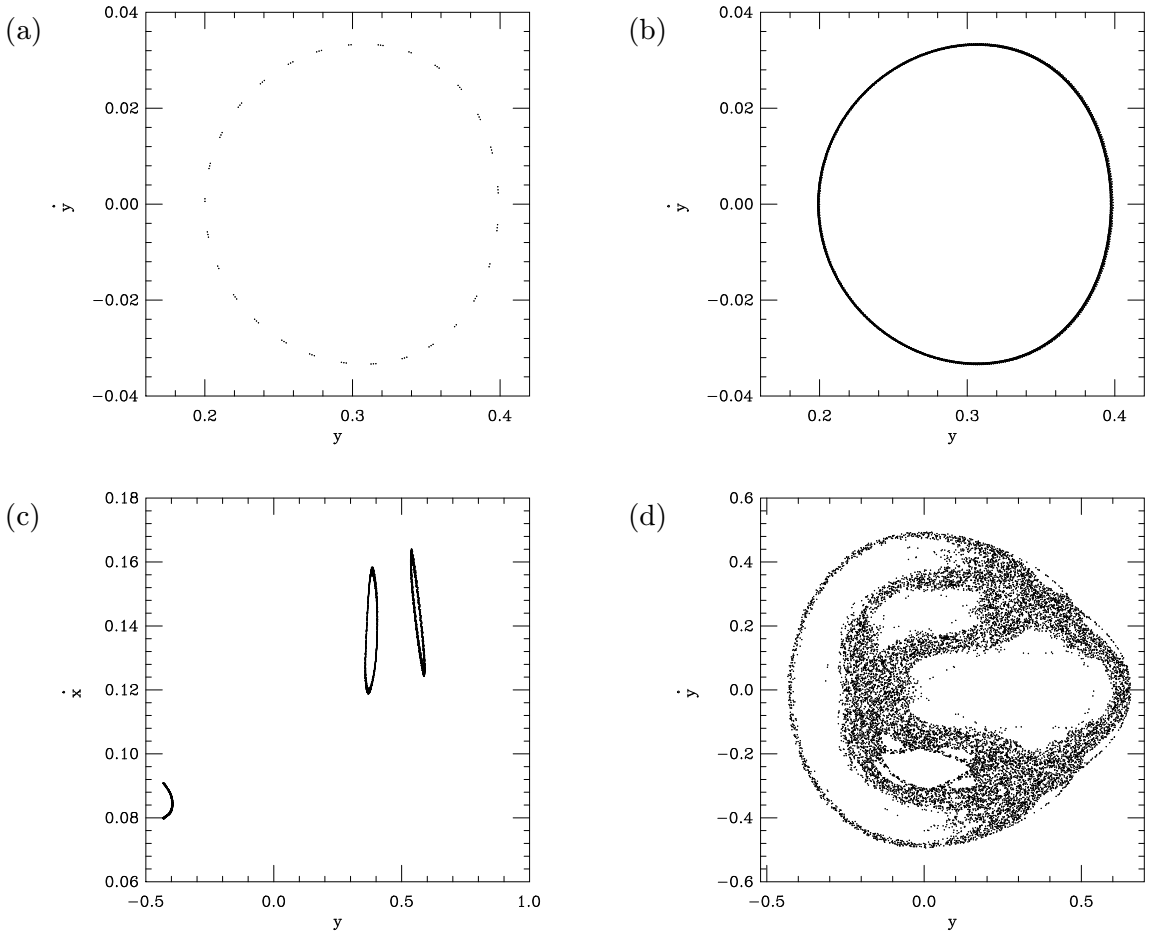


Figure 2. (a) Poincaré section for the Hénon-Heiles potential with particular initial conditions under which the motion is regular. (b) As more and more orbits are integrated, the Poincaré section fills in to a one-dimensional curve. (c) Poincaré sections need not be simply connected. (d) For some initial conditions, the Poincaré set becomes fuzzy; the set is no longer one-dimensional, but it still takes up zero area. One might think the fuzziness is a result of transients, but I've suppressed these. Nor is it obvious that the dark areas get filled in completely as one integrates further in time. Any resemblance to a Munch painting is purely coincidental.

3(a) shows the difference in θ as a function of time between two trajectories that start out quite close; the difference decreases exponentially with time. However, above some critical driving force, two initially close trajectories diverge exponentially, as shown in (b). In both cases, for sufficiently small times, the divergence can be written

$$\lim_{\Delta\theta(t=0)\to 0} \Delta\theta \sim \exp(\lambda t) \quad .$$

When the Lyapunov exponent, λ, is positive, as in Figure 3(b), the dynamics is chaotic. Many practical problems are difficult precisely because of a positive λ; the weather, for example, is predictable for times less than roughly λ^{-1}, beyond which the sensitive dependence on initial conditions makes long-range forecasting impossible.

Mitchell Feigenbaum and others have found that disparate physical systems exhibiting chaos fall into a few classes. In many cases, as a changing parameter drives the system from regular to

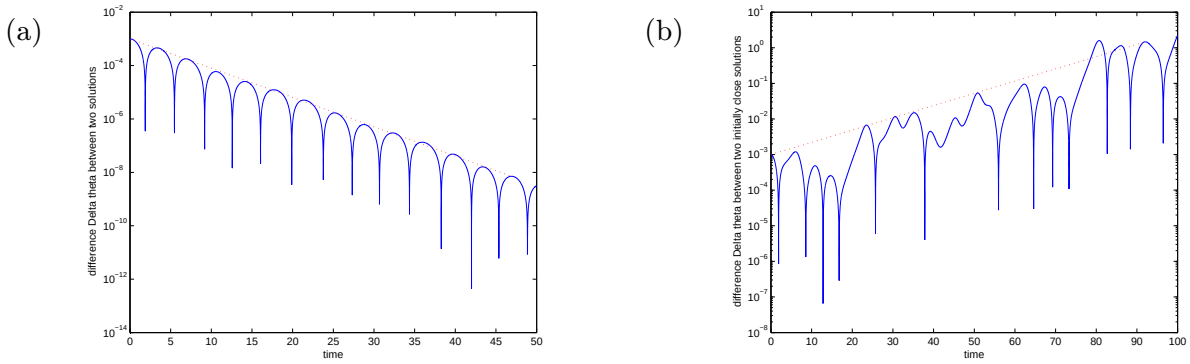


Figure 3. Negative (a) and positive (b) Lyapunov exponents in the forced, damped pendulum. Adapted from Giordano, *Computational Physics*, Fig. 3.5.

chaotic motion, a series of period doublings can be observed. In our system, these might take the form of first one, then two, then four pieces of a Poincaré section, with the doublings accumulating at the critical value of the parameter at which full-blown chaos ensues. This doubling might also be observable in the power spectrum.

Possible investigations for the project could include computation of the Lyapunov exponent as a function of energy with other parameters fixed, the calculation of the Hausdorff dimension of chaotic sections, or looking for evidence of a period-doubling route to chaos. There are concise discussions of chaos in both Giordano (§§3.2–3.7) and Thijssen (pp. 2–4 and 11–13).

Physics Z-5156, Fall 2006

Chapter 6: The Pricing of American Put and Call Options

DISCLAIMER: THIS CHAPTER SHOULD NOT BE CONSTRUED AS INVESTMENT ADVICE. I DON'T WANT TO GET IN TROUBLE WITH THE SECURITIES-AND-EXCHANGE COMMISSION. OPTIONS CAN BE VERY RISKY; RECENTLY, A MULTI-BILLION-DOLLAR INVESTMENT BANK WITH A 200-YEAR HISTORY WENT BANKRUPT BECAUSE OF BAD CHOICES IN DERIVATIVE SECURITIES.[1]

Stocks and Options

They sought it with thimbles, they sought it with care;
They pursued it with forks and hope;
They threatened its life with a railway share;
They charmed it with smiles and soap.

To raise money for expansion, a company may sell *shares*, each representing partial ownership. Anyone may subsequently buy or sell shares on a secondary market, such as a stock exchange. The price at which these shares sell fluctuates on a timescale of minutes or less and is influenced by (among other factors) the company's recent performance, its ability to pay dividends to shareholders, the public perception of what it's likely to do in the next months or years, interest rates, psychology, and mass hysteria.

Institutions, such as pension funds, and individuals who invest in stocks are not always comfortable with the idea that the future value of their investments may be a function of mass hysteria. Other people *like* volatility and seek a way of realizing higher gains than possible with a stock (at the risk of greater losses). Options and similar *derivatives* provide a means to manage risks and potential gains. Options and futures contracts are probably even more important in commodity (*e.g.*, cotton, copper, pork bellies) and currency markets, where large fluctuations would otherwise seriously threaten a farmer's, supplier's, or international company's ability to carry on business. Recently, they have received the attention of electric utilities.[2] We consider stocks in this course, because data on stock options are easily accessible.

Perhaps the simplest derivative investment is a futures contract, in which two parties agree to carry out a transaction (for instance, the sale of 100 metric tonnes of pork bellies) on a particular date at a particular price. A European *put* or *call* is similar, except it is optional on one party. The bearer of a **call** contract has the right, but not the obligation, on the expiration date to **purchase** 100 shares of a particular stock at a previously-agreed *strike price*. Obviously, he will do so only if the strike price is less than the market price on that day. The bearer of a **put** contract has the right, but not the obligation, on the expiration date to **sell** 100 shares of a particular price at a previously-agreed strike price. He will do so only if the strike price is greater than the market. Like stocks and bonds, options trade on the secondary market. Unlike securities, an individual may issue (write) an option, which then becomes binding on him.

An American option permits exercise on the same terms any time on or before the expiration date. An option that can be exercised is termed "in the money;" for a call, this means that the

[1] J. Rawnsley, *Total Risk: Nick Leeson and the Fall of Barings Bank*, Harper-Collins, New York, 1995. I understand that Nick Leeson was one of the few derivatives traders without a Ph.D. in math or physics.

[2] I wrote that sentence in 1999, well before the California power crisis of spring 2001. One of the peculiar features of California electric deregulation was a prohibition on the use of futures contracts by the utility companies.

exercise (or strike) price E is less than the current market value S of the stock, while for an in-the-money put, $E > S$. The geographical names appear to be historic; both kinds of options trade on all continents. However, stock options (not index options) trading on the Chicago Board and other options exchanges in the United States are all of the American variety. Other kinds of options exist; one that's half-way between the two is called "Bermudan," while exotic options involving complicated functions of a stock's history are generically referred to as "Asian."

My treatment below mostly follows that of Wilmott *et al.*, *Option Pricing* (see the reference list toward the end).

Why this is physics

I am told that a majority of the people trading derivative securities for a living have Ph.D.s in either physics or mathematics. Beside the obvious necessity for quantitative analysis, the reason becomes clear once we consider the fundamental model used in the industry. As with most physical models, we begin with a simplification: all the many factors going in to the price of a stock are inherently unpredictable and so constitute essentially noise. (If I *could* reliably predict the future price of a stock, I would quickly make enough money to buy out the National Science Foundation and to rename USF). On the other hand, over time, stocks tend to appreciate in value. So we model the *return* on a stock as a biased random walk; on average, the rate of return (*i.e.*, percentage per year) might be μ, but on top of this we add noise.

Thus if we know the current stock price, S, we model the infinitesimal increment dS over an infinitesimal period of time dt as

$$dS(t) = \mu S(t)dt + \sigma S(t)dX(t) \quad , \tag{1}$$

where $dX(t)$ represents a random variable, positive or negative, drawn from a Gaussian distribution of variance dt and completely uncorrelated with $dX(t')$ for $t \neq t'$. I will show that under these assumptions, $\ln S$ follows a biased random walk.

The simplest investment: compound interest

As the simplest example of an investment, consider a deposit $S(0)$ left in a bank or United-States Treasury bill for a year, an investment carrying negligible risk to principal. Let the interest rate be r, for example 6% *per annum*, which we assume does not change. If the deposit pays simple interest, in one year it will be worth $S(1\text{year}) = (1 + r)S(0) = 1.06S(0)$. If, however, for the same quoted annual rate, the bank compounds the interest monthly, each month for twelve months I will multiply the balance from the end of the previous month by the factor $(1 + r/12)$, or 1.005 if $r = 6\%$. Thus $S(1\text{year}) = (1 + r/12)^{12}S(0)$. More generally, $S(t) = (1 + rt/n)^n S(0)$, where n is the number of periods in a time t.

The most liquid institutional money-market instruments might compound interest daily (multiplying each previous day's balance by $1 + r/365.25$). It is convenient to take the limit as $n \to \infty$. By a simple binomial expansion and Taylor series, we find

$$S(t) = e^{rt}S(0) \quad , \tag{2}$$

which solves the differential equation

$$\frac{dS}{dt} = rS(t) \tag{3}$$

for the initial condition $S(0)$. Equation (3) states that the money $S(t)$ increases proportionately at the rate r.

Money can be negative: that represents taking out a loan, which one might do in order to invest in stocks (not necessarily recommended). For the purposes of this course, we will assume that the

same interest rate r governs both bank (or Treasury) investments and loans. We'll pretend that r does not change and so is known in advance and ignore any dependence on maturity (*e.g.*, for a Treasury instrument), assuming we can easily redeem our investment (or pay back our loan) at will. Assume there are no costs (*e.g.*, broker's commissions) for any of the stock or other transactions we make; this is not a bad approximation for large, institutional investors. We also ignore inflation and all consequences of tax law.[3]

Arbitrage and European put-call parity

Stocks, unlike bank deposits, involve risk; it is difficult to calculate the "right" value for a stock. However, the Black-Scholes analysis provides a way of calculating the value of an option. The analysis relies on the concept of arbitrage. If someone in New York is willing to sell a stock at \$1.125, and someone in Philadelphia is willing to buy the same stock at \$1.133, an arbitrager will buy in New York and simultaneously sell in Philadelphia, making a riskless profit (we ignore transaction costs). Similarly, if an option is selling at the "wrong" price, an arbitrager can make risk-free money by borrowing money and buying (if the price is low) or selling short (if it's high) and investing the proceeds.

As an example of the kind of arbitrage argument that later leads to Black-Scholes, consider buying one unit of stock currently trading at price $S(t = 0)$ and a European put with exercise price E due to expire in time T. At the same time, write (sell) a European call with the same exercise price and expiration.

While the correct price may not be apparent now, the future correct price at expiration is very easy to calculate. If $S(T) < E$, the call expires worthless, but I can exercise the put, selling the stock for E. On the other hand, if $S(T) > E$, my put is worthless, but the holder of the call can exercise it, forcing me to sell my stock, again for E. If $S(T) = E$ exactly, both options are worthless, and I sell my stock for E. Thus the payoff at expiration is E in all cases, without any risk.[4]

Thus the portfolio $\Pi = S + P - C$ resembles a bank deposit, so it had better cost the same at time $t = 0$ as does a bank deposit paying E after time T; from (2), this is Ee^{-rT}. If $\Pi < Ee^{-rT}$, an arbitrager can borrow money at rate r to buy more Π, risk free. Conversely, if $\Pi > Ee^{-rT}$, the arbitrager sells the stock short, writes a put, and buys a call, investing the free money at rate r.

Professionals with huge credit lines constantly look for such investments, and by taking advantage of any almost immediately drive the market prices in the direction to make further arbitrage impossible. Thus, if the markets operate efficiently, there should be no arbitrage opportunities left; in particular (for European options),

$$S + P - C = Ee^{-rT} \quad . \tag{4}$$

This gives a relation between the three instruments (stock, put, call) and the (assumed) fixed interest rate; Black-Scholes will provide a mechanism to price P and C individually.

Random processes

Before explaining the noisy part of a stock price modeled in (1), let me review a more familiar random process, the random walk. Starting at position $Y_0 = 0$, at each time step i, I add to Y a random variable X_i, which is $+1$ with probability $1/2$ and -1 with probability $1/2$. The random variable is unbiased, $\langle X_i \rangle = 0$, and I assume that all the time steps are uncorrelated,

[3] By assuming the spherical horse, we have firmly established this as physics. Of course, such considerations can be put in later.

[4] Even if the company that issued S declares bankruptcy, $S(T) = 0$ is just a special case of $S(T) < E$.

$\langle X_i X_j \rangle = \delta_{ij}$. Then at the time given by n timesteps, the position $Y_n = \sum_{i=1}^{n} X_i$. The mean position $\langle Y_n \rangle = \langle X_1 \rangle + \langle X_2 \rangle + \ldots = 0$. but the mean square position $\langle Y_n^2 \rangle = n$, which since the mean vanishes is also the variance. In other words, we go nowhere in particular, but we end up a distance $\sqrt{\text{time}}$ (on average) from the origin.

The deviate, or random variable, X_i stands for an unknown value $+1$ or -1. If we wish to compute something definite, we can calculate the moments of X_i, or equivalently the probability distribution of Y_n. If n is even, the probability that $Y_n = 2m$ is

$$\text{Prob}_{Y_n}(2m) = \frac{1}{2^n} \begin{pmatrix} n \\ \frac{n}{2} - m \end{pmatrix} \quad . \tag{5}$$

Consider the large-n limit. By taking the logarithm, applying Stirling's approximation, and Taylor expanding the resulting logarithms (being careful to expand to a consistent order in $1/n$), we recover the well-known result that the binomial distribution reduces to a Gaussian with variance n/4:

$$\lim_{n\to\infty} \text{Prob}_{Y_n}(2m) = \sqrt{\frac{2}{\pi n}} e^{-2m^2/n} \quad . \tag{6}$$

This is just a special case of the central-limit theorem, which states that the sum of many independent random processes is Gaussian, provided the processes' second moments don't diverge.[5] In particular, a sum of Gaussian processes is Gaussian. If we imagine the stock price receiving a tiny kick at every tiny time step, we're going to end up with a Gaussian process anyway, so we might as well model the small kicks as also Gaussian. Otherwise the exact form of our noise will depend on the time scale.

The Gaussian process is conveniently scale invariant. Consider a random kick δX every time step δt (which need not be small) with probability density

$$\text{P}_{X,\delta t}(\delta X) = \frac{1}{\sqrt{2\pi\delta t}} e^{-(\delta X)^2/(2(\delta t))} \quad . \tag{7}$$

This normalized Gaussian has variance δt. What if we look only every $2\delta t$? The probability distribution for the noise on the longer time scale (*i.e.*, two time steps) is

$$\text{P}(\delta X) = \int du\, \text{P}_{X,\delta t}(u) \text{P}_{X,\delta t}(\delta X - u) \quad , \tag{8}$$

which after a little algebra comes out to exactly $\text{P}_{X,2\delta t}(\delta X)$ in (7), thus establishing the scale invariance of uncorrelated Gaussian noise.

We see explicitly in (7) that $\delta X \sim (\delta t)^{1/2}$, just as in the binary random walk.

Estimating volatility

If we assume Gaussian-distributed uncorrelated noise as in (1) and (7), we can measure the historical volatility of a stock over some period as the suitably normalized standard deviation of the population of periodic returns on the stock.[6]

[5] The convergence is non-uniform, which may account for the fact that experiments often show larger tails than would be predicted.

[6] Insofar as dividends represent return of stock-holder equity, stock prices should be adjusted for them. The Yahoo Web site referred to toward the end of the chapter does this, or you can avoid the subtlety by picking stocks that pay no dividends. Such stocks have tended to be more interesting lately anyway. I will provide some scripts and programs to help out.

As a practical matter, daily stock closing prices are more readily available than hourly or more frequent measures, so I'll assume you have a list of dates and closing values. The rate of return between day t_i and day t_{i-1} is

$$R_i = \frac{S(t_i) - S(t_{i-1})}{(t_i - t_{i-1})S(t_{i-1})} \quad , \tag{9}$$

indicating the fractional profit or loss in the period $t_i - t_{i-1}$. I write $t_i - t_{i-1}$ in the denominator instead of 1 day, because stocks do not trade publically on weekends or holidays; intervals over these times will be longer. If you want an annualized rate of return, you will need to measure time in years (days divided by 365.25, roughly); you can do the division later. The units of R_i are year^{-1}.

Before scaling, the volatility $\bar{\sigma}$ is simply the standard deviation, estimable as usual from the population by

$$\bar{\sigma} = \sqrt{\frac{1}{N-1}\sum (R_i - R)^2} \quad , \tag{10}$$

where N is the number of returns and R the average periodic return. We have already discussed robust one-pass and two-pass algorithms for the numerical computation of (10).

However, since the standard deviation of δX in (7) depends on the time scale δt, we must multiply by the square root of the time scale we measured:

$$\sigma = \bar{\sigma}\sqrt{\delta t} \quad . \tag{11}$$

As required in (1), σ has units of year$^{-1/2}$. The typical time scale $\delta t \approx 1$ day.

One might argue in favor of using $\delta t = \langle t_i - t_{i-1} \rangle$ and weighting daily returns differently after weekends and holidays; I found reasonably close agreement with published volatility estimates using the procedure just described. Although our model (1) assumes a fixed σ, the estimate of σ from any sample population will vary depending on the days included. Furthermore, the model may not apply well to stocks that undergo long periods of stasis before suddenly becoming volatile. Many sources uses a trailing period of 30 days to estimate volatility; in the assignment, you will compare this estimate to that inferred by fitting the Black-Scholes model to actual options trades.

Stochastic differential equations and Itô's lemma

Equities do not trade on a discrete time grid; they trade whenever a buyer and a seller agree on a price, and in any case it is more convenient analytically to deal with continuous than with discrete time. The noise term $dX(t)$ in (1) is supposed to represent the limit of δX for infinitesimal time, $\delta t \to dt$. This sort of equation is common in hydrodynamics, statistical mechanics, and plasma physics; dividing through by dt, we write

$$\dot{S}(t) = \mu S(t) + \sigma S(t) L(t) \quad , \tag{12}$$

immediately recognizable as a Langevin equation, with $L(t) = dX/dt$ a noise term satisfying $\langle L(t) \rangle = 0$ and $\langle L(t)L(t') \rangle = \delta(t - t')$.[7]

Because we imagine our kicks to occur instantaneously and infinitely often, they are also unbounded; their density makes them behave in some sense worse than isolated Dirac-δ functions. Put another way, the Wiener process $X(t) = \int_0^t dX(t')dt'$ is continuous, but its derivative $L(t)$ exists nowhere. It is not surprising, then, that the usual rules of calculus fail to apply to symbols such as dX and dS and to equations such as (1) and (12).

First, one must note that these two equations are in fact ill-posed without additional information. In integrating the last term of (1), are we to write

$$\int_0^t S(t)dX(t) = \lim_{n\to\infty} \sum_{j=0}^{n-1} S(jt/n)dX(jt/n) \tag{13}$$

[7] The higher moments factor; see van Kampen, *Stochastic Processes in Physics and Chemistry*, Elsevier, 1981.

or

$$\int_0^t S(t)dX(t) = \lim_{n\to\infty} \sum_{j=0}^{n-1} (1/2)[S(jt/n) + S((j+1)t/n)]dX(jt/n) \quad ? \tag{14}$$

We have discussed such choices numerically in the context of quadrature, but mathematically we have always assumed them equivalent in the limit. For stochastic variables, however, they lead to *different* results. The second choice is due to Stratonovich, the first to Itô; see the reference of footnote 7. In (14), the instantaneous change in the asset S is determined (*via* (12)) in part by its future value. To avoid such a non-physical (or advanced) interpretation, we use (13) exclusively.

To make use of equations such as (1), we require the transformation laws that replace the chain rule of classical calculus. A proper treatment begins from one of two starting points. The first very carefully Taylor expands integrals of the form (13); see the book by Wilmott *et al.* in the references. The second has a very nice graphical interpretation. Although dX is problematic, the probability density $P(S,t)$ is well behaved: at any given time t, the integral $\int_a^b P(S,t)dS$ gives the probability that the asset will have a price S between limits a and b. Now we usually start out knowing exactly the price $S(0)$ (because $t = 0$ is now and we can check on the World-Wide Web). Thus $P(S,0) = \delta(S - S(0))$. The further we go into the future, the fuzzier the estimate. If you will imagine plotting a three-dimensional graph with time as the horizontal axis and price S as the vertical, and let $P(S,t)$ be the height out of the page, we have a delta function at $t = 0$, which diffuses (widens), drifts, and flattens as we progress to the right (t increasing). The differential equation for $P(S,t)$ given (12) is called a Fokker-Planck equation and is a special instance of a more general master equation.

Given the convention (13), one can show the non-linear stochastic differential equation

$$\dot{y} = A(y) + C(y)L(t) \tag{15}$$

equivalent to the Fokker-Planck equation

$$\frac{\partial P(y,t)}{\partial t} = -\frac{\partial}{\partial y}\Big(A(y)P(y,t)\Big) + \frac{1}{2}\frac{\partial^2}{\partial y^2}\Big(C(y)^2 P(y,t)\Big) \quad . \tag{16}$$

For details, refer to the reference in footnote 7 or to Doob, *Stochastic Processes*, Wiley, 1953.

Since all the derivatives in (16) are well-defined, we can apply the chain rule to *them* for a function $\bar{y} = \phi(y)$ to find $\dot{\bar{y}} = \bar{A}(\bar{y}) + \bar{C}(\bar{y})L(t)$ with

$$\bar{A}(\bar{y}) = \phi' A(y) + \frac{1}{2}\phi'' C(y)^2 \quad ; \quad \bar{C}(\bar{y}) = \phi' C(y) \quad . \tag{17}$$

This is called Itô's lemma. Applied to (1) and generalized to a function $f(S,t)$ that depends on t explicitly as well as through $S(t)$, it states

$$df = \left(\mu S\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} + \frac{\partial f}{\partial t}\right)dt + \sigma S\frac{\partial f}{\partial S}dX \quad ; \tag{18}$$

as a particular example, take $f(S) = \ln S$. Then df obeys the equation

$$df = (\mu - \frac{1}{2}\sigma^2)dt + \sigma dX \quad . \tag{19}$$

This is simpler than (1), stating that the logarithmic asset price, f, follows a normally distributed, biased random walk. For this reason, S in (1) is called a log-normal walk.

Note that (19) is *not* what one would get naively. For example, if we were allowed to write $\dot{f} = \dot{S}/S$ (we're not), we could divide both sides of (12) by S and multiply by dt to get

$$df \overset{\text{wrong}}{=} \mu dt + \sigma dX \quad , \tag{20}$$

differing from (19) by the absence of the extra drift piece $\frac{1}{2}\sigma^2$. At first this seems paradoxical: setting $\mu = 0$, we see from (19) that, on average, $f = \ln S$ *decreases* with time. (Don't worry about this as a model of stock price: we can always restore the positive μ.) However, doesn't (1) (or (12)) appear to be neutral with $\mu = 0$? No: consider the discrete-time version. Imagine we start with \$1 and that the first two random noises are $\delta X_1 = +.1$ and

$\delta X_2 = -.1$. We end up with $0.9 \cdot 1.1 \cdot \$1 = \$0.99$. Conversely, if $\delta X_1 = -.1$ and $\delta X_2 = +.1$, we again hold at the end $1.1 \cdot 0.9 \cdot \$1 = \$0.99$ (obviously the same, since, as Tom Lehrer sings, multiplication is commutatitve). Clearly the slow average drift continues to hold for any number of multiplied factors so long as we obey the choice (13); even in the limit, the unbiased dX leads to a biased df. This helps make the result (19), and the need for (18), more plausible.[8]

Black-Scholes

We're now ready to apply an arbitrage argument to the pricing of options. We require one result from the previous section, (18), which replaces the chain rule of ordinary calculus when applied to the specific equation (1).

The price $V(S,t)$ of a put or call option (consider European options at first) plausibly depends on its exercise price, the time $T-t$ to expiration, the price $S(t)$ of the underlying asset, and the model parameters μ and σ that describe the (expected, or guessed) long-time drift of the stock and the strength of stochastic noise. I'll delay consideration of a dividend yield on the stock.

We now construct a simpler portfolio than (4): we purchase some derivative V (put or call) and sell short a relative fraction of shares Δ in the underlying, so that the entire portfolio is

$$\Pi = V - S\Delta \quad . \tag{21}$$

Linear operations on stochastic variables are safe, so we can also write $d\Pi = dV - \Delta dS$, assuming we hold Δ constant. Engaging in the arbitrage technique (21) is called "Delta hedging;" in practice, the arbitrager has to re-calculate and change Δ quite often. In three lines, we'll decide what Δ should be to yield a portfolio instantaneously without risk.

Itô's lemma (18) tells us

$$dV = \left(\mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t}\right)dt + \sigma S\frac{\partial V}{\partial S}dX \quad , \tag{22}$$

so that the differential portfolio value is given by

$$d\Pi = \left(\sigma S\frac{\partial V}{\partial S} - \Delta\sigma S\right)dX + \left(\mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t} - \mu S\Delta\right)dt \quad . \tag{23}$$

(I've plugged in (1).) By setting $\Delta = \partial V/\partial S$ (evaluated at the current time and price), I can eliminate the entire stochastic term (dX), yielding a *completely deterministic* equation:

$$d\Pi = \left(\frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t}\right)dt \quad . \tag{24}$$

Because we know (within the model) exactly what Π will be worth in a short time, the portfolio carries no risk. Appealing again to the principle that all risk-free investments must bear the same yield (else make George Soros very wealthy), we decide that

$$d\Pi = r\Pi dt \quad , \tag{25}$$

[8] It's unfortunate the financial people didn't write

$$dS(t) = (\mu + \frac{1}{2}\sigma^2)S(t) + \sigma S(t)dX(t)$$

instead of (1), but one can always think of the drift absorbing the extra term.

where r is the rate on short-term United-States Treasury bills. Combining (21), (24), and (25), we get

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \quad , \tag{26}$$

the Black-Scholes equation.

Not only has the choice of Δ between (23) and (24) eliminated randomness: it's also got rid of the drift term, μ. Amazingly, while the price of an option may depend on many things, it does not depend on whether the overall trend of the underlying stock is up or down. This contradicts common sense, since if a stock (say AMZN) falls, puts will increase in value and calls fall. For someone who holds on to an option for a long time (or until expiration), μ certainly *does* matter. So why is it absent in the Black-Scholes equation?

The first answer is that the Black-Scholes analysis involves only an instant of time. From our review of random walks, in which $dX \sim \sqrt{dt}$, it is clear that for short enough time scales, the stochastic term in dS, proportional to dX, completely dominates the drift term proportional to dt, so μ doesn't matter. A second answer is that Black-Scholes argues from arbitrage. Since *any* arbitrage opportunity yields infinite profits, at least in the model, it mustn't exist. An infinite constraint overcomes other influences. Finally, the model (1) fits stock prices moderately well *after the fact*, and one may even be able to estimate σ in advance, if it doesn't change too quickly (we've assumed it doesn't change at all). However, if anyone could predict μ accurately, she'd have a free-money machine quite independent of options.

Note that Black-Scholes uses a self-consistent argument: we assume we *can* calculate the value $V(S,t)$ of an option and its derivatives in order to derive an equation that *does* let us calculate them. The argument most likely needs some modification in the real world, particularly when the options markets have limited liquidity (few people interested in buying or selling).

Any physicist or applied mathematician should be able to find numerous extensions to the Black-Scholes model, and a few appear in the books cited in the reference section. However, here is one important difference between finance and physics: if you come up with a really good improvement, don't publish! Unlike stocks, options are a zero-sum game: for every dollar someone makes, someone else loses a dollar. So as soon as everyone else catches on to your new trick, it ceases to be useful.

One improvement I'll quote (see Wilmott for the derivation): if a stock pays a dividend at rate D_0, and if we imagine that the company disburses the dividend continuously instead of in one lump each quarter (more spherical horses), the modified Black-Scholes equation reads

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - D_0)S\frac{\partial V}{\partial S} - rV = 0 \quad . \tag{27}$$

American options and free boundary conditions

For European options, Black-Scholes has a closed-form solution; see page 100 in Wilmott *et al.* The early-exercise feature of American puts and calls, however, ensures a minimum value for in-the-money options. As an extreme example, consider a stock with no volatility and no drift (maybe it's been nationalized by an evil government, which promises to buy and sell at a fixed price). A European put option at $E > S$ (now S is fixed for all time) is clearly worth money, but it's worth less than $E - S$, because it can't be exercised for a time T. Specifically, right now (time 0), it's worth $(E - S)e^{-rT}$. However, the same option with American rules is worth $E - S$, since the holder can exercise it immediately. (It isn't worth any more than $E - S$ if the price of S truly is fixed for all time; this is of course an artificial example.)

So long as they're worth *more* than $E - S$ (put, or $S - E$, call), American options are subject to the same arguments that led to Black-Scholes, (27). However, whenever an in-the-money option would otherwise fall below $|E - S|$, that "payoff" function provides the floor for the option.

This does *not* mean that $C_{\text{American}} = \max(C_{\text{European}}, S - E)$ (call) or that $P_{\text{American}} = \max(P_{\text{European}}, E - S)$ (put). This is because a partial differential equation, such as Black-Scholes, determines the value of a function at some point in space and time based on the value of the function at neighboring points of space and time. However, these points affect their neighbors, *et cetera*, out to all regions of the time-price plane. Thus the conditions (I'll drop geographic subscripts and assume everything is American)

$$C(S,t) \geq \max(S - E, 0)$$

and

$$P(S,t) \geq \max(E - S, 0) \tag{28}$$

affect the solution globally, just as the initial value determines the solution globally in an ordinary differential equation, and just as the boundary values determine the solution to Laplaces's equation in electrostatics. However, the boundary conditions (28) suffer from one complication: *we don't know in advance where in (S,t) they apply.* Such conditions, for which we must solve as we solve the differential equation, are called free boundaries. A closely related physical example is Stefan's problem of the propagation of the solid-liquid front in melting ice.[9]

Wilmott *et al.* suggest a simple example of a free-boundary problem: consider a taut string tied firmly to the x axis at points ± 1 that must pass over an obstacle $f(x) = \max(1 - (2x + 0.5)^2, 0) + \max(0.5 - 0.5(2x - .8)^2, 0)$ (see figure 1).

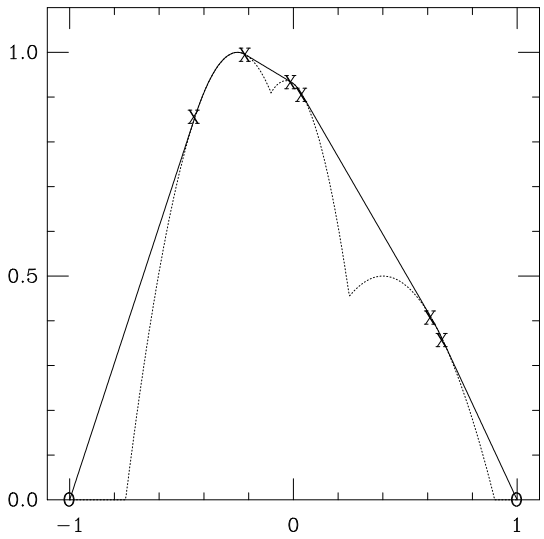


Figure 1. The obstacle problem. The string is solid, the obstacle dotted. "X" mark the free boundaries, between which the string does not bend.

Letting $y(x)$ be the string's displacement at x, tautness means $y'' = 0$ except where the string makes contact with the obstacle, where $y(x) = f(x)$. The string cannot pass through the obstacle,

[9] See Crank, *Free and Moving Boundary Problems*, Oxford, 1984 or Guenther and Lee, *Partial Differential Equations of Mathematical Physics and Integral Equations*, Prentice-Hall 1988, Dover 1996.

so we can also write $y(x) \geq f(x)$. One particularly elegant solution to the problem combines the differential equation with the constraint to write

$$y''(y - f) = 0 \quad , \tag{29}$$

accompanied by the additional conditions $y'' \geq 0$ and $y - f \geq 0$ and the fixed boundary conditions $y(-1) = y(0) = 0$. Tautness also requires continuity of y'.[10] Since the possibility of early exercise of an American-style option turns Black-Scholes into a similar inequality, we can formulate our current problem similarly.

Scaling the differential inequality

It is a familiar theme in theoretical physics to rescale dynamical variables into dimensionless forms independent of irrelevant measuring conventions. In the Lennard-Jones problem, once we scaled the radial distance by the Lennard-Jones length σ, the only physical length in the problem, it no longer mattered whether we measured distance in meters, yards, or parsecs. Similarly, a stock split[11] should not change the scaled variables.

In finance, money plays the role of distance.[12] The only known length (\$) scale in the problem is the exercise price, E, of the option. Furthermore, we should take the logarithm of stock price S; our model (1) of noise affects prices proportionately, not absolutely, so we exercise a random walk of scale independent of stock price only when we take the logarithm, as in (19), defining a scaled asset price x by

$$\begin{aligned} S(t) &= Ee^x \\ x &= \ln(S/E) \quad . \end{aligned} \tag{30}$$

Note that the transformation preserves the sense of stock price increasing or decreasing and that it maps the interval $(0, \infty)$ to $(-\infty, \infty)$.

The important dynamical time in the problem is time remaining until expiration of the option, $T - t$. Interest rate, dividend yield, and volatility all set time scales. (So does drift μ, but it plays no role in the Black-Scholes equation (27).) Note from (1) or from (11) that σ has units of $(\text{time})^{-1/2}$; an additional factor of two turns out to be convenient, so we write

$$t = T - 2\tau/\sigma^2 \quad . \tag{31}$$

Finally, we scale the option price V by the only length scale in the problem, E, so

$$V(S, t) = Ev(x, \tau) \quad . \tag{32}$$

In the variables transformed by (30)–(32), Black-Scholes (27) appears as

$$\frac{\partial v}{\partial \tau} = \frac{\partial^2 v}{\partial x^2} + \frac{\partial v}{\partial x}(k_2 - 1) - k_1 v \quad , \tag{33}$$

[10] Except if the obstacle has a sharp edge, which possibility we exclude to avoid cutting our string. Without the continuity condition, we could propose $y(x) = \max(0, f(x))$, which obviously violates tautness.

[11] A company may declare a stock split, at which (typically) every share of outstanding stock is replaced by two. The market value per share immediately falls in half, and option exercise prices are automatically adjusted. Companies sometimes do this if their stock price has risen so much that the average investor can no longer afford an even lot of 100 shares. More often, the motivation is purely psychological.

[12] This contradicts the folk wisdom that "time is money." In fact, since $dX \sim \sqrt{dt}$, one might more properly say "time is money squared."

where

$$\begin{aligned} k_1 &= 2r/\sigma^2 \\ \text{and} \quad k_2 &= 2(r - D_0)/\sigma^2 \end{aligned} \tag{34}$$

are dimensionless constants describing interest and dividend rates in terms of the time scale set by volatility.[13]

Equation (33) resembles a diffusion equation with additional source terms. We can make these vanish by one additional transformation of the scaled option value v: plugging

$$v(x,\tau) = e^{\alpha x + \beta \tau} u(x,\tau) \tag{35}$$

in to (33), with

$$\begin{aligned} \alpha &= -\frac{1}{2}(k_2 - 1) \\ \text{and} \quad \beta &= -(\frac{1}{4}(k_2 - 1)^2 + k_1) \quad , \end{aligned} \tag{36}$$

we get the pure diffusion equation

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2} \quad . \tag{37}$$

Since the early-exercise feature of an American option can only make the value higher, never lower, with time than it would be otherwise, we should really write

$$\frac{\partial u}{\partial \tau} \geq \frac{\partial^2 u}{\partial x^2} \quad . \tag{38}$$

To set boundary conditions, first note that the option value at expiration, $\tau = 0$, is the payoff $\max(S - E, 0)$ for the call or $\max(E - S, 0)$ for the put. Transformed into dimensionless variables, this takes the less pleasing form

$$g(x,\tau) = \begin{cases} e^{-\alpha x - \beta \tau} \max(e^x - 1, 0) & \text{call} \\ e^{-\alpha x - \beta \tau} \max(1 - e^x, 0) & \text{put.} \end{cases} \tag{39}$$

The transformation (35) introduced the explicit time variable, τ, which we can set equal to zero for the "final" condition analagous to the initial condition we studied for ordinary differential equations. We leave it in (39), because the free boundary condition for American options takes the identical form evaluated at (backward) time $\tau > 0$. Thus both the free and the initial boundary conditions are expressed by (39) and

$$u(x,\tau) \geq g(x,\tau) \quad . \tag{40}$$

We know two other fixed boundaries: if the stock price S falls to zero, the holder should certainly exercise a put, since the stock cannot fall below zero. Thus $P(0,t) = E$. Similarly, in the limit of infinite stock price, the holder of a call should exercise: $\lim_{S\to\infty} C(S,t) = S - E$. Conversely, the put becomes worthless in the limit $S \to \infty$, and the call worthless if the company ever declares bankruptcy.[14] Expressed in dimensionless variables, for either type of option

$$\lim_{x \to \pm\infty} u(x,\tau) = \lim_{x \to \pm\infty} g(x,\tau) \quad . \tag{41}$$

[13] I borrow the notation used by Wilmott *et al.*

[14] The model (1) does not permit the stock ever to recover from $S = 0$.

In addition, there is a somewhat subtle condition of continuity. I refer the reader to Wilmott *et al.*, who demonstrate an arbitrage opportunity at the free boundary (the point on one side of which $V(S(t),t)$ satisfies (28) as an equality) unless $\partial V/\partial S$ is continuous across the boundary and so equal to -1. For the payoff functions of our model (the right-hand sides of (28)), $\partial V/\partial S$, and hence $\partial u/\partial x$, must simply be continuous everywhere.[15]

By analogy to the obstacle problem, (29), we express the differential inequality with free boundaries as

$$\left(\frac{\partial u}{\partial \tau} - \frac{\partial^2 u}{\partial x^2}\right)\Big(u(x,\tau) - g(x,\tau)\Big) = 0 \tag{42}$$

with the conditions (38), (40), and (41).

In the context of Brownian motion, we should contrast the diffusion equation for option value, (37), with that eventually derivable for the probability distribution of underlying asset price S from the Fokker-Planck equation (16), for while the boundary conditions for $u(x,\tau)$ in the price-time plane are completely known, (16) instead describes the diffusive fuzziness of our future knowledge of stock prices, and P can be reset to a Dirac-δ function every time we check the actual market.

Discretization and stability

To solve a partial differential equation such as (37) or (42), we must first discretize space x and time τ and the spatial and temporal derivatives of u. Depending on the method, we may need additionally to solve a system of linear equations.

We first consider numerical solution of (37) for European options and thus fixed boundary conditions. The discretization scheme, if not the solution method, will carry straightforwardly to the American case. For any method, we must obviously replace the continuous variables x and τ with discrete versions, $x = n\delta x$, $\tau = m\delta\tau$. The goal is to find the scaled option prices $u_n^m = u(n\delta x, m\delta\tau)$ on the grid of integers (m,n).

In numerically approximating a first derivative, we have the three obvious choices of figure 2 (see *Numerical Recipes*, §5.7 for details):

$$\frac{\partial u}{\partial \tau}(x,\tau) \approx \begin{cases} \dfrac{u(x,\tau+\delta\tau)-u(x,\tau)}{\delta\tau} + \mathcal{O}(\delta\tau) & \text{forward difference} \\ \dfrac{u(x,\tau)-u(x,\tau-\delta\tau)}{\delta\tau} + \mathcal{O}(\delta\tau) & \text{backward difference} \\ \dfrac{u(x,\tau+\delta\tau/2)-u(x,\tau-\delta\tau/2)}{\delta\tau} + \mathcal{O}((\delta\tau)^2) & \text{central difference.} \end{cases} \tag{43}$$

One normally prefers the central difference for analyzing numerical data.[16] For reasons of numerical stability, sometimes the forward or backward difference may be better or more applicable in the solution of partial differential equations.

To approximate the second spatial derivative in (37), we start with the central difference $\frac{\partial u}{\partial x}$ and apply a central difference to that, giving a symmetric central second derivative,

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\delta x,\tau) - 2u(x,\tau) + u(x-\delta x,\tau)}{\delta x^2} + \mathcal{O}(\delta x^2) \quad . \tag{44}$$

[15] Other models may introduce discontinuities in the derivative of V if they have discontinuities, other than at $S = E$, in the derivative of the payoff function.

[16] One of my current projects has me estimating a numerical derivative evaluated in the limit $\tau \to 0$. On this particular project, there is a kink exactly at $\tau = 0$, but the derivative exists everywhere else. Since I cannot straddle $\tau = 0$, I'm stuck extrapolating a limiting series of forward differences.

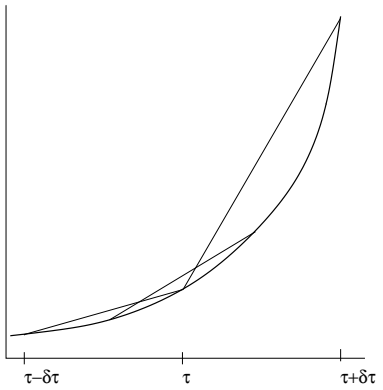


Figure 2. Comparison of three methods for numerically calculating the derivative with the same $\delta\tau$ (horizontal spread): the left chord represents a backward difference, the right chord a forward difference, and the chord straddling the point in question the central difference. Clearly the last most closely approximates the derivative.

(I again refer the reader to *Numerical Recipes* for an analysis of the order of accuracy.)

Perhaps the simplest method for integrating the diffusion equation sets the forward difference for time (43) equal to the central spatial difference (44), yielding (with the notation $u_n^m = u(n\delta x, m\delta\tau)$)

$$\frac{u_n^{m+1} - u_n^m}{\delta\tau} = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{\delta x^2} \quad , \tag{45}$$

or

$$u_n^{m+1} = u_n^m + \alpha(u_{n+1}^m - 2u_n^m + u_{n-1}^m) \quad , \tag{46}$$

where the parameter $\alpha = \delta\tau/(\delta x)^2$ measures how long a time step we're taking for a given spatial step. Subscripts are price, superscripts time, so (46) represents an *explicit* formula for $u(x, \tau + \delta\tau)$ given $u(y, \tau)$ for three separate prices y. Figure 3a illustrates the method.

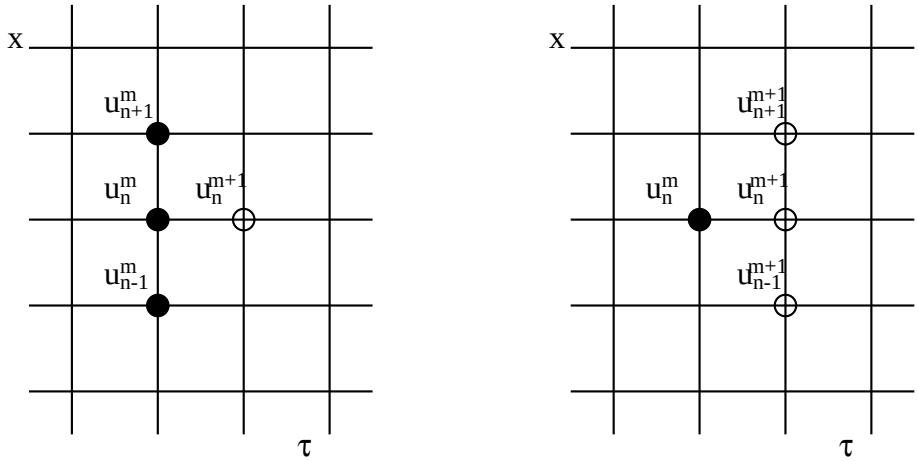


Figure 3. (a) The explicit method solves for u_n^{m+1} (open circle) in terms of the already-known points u_{n-1}^m, u_n^m, and u_{n+1}^m (closed circles). The horizontal axis is scaled backward time, the vertical axis scaled logarithmic option value ("space"). This method can be unstable, depending on the grid spacing. (b) The implicit method solves for u_{n-1}^{m+1}, u_n^{m+1}, and u_{n+1}^{m+1} in terms of the already-known point, u_n^m. This method is stable.

As we saw with the Euler method for ordinary differential equations, explicit methods can suffer from instabilities, although in this case of a somewhat different nature. I follow the von-Neumann stability analysis from *Numerical Recipes*, §19.2. Imagine that roundoff has resulted in

a small error in u, so that we calculate not u, but

$$\hat{u}_n^m = u_n^m + E_n^m \quad , \tag{47}$$

where I can expand the noise E^m at time $m\delta\tau$ in its spatial Fourier components,

$$E_n^m = \sum_k E^m(k) e^{ikn} \quad . \tag{48}$$

The correct u is certainly a solution of (46), as is the noisy solution (47) by construction. By linearity so is the noise or any one of its Fourier components. Thus I can analyze the noise one wavenumber k at a time.

Von Neumann asks whether a particular noise component will tend to be amplified or tend to be damped out as a function of time. In the former case, the method is unstable. Plugging a single term from the right-hand side of (48) in to (46), I find

$$E^{m+1}(k) = E^m(k)(1 + 2\alpha[\cos k - 1]) \quad . \tag{49}$$

The term in parentheses represents the factor by which the k^{th} Fourier component of error increases at each time step. Its absolute value had better be less than unity for *all* non-zero values of k, for which I require (for positive α)

$$0 < \alpha < 1/2 \quad . \tag{50}$$

This means that once we have chosen a spatial step size δx, we are obliged to pick a sufficiently small $\delta\tau$, or the error terms will quickly blow up. Small $\delta\tau$ is desirable anyway from the point of view of accuracy, but it can also slow the calculation considerably. The refinement developed below allows larger time steps, speeding the calculation, while simultaneously improving accuracy.

As with ordinary differential equations, we are led to an implicit method. Replacing the left-hand side of (45) with the backward time difference, I find instead of (46)

$$(1 + 2\alpha)u_n^{m+1} - \alpha u_{n+1}^{m+1} - \alpha u_{n-1}^{m+1} = u_n^m \quad . \tag{51}$$

Repeating the analysis that led to (49), I compute

$$E^{m+1} = (1 + 2\alpha(1 - \cos k))^{-1} E^m \quad . \tag{52}$$

The coefficient multiplying the error is now less than 1 in magnitude for all k (except 0, for which it is marginal) and all $\alpha > 0$.[17]

As figure 3b illustrates, a single known point at time m sets a relation among three unknown points at time $m + 1$. If we think of $\mathbf{u}^m$ at fixed time as a vector whose components are the scaled values of the option at that time for the various scaled strike prices, equation (51) takes the form of a linear system of equations,

$$\mathrm{T}\mathbf{u}^{m+1} = \mathbf{u}^m + \mathbf{b}^m \quad , \tag{53}$$

where T is a symmetric, tridiagonal "Toeplitz" matrix with entries $1 + 2\alpha$ all along the diagonal and $-\alpha$ all along the two subdiagonals. (Toeplitz just means that the entries in each subdiagonal are all equal: $T_{i,i+j} = T_{i+1,i+j+1}$ within the range of indices.) The vector $\mathbf{b}^m$ wouldn't be necessary

[17] I needn't worry about the zero-wavenumber (infinite-wavelength) error mode, which the boundary conditions at $x \to \pm\infty$ suppress.

if our system were infinite (in "space" x). However, since we have to stop for some large spatial subscripts $n = \pm N$, we absorb the boundary conditions into $\mathbf{b}$:[18]

$$\begin{aligned} b^m{}_N &= \alpha u^m{}_{N+1} \\ b^m_{-N} &= \alpha u^m_{-N-1} \\ b^m_j &= 0 \quad , \; j \neq \pm(N+1). \end{aligned} \tag{54}$$

We can never reach the limits $x = n\delta x \to \pm\infty$ in (41); instead, we go as far out as we can afford (in computer time or memory), and approximate $b_{\pm N} = u_{\pm(N+1)} \approx g(\pm(N+1)\delta x) \equiv g_{\pm(N+1)}$ (compare equation (41), where u isn't set equal to g until $\pm\infty$, although the free boundary may *make* it equal for finite x).

The matrix T is very easily inverted[19], letting us write $\mathbf{u}^{m+1} = T^{-1}(\mathbf{u}^m + \mathbf{b}^m)$. Since the matrix is the same at every time step, we need perform the inversion but once. However, while the sparse matrix T takes up practically no memory storage, its inverse T^{-1} is dense. Thus if we have many spatial steps, it may be preferable to use one of the linear-equations algorithms described in *Numerical Recipes* chapter 2 or in Wilmott *et al.* or in Golub and van Loan.

The discussion of the fully explicit and fully implicit methods sets the stage for the Crank-Nicolson discretization, which is simply the average of (46) and (51):

$$(1+\alpha)u^{m+1}_n - \frac{\alpha}{2}(u^{m+1}_{n+1} + u^{m+1}_{n-1}) = (1-\alpha)u^m_n + \frac{\alpha}{2}(u^m_{n+1} + u^m_{n-1}) \quad . \tag{55}$$

This equation can be cast in the form

$$\mathrm{C}\mathbf{u}^{m+1} = \mathrm{D}\mathbf{u}^m + \mathbf{b}^m \tag{56}$$

for tridiagonal Toeplitz matrices C and D and a suitable definition of $\mathbf{b}$ and solved as before (see Wilmott *et al.* for details). While no more difficult than the implicit method, Crank-Nicolson achieves superior convergence by averaging the forward and backward time differences to achieve a central difference for the time derivative.

Methods for free-boundary problems

The methods of the previous section all work on problems with fixed boundary conditions. To quote Wilmott *et al.*, "the chief problem with free boundaries, from the point of view of numerical analysis, is that we do not know where they are." Therefore, we can't apply them until we've found them. This sits well enough with the explicit method, (46)—one simply replaces locally a solution violating (40) with (39)—but there's no obvious way of doing this with the two implicit methods, (53) and (56).

Our reference (Wilmott *et al.*; see also Crank from footnote 9) recommends an iterative approach, the Projected Successive Over-Relaxation method (PSOR), which can be shown to converge to the correct result in the presence of free boundaries.

Just as the previous section needed to consider first one algorithm, then its antithesis, and finally their synthesis in order to make sense of the final result, so in order to explain PSOR must we first understand Jacobi's iterative method, then its refinement, Gauss-Seidel relaxation, in order to combine them into the Successive Over-Relaxation (SOR) algorithm. PSOR is a slight modification of SOR to take account of the condition (40).

[18] Alternatively, we could place non-zero terms in the lower-left and upper right corners of the matrix T. However, this would destroy most of its wonderful tridiagonal properties.

[19] See Golub and van Loan, *Matrix Computations*, 3rd edition, Johns Hopkins, 1996, §4.7.4.

Discretization of Relation (42)

In the solution of equation (42), the accompanying conditions, (40), (41), and the continuity requirement $\partial V/\partial S = -1$, are as essential as the analogous conditions were to the obstacle problem of figure 1. The proposed solution algorithm must therefore not only solve (42) but also must satisfy the conditions. I will not prove that the following method accomplishes this, but the fact is easy enough to verify numerically. (See Crank for a proof.)

I follow Wilmott (chapter 21) in applying Crank-Nicolson discretization (55) to the first set of parentheses in (42). The second set of parentheses evaluated at time $\tau = (m+1)\delta\tau$ is represented simply by $u_n^{m+1} - g_n^{m+1}$. Collecting all the time-m terms from the right-hand side of (55) in a vector

$$b_n^m = (1-\alpha)u_n^m + \frac{\alpha}{2}(u_{n+1}^m + u_{n-1}^m) \quad , \tag{57}$$

we have for the approximation of (42)

$$\left(u_n^{m+1} - \frac{\alpha}{2}(u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}) - b_n^m\right)(u_n^{m+1} - g_n^{m+1}) = 0 \quad . \tag{58}$$

As with the example of the implicit method (see (54)), the general form of (57) must differ for the extreme spatial indices n. Here, we take n to range from $-N+1$ to $N-1$; the extremal values replacing (57) are

$$b_{\pm(N-1)}^m = u_{\pm(N-1)}^m + \frac{\alpha}{2}(g_{\pm N}^m - 2u_{\pm(N-1)}^m + u_{\pm(N-2)}^m + g_{\pm N}^{m+1}) \quad . \tag{59}$$

This defines b at time m partly in terms of g at time $m+1$, but it presents no difficulty, as the scaled payoff function g is known from (39) in advance for all times.

In terms of the Toeplitz matrix

$$\mathrm{C} = \begin{pmatrix} 1+\alpha & -\alpha/2 & 0 & \dots & 0 \\ -\alpha/2 & 1+\alpha & -\alpha/2 & & 0 \\ 0 & -\alpha/2 & 1+\alpha & \ddots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & & -\alpha/2 & 1+\alpha & -\alpha/2 \\ 0 & & 0 & -\alpha/2 & 1+\alpha \end{pmatrix} \tag{60}$$

with diagonal $1+\alpha$ and $-\alpha/2$ subdiagonals, (58) becomes

$$(\mathbf{u}^{m+1} - \mathbf{g}^{m+1}) \cdot (\mathrm{C}\mathbf{u}^{m+1} - \mathbf{b}^m) = 0 \quad . \tag{61}$$

The boundary conditions are succinctly expressed by the requirement that no component of either of the two orthogonal vector factors on the left-hand side of (61) may go negative.

At the beginning of each time step m, we know the value of $\mathbf{u}^m$, and *via* (57) and (59) that of $\mathbf{b}^m$. Of course we also know the payoff $\mathbf{g}^{m+1}$; there remains only to solve (61) for $\mathbf{u}^{m+1}$.

Iterative solutions for linear systems

Solutions of linear-algebra problems such as (53), (56), and (61) fall into two general categories. The more familiar algorithms, such as, for example, the technique one would use to solve $\mathrm{A}\mathbf{u} = \mathbf{b}$

with pen and paper for small A (say 2×2), are deterministic: one knows in advance exactly how many steps are required for a solution, and that solution is exact except for roundoff error.[20]

Alternatively, one can solve systems of equations iteratively. Beginning with a *guess* for the solution $\mathbf{u}$, one applies some procedure that refines the guess. One repeats the process with the refinement from the previous step as the guess for the next step until $\mathbf{u}$ stops changing appreciably. Of course, we employed iteration in the zero-finding subroutine for the first mechanics project. One would like some theoretical assurance first that the sequence of guesses converges and second that it converges to the right answer.[21]

Jacobi's method and the Gauss-Seidel method (Numerical Recipes §19.5) solve the linear system

$$\mathrm{A}\mathbf{u} = \mathbf{b} \tag{62}$$

for $\mathbf{u}$ by first writing

$$\mathrm{A} = \mathrm{L} + \mathrm{D} + \mathrm{R} \quad , \tag{63}$$

where D is diagonal, L has non-zero components only below the diagonal (*i.e.*, lower-left, or column index less than row index), and R has non-zero components only above the diagonal. Jacobi notes that D is very easy to invert, while L+R is not so easy. Specifically, the inverse of a diagonal matrix is another diagonal matrix whose entries are the inverses of the original diagonal, $(\mathrm{D}^{-1})_{ii} = (\mathrm{D}_{ii})^{-1}$. Rearranging some terms in (62) gives

$$\mathbf{u} = \mathrm{D}^{-1}\mathbf{b} - \mathrm{J}\mathbf{u} \quad , \tag{64}$$

where the Jacobi matrix

$$\mathrm{J} = \mathrm{D}^{-1}(\mathrm{A} - \mathrm{D}) \quad . \tag{65}$$

This inspires an iterative scheme wherein the $k+1^{\mathrm{st}}$ estimate of $\mathbf{u}$ is given in terms of the k^{th} by

$$\mathbf{u}^{(k+1)} = \mathrm{D}^{-1}\mathbf{b} - \mathrm{J}\mathbf{u}^{(k)} \quad . \tag{66}$$

The initial guess $\mathbf{u}^{(0)}$ hardly matters at all. By an argument similar to von Neumann's earlier, (66) converges if and only if all eigenvalues of J fall inside the unit circle in the complex plane. The largest (in magnitude) such eigenvalue, λ_J, determines how quickly $\mathbf{u}$ converges to the right answer.

Jacobi's method converges too slowly for most purposes; Gauss-Seidel is a variant that treats the lower-triangular matrix $L + D$ as "easy." Lower-triangular matrices are nearly as easy as diagonal ones. Let B be a lower-triangular matrix; we wish to solve Bu = b. The first row of the matrix equation reads $B_{11}u_1 = b_1$, which immediately gives us $u_1 = b_1/B_{11}$. The second row reads $B_{21}u_1 + B_{22}u_2 = b_2$. Since we know u_1, we immediately get u_2, and so on. Thus one iterates

$$(\mathrm{L} + \mathrm{D})\mathbf{u}^{(k+1)} = \mathbf{b} - \mathrm{R}\mathbf{u}^{(k)} \quad . \tag{67}$$

Gauss-Seidel also converges slowly, but the clever numerical analysts have shown that the following can converge very rapidly for the right choice of $1 < \omega < 2$:

$$\mathbf{u}_{\mathrm{SOR}}^{(k+1)} = \omega\mathbf{u}_{\mathrm{GS}}^{(k+1)} + (1-\omega)\mathbf{u}_{\mathrm{SOR}}^{(k)} \quad . \tag{68}$$

[20] The roundoff error isn't always small in the most straightforward treatment of the linear system $\mathrm{A}\mathbf{u} = \mathbf{b}$; see the discussions in *Numerical Recipes* about matrix conditioning.

[21] I refer the curious reader to the books already mentioned, especially Golub and van Loan for the unrestricted problems and Crank for (61) with its boundary conditions.

Here GS stands for "Gauss-Seidel" and SOR for "successive overrelaxation:" if we think of Gauss-Seidel as "relaxing" $\mathbf{u}$ from the guess toward the right answer, any $0 < \omega < 1$ in (68) will cause the solution to "relax" less rapidly. Thus $1 < \omega < 2$ "overrelaxes."

It can be shown that the optimal value for ω in the long-time limit is given in terms of the dominant eigenvalue of the Jacobi matrix by $\omega = 2/(1 + \sqrt{1 - \lambda_J^2})$. As the authors of *Numerical Recipes* point out, the calculation of λ_J is usually more involved than the solution of (62), so it's difficult to apply this optimization. However, for our particular matrix (60), the largest eigenvalue of J is easily calculated, at least in the limit of infinite size of the matrix: $\lambda_J = \alpha/(1 + \alpha)$. Although *Numerical Recipes* recommend an iterative scheme for ω called Chebyshev acceleration, their analysis applies only to SOR, not to the projected SOR we consider next, and I got better results using the asymptotically optimal ω throughout.

Projected Successive Overrelaxation

SOR solves the system (56), but the constrained system (61) requires only one additional step in the algorithm: we replace (68) with

$$\mathbf{u}_{\mathrm{SOR}}^{(k+1)} = \max(\mathbf{g},\ \omega \mathbf{u}_{\mathrm{GS}}^{(k+1)} + (1 - \omega)\mathbf{u}_{\mathrm{SOR}}^{(k)}) \quad , \tag{69}$$

with the maximum applied component-wise. I have suppressed the time indices (too many superscripts!), evaluating $\mathbf{g}$ and $\mathbf{u}$ both at time $m + 1$ from (61) for all steps of the PSOR iteration.

Anticlimax

After all this work, the actual implementation of two subroutines to calculate the Black-Scholes value of an American option on pages 329 and 331 in Wilmott *et al.* must seem disappointingly simple. You should flesh out these two subroutines with a few subroutines of supporting code, *paying attention to the copy of the e-mail I sent to the authors*. The supporting code will have to allocate some arrays and convert between scaled and real-world variables by way of (30)–(36). Finally, embed *that* program in a larger program to calculate the volatility σ implied by a particular trading price for an option. There may be a better method, but what springs to mind is a zero-finding program for

$$V(S, t; \sigma) - V_{\mathrm{data}} = 0 \quad , \tag{70}$$

where $V(S, t; \sigma)$ is the predicted value of the option, S the current underlying stock price, t the current time, and V_{data} the actual market price of the option.

references and resources for this project

(*) indicate the most important for the course

* P. Wilmott, J. Dewynne, S. Howison, *Option Pricing: Mathematical Models and Computation*, Oxford Financial Press, Oxford, 1993.	Most of our project is based on this book, on reserve at the library.
F. Black, M. Scholes, "The Pricing of Options and Corporate Liabilities," *J. Political Econ.* **81** (1973), 637–654.	This is the Nobel-prize-winning paper that founded the field. It is available from USF IP addresses through the library's on-line journal collection.
Chicago Board Options Exchange, *Characteristics and Risks of Standardized Options*	`http://www.optionsclearing.com/publications/risks/download.jsp`
* `http://www.cboe.com`	Place to get free current and historical data on the prices of options and stocks
* `http://www.etrade.com`	Another place for free current and historical data
`http://finance.yahoo.com`	Free current and historical data on stocks (no options)
Laloux *et al.*, *Phys. Rev. Lett.* **83** (1999), 1467; Plerou *et al.*, *Phys. Rev. Lett.* **83** (1999), 1471.	Recent pair of articles on financial modeling in *Physical Review Letters*

This week's project

The average rate of return μ of a security drops out in pricing an option in the Black-Scholes model, but we still need to estimate the volatility σ. Often, one turns this around, and uses the current market premiums paid for options to estimate the market's view of future volatility on a stock. Our best estimate of volatility on a particular date comes from calculating the variance in closing share prices for some length of time (60 days, say) with the target date in the center. The past 30 days are no problem. Unfortunately, traders do not have the luxury of downloading the next 30 days of future data.

Picking times in the past, compare volatility estimates based on the trailing N days, N days centered on the date in question, and N days of future data to the volatilities implied by options trading that day. Use the resources indicated above for historical data; be sure to note that options listing 0 volume for a particular date did not trade at all, so their closing prices are meaningless. Note also that stock options in this country always expire on the third Friday of the month in question. Empirically, options of different expirations do not always imply the same volatility (see Wilmott *et al.*).

You will need to write a program to solve the Black-Scholes equation for an American option (this is thoroughly outlined in Wilmott *et al.*); depending on what you decide to emphasize, you may also find a linear-fitting program helpful (see *Numerical Recipes*). Other auxilliary programs or scripts may be useful for collecting data from Web addresses. I will provide an example program for calculating the number of days between two dates.

Physics Z-5156, Fall 2006

Chapter 7. Experimental Control and Data Acquisition with Labivew: Mechanical Feedback in a Twyman-Green Interferometer

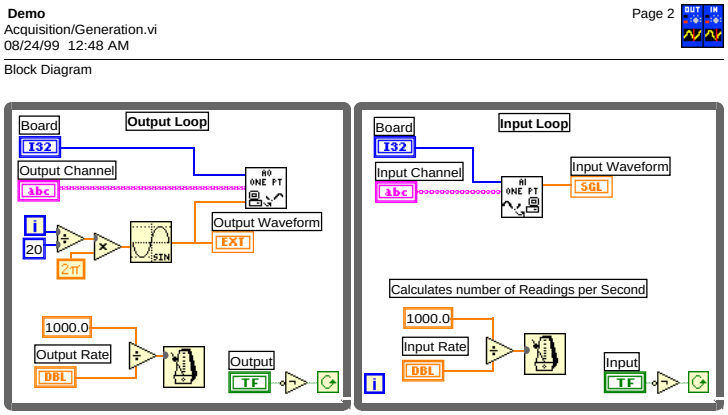
This chapter currently exists as a partial draft. I shall hand out more-complete notes as we approach the project.

references

1. Latest edition of *Labview User Manual*
2. I.J. Bigio, T.R. Gosnell, P. Mukherjee, J.D. Safer, "Microwave-Absorption Spectroscopy of DNA," *Biopolymers* **33**, 147–150 (1993).
3. N.D. Weston, P. Sakthivel, P. Mukherjee, "Ultrasensitive Spectral-Trace Detection of Individual Molecular Components in an Atmospheric Binary Mixture," *Applied Optics* **32**, 828–835 (1993).

References 2 and 3 use the Mach-Zehnder geometry; we're currently planning on using Twyman-Green.

Example of a Labview Program:



Labview is a graphical programming language for data acquisition and experimental control. We will be using the language to establish a mechanical feedback loop controlling a correction plate in an optical interferometer. The glass plate adjusts the optical path length on one arm to correct for slow fluctuations. References (2) and (3) use such an interferometer to measure changes in a gas sample; these changes occur much faster than the slow drifts we correct for with the rotating plate.

One warning about the programming metaphor in Labview: it appears on the surface to simulate a digital circuit. Timing mechanisms and loop structure, however, are much closer to those in a traditional language than to the principles of digital-circuit design.

Because we have only one interferometer and only one non-demonstration copy of the software, students will have to sign up for time this year.

Physics Z-5156, Fall 2006

Chapter 8: Models of magnetic materials: symmetry in quantum mechanics

I will provide in class further information clarifying which parts of these notes we will apply to a computing project. The written exercises and problems from these notes are for background and need not be turned in.

Interactions involving localized magnetic moments appear to play an important role in a number of new and still ill-understood materials, notably high-temperature superconductors, heavy-Fermion metals, and colossal-magnetoresistive perovskites. While one usually wishes to learn the effect of magnetism on itinerant (conduction) electrons, much can often be gained by looking at the fixed magnetic spins; including the conduction electrons generally increases the complexity enormously. We shall examine one of the simplest spin problems and briefly discuss how to extend it into current research.

A spin chain models a highly anisotropic (directional) material with a line of discretely-spaced magnetic moments, each interacting with its immediate neighbors. The magnetic moment at each site (which for convenience we'll call spin) comes from a combination of electron spin and electron orbital angular momentum; nuclear moments play no important role. When the spin is very large, for instance 15/2 in the rare-earth metals dysprosium and erbium, classical models are appropriate.[1] For small spins, such as ½ or 1, however, quantum effects dominate.

We begin with a review of spin. Because spins can take on only particular values, their quantum description is actually simpler than that of moving particles and particularly amenable to numerical treatment. The student needs, however, to absorb some amount of formalism. We then examine how spatial and spin symmetries make the system computationally tractable and finally plan our computer assault.

1. Quantum-mechanical background

Some of you may not yet have taken a full-blown quantum-mechanics course. This section aims to equip you to understand the problem of a magnetic spin chain; for derivations and a detailed understanding of the physics, see any elementary quantum-mechanics text book, e.g., Liboff §11.6 or Feynman vol. 3. Those with weaker backgrounds especially should try the exercises, solutions to which will follow.

First consider just two interacting spins of any sort (classical or quantum). The very simplest energy we can imagine is just their dot product times some constant. Calling the spins $\mathbf{S}_1$ and $\mathbf{S}_2$,

$$H = J\mathbf{S}_1 \cdot \mathbf{S}_2 = J(S_1^x S_2^x + S_1^y S_2^y + S_1^z S_2^z) \quad . \tag{1}$$

This is the Heisenberg Hamiltonian; it takes its microscopic derivation from processes such as indirect exchange, superexchange, and double exchange, depending on the system, but we shall not concern ourselves with those processes here.[2] Certainly, (1) seems intuitively plausible if whatever interaction exists between spins can be Taylor expanded. If there is to be no externally-imposed directional dependence, (1) is the first term in this expansion, aside from a constant. (Indeed, for the case of quantum spin ½, it is the *only* term.)

[1] That large angular momenta approach classical behavior is most evident in the coherent-state representation.

[2] See D. Mattis, *Magnetism Made Simple*, World Scientific, 2006 or S. Blundell, *Magnetism in Condensed Matter*, Oxford, 2001.

At zero temperature, a system contains as little energy as quantum mechanics allows; all the rest has flowed to a reservoir. Thus we try to minimize H in (1). A negative J therefore means that the spins wish to point in the same direction and so models ferromagnetism; a positive J models antiferromagnetism, in which the spins prefer to point oppositely.

Exercise. The energy function (1) is solved trivially for *classical* continuous spins at zero temperature. Convince yourself that for ferromagnetic (negative) J, the lowest-energy state has all the spins parallel, while for antiferromagnetic (positive) J, spins alternate. The classical problem starts to become interesting when we both turn on temperature and also add preferred spin directions. The quantum problem is interesting all by itself.

There are two things you need to know about quantum mechanics:

1. When we measure a spin, it is either $\uparrow$ or $\downarrow$ (Stern-Gerlach).
2. Before we measure a spin, it has some probability amplitude to be $\uparrow$ and some probability amplitude to be $\downarrow$. The probability amplitude is a complex number, the modulus squared of which gives the probability.

We represent quantum spins $\frac{1}{2}$ as vectors ("spinors") in the two-component space of $\hat{\mathbf{z}}$ projections:

$$\psi = \begin{pmatrix} a \\ b \end{pmatrix} = a\psi_\uparrow + b\psi_\downarrow \quad . \tag{2}$$

An electron in state ψ is in a superposition of states "spin up" and "spin down." If we were to measure the $\hat{\mathbf{z}}$ component of spin, we would find it spin up $(+1/2)$ with probability $|a|^2$ and spin down $(-1/2)$ with probability $|b|^2$. The expected spin is $+(1/2)|a|^2 - (1/2)|b|^2$. Evidently, normalization requires $|a|^2 + |b|^2 = 1$, so the expected spin is $|a|^2 - 1/2$. After the measurement, the spin would be in one of the two S^z basis states, either $\left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$ or $\left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$.

Recall Dirac's formalism, in which a measurement corresponding to an operator is effected by taking the bracket of the complex conjugate transpose of the state vector, the operator, and then the state vector:

$$\langle S^z \rangle = (a^* \quad b^*) S^z \begin{pmatrix} a \\ b \end{pmatrix} \quad . \tag{3}$$

(The asterisk (*) represents complex conjugation, the action of reversing the imaginary part's sign.)

Exercise. comparing (3) with the expected probabilities of a spin being up or down, show that S^z for a single spin has the matrix representation

$$S^z = \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad . \tag{4}$$

(Note that we're ignoring $\hbar$; really S^z is $\hbar$ times what we just said, because electrons have angular momentum $\pm\frac{1}{2}\hbar$.)

Exercise. Convince yourself that an S^z basis state, either up, $\left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$, or down, $\left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$, is an *eigenvector* (proper or "own" state) of the matrix S^z, while general linear combinations of these basis states are not. An eigenvector $\mathbf{v}$ of a matrix M satisfies $M\mathbf{v} = \lambda\mathbf{v}$, where λ is just a number, not a matrix, and is called the *eigenvalue*. The eigenvalues of the matrix S^z are $\pm\frac{1}{2}$, the two possible values for the m quantum number.

Because any measurement leaves a system in an eigenstate of the corresponding operator, these eigenstates take on a central significance in quantum mechanics. We have already seen that the two possible eigenstates of S^z form a basis for the vector space of possible configurations of a single spin $\frac{1}{2}$ (see equation (2)). As another example, the eigenstates E of an energy operator (Hamiltonian) H of a system, such as (1), are the only allowed levels of a system: the Schrödinger equation

$$H\psi = E\psi \tag{5}$$

is an eigenvector equation. Because spin systems have only finitely many possible states, H is easily represented as a matrix. In a "modern physics" course, you may have seen the Schrödinger equation for a free particle. There, $H = \frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}$ plays the role of a matrix of infinite dimension, so what we are doing here really is enormously simpler.

(We review some of the correspondences between continuum wave mechanics and the simpler spin case. For a particle, we call $\psi(x)$ the probability amplitude for finding the particle at position x, because, as Feynman put it, "we don't know what it means." For each x, $\psi(x)$ is a complex number; since there are uncountably infinitely many positions x, ψ corresponds (poetically) to an infinite-dimensional column vector. The probability of finding the particle in some interval (x_1, x_2) is the integral from x_1 to x_2 of the absolute value squared of ψ: $P(x_1, x_2) = \int_a^b \psi^*(x)\psi(x)dx$. Since the probability of finding the particle *some*where is unity, $\int_{-\infty}^{\infty} \psi^*(x)\psi(x)dx = 1$. Roughly speaking, this adds the probability of finding the particle at position 0 plus that at position $0 + \epsilon$ plus that at position $0 + 2\epsilon$ and so forth, including negative positions, too. In the case of spin-½, there are only two possibilities, up and down, instead of infinitely many x's. Therefore, we add just the probabilities that the spin is up and that it is down, arriving at the normalization condition $\psi_\uparrow^*\psi_\uparrow + \psi_\downarrow^*\psi_\downarrow = \psi^\dagger\psi = 1$, where $\psi^\dagger$ means the row-vector version of ψ, complex conjugated.)

While successive consecutive measurements of any one component of spin (*e.g.*, S^z) will always yield the same result, consecutive measurements of perpendicular spin components are uncorrelated: if I measure S^z as spin up, a subsequent measurement of S^x will have a fifty-fifty chance of being "spin right" or being "spin left." This means that if $|z\rangle$ is any eigenvector of S^z,[3] either $e^{i\theta}\binom{1}{0}$ or $e^{i\theta}\binom{0}{1}$ (but not a combination of them), where θ is an arbitrary real number, the expectation value $\langle z|S^x|z\rangle$ is zero. One can show that this implies that the expectation of $S^xS^z + S^zS^x$ is zero for *any* state:

$$\langle S^xS^z + S^zS^x\rangle = (a^* \quad b^*)(S^xS^z + S^zS^x)\begin{pmatrix} a \\ b \end{pmatrix} = 0 \tag{6}$$

for *any* choice of a and b.

Extending this argument[4] lets us write, for spins ½,

$$S^x = \frac{1}{2}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \quad S^y = \frac{i}{2}\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \quad . \tag{7}$$

Exercise. Verify that the choice of spin matrices in equations (7) and (4) satisfies (6). It is likewise true that the expectation of $S^xS^y + S^yS^x$ is zero, but the expectation of the square of any spin component is always $+\frac{1}{4}$.

The following combinations of spin matrices are often very convenient:

$$S^+ = S^x + iS^y; \quad S^- = S^x - iS^y \quad . \tag{8}$$

For spins-½,

$$S^+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \ , \quad S^- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \quad . \tag{9}$$

Exercise. To see why these are called *raising* and *lowering* operators, apply S^+ and S^- to the basis states $\binom{1}{0}$, representing an up spin, and $\binom{0}{1}$, representing a down spin. *By this, I mean to look at the vectors $S^+\psi$ and $S^-\psi$ for each of the two basis states ψ.*

[3] I have used Dirac's "bracket" notation, in which $|\uparrow\rangle = \binom{1}{0} = \psi_\uparrow$ and $|\downarrow\rangle = \binom{0}{1} = \psi_\downarrow$.

[4] A careful development of this subject would include a discussion of spin rotations and the group $SU(2)$. For now, note that the "commutator" $S^zS^x - S^xS^z = iS^y$ is non-zero (and its expectation value is imaginary, not real). If we could measure S^z and S^x simultaneously, the result would correspond to the operator S^zS^x, which would be Hermitian and equal to S^xS^z. It isn't, they don't, and we can't.

Exercise. Use the matrix forms (4) and (7) and the definition (8) to rewrite the Heisenberg energy function (Hamiltonian) (1)

$$H = J(S_1^z S_2^z + \frac{1}{2}(S_1^+ S_2^- + S_1^- S_2^+)) \quad . \tag{10}$$

Let's examine what each term in (10) does to a combination of spin states, $\psi = \psi_1\psi_2 = \begin{pmatrix} a_1 \\ b_1 \end{pmatrix}_1 \begin{pmatrix} a_2 \\ b_2 \end{pmatrix}_2$. Note that the subscript-1 matrices act only on the subscript-1 vector, and likewise with the subscript-2 matrices. Matrices acting on the two spaces are not to be multiplied together. If you prefer, you can think of a *four*-dimensional spin space[5] spanned by basis vectors meaning "up$_1$ up$_2$," "up$_1$ down$_2$," "down$_1$ up$_2$," and "down$_1$ down$_2$." In fact, we'll have to move to this way of thinking soon enough, so you'll be ahead of the game.

The first term in (10) is called the Ising term. It says there's an energy cost $J/4$ whenever the two spins point the same way, either both up or both down. When they point oppositely, the cost is $-J/4$. If we were to stop here, we would say (for positive J) that the ground states $\uparrow_1\downarrow_2$ and $\downarrow_1\uparrow_2$ were degenerate with energy $-J/4$, the excited states $\uparrow_1\uparrow_2$ and $\downarrow_1\downarrow_2$ likewise degenerate with energy $+J/4$. This is essentially classical behavior. Note that S^z basis states (up or down) are eigenstates of the first term: it does not change either spin.

The next two terms allow for the spins to flip. If both spins point the same way, these two terms kill the vector and so fail to contribute (*e.g.*, $S_1^+ S_2^- | \uparrow_1\uparrow_2\rangle = 0$ because $S_1^+ | \uparrow_1\rangle = 0$). If one is up and the other is down, however, the action of these two terms will yield a state in which the one that was up is down and the one that was down is up. The spins have exchanged sign. This spoils the old conservation law of just the Ising piece: it is no longer true that the energy function (Hamiltonian) leaves the S^z components of the two spins invariant. In consequence, eigenstates of S_1^z and S_2^z will no longer be eigenvectors. However, a more general conservation law still applies: the *total* $\hat{\mathbf{z}}$ component of spin, $S^z_{\text{total}} = S_1^z + S_2^z$, is conserved, because H leaves alone the total number of up and down spins.[6]

We've been speaking rather abstractly; it's better to try everything out explicitly. I mentioned above that we could think of the states of the two spins as vectors with four components, the first representing the probability amplitude for both spins being up, the second for $\uparrow_1\downarrow_2$, *etc.* Expressing the four-dimensional basis in terms of the two two-dimensional ones,

[5] This is the tensor product of the two original spin spaces.

[6] There's an additional, more subtle, conservation: the Hamiltonian does not change the total spin represented by the operator $\mathbf{S}\cdot\mathbf{S}$. We'll mention this again but not make use of it in this class.

$$
\begin{aligned}
|1\rangle &= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix}_2 = |\uparrow_1\uparrow_2\rangle \\
|2\rangle &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix}_2 = |\uparrow_1\downarrow_2\rangle \\
|3\rangle &= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix}_2 = |\downarrow_1\uparrow_2\rangle \\
|4\rangle &= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix}_2 = |\downarrow_1\downarrow_2\rangle \quad .
\end{aligned}
\tag{11}
$$

We're going to need four-dimensional matrices, too. By the rules of multiplying a row vector times a matrix times a column vector, the element of the matrix in the i^{th} row and j^{th} column is the result of multiplying the i^{th} basis row vector times the operator times the j^{th} basis column vector:

$$
S_1^z S_2^z = \begin{pmatrix} \langle 1|S_1^z S_2^z|1\rangle & \langle 1|S_1^z S_2^z|2\rangle & \langle 1|S_1^z S_2^z|3\rangle & \langle 1|S_1^z S_2^z|4\rangle \\ \langle 2|S_1^z S_2^z|1\rangle & \langle 2|S_1^z S_2^z|2\rangle & \langle 2|S_1^z S_2^z|3\rangle & \langle 2|S_1^z S_2^z|4\rangle \\ \langle 3|S_1^z S_2^z|1\rangle & \langle 3|S_1^z S_2^z|2\rangle & \langle 3|S_1^z S_2^z|3\rangle & \langle 3|S_1^z S_2^z|4\rangle \\ \langle 4|S_1^z S_2^z|1\rangle & \langle 4|S_1^z S_2^z|2\rangle & \langle 4|S_1^z S_2^z|3\rangle & \langle 4|S_1^z S_2^z|4\rangle \end{pmatrix} = \frac{1}{4}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad . \tag{12}
$$

The off-diagonal elements of $S_1^z S_2^z$ all vanish because the S^z operators flip no spins.[7] The matrix (12) is the *outer product* of the matrices S_1^z and S_2^z given by (4). You may wish to look at some outer-product functions for Mathematica, Maple, or Matlab, available in the directory `/home/5156/assignments/spins`.

The remaining matrices are very easy to calculate. $S_1^+ S_2^-$ gives zero unless acting to the right on $|3\rangle = |\downarrow_1\uparrow_2\rangle$ and to the left on $\langle 2| = \langle\uparrow_1\downarrow_2|$, so

$$
S_1^+ S_2^- = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad ; \tag{13}
$$

$S_1^- S_2^+$ is similarly easily calculated (it's the Hermitian conjugate, $(S_1^+ S_2^-)^\dagger$), and we combine the matrices to get (from (10))

$$
H = \frac{J}{4}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad . \tag{14}
$$

[7] This is because we are representing spins in the S^z basis. If we were representing spins in the basis consisting of "spin right" and "spin left," this operator would flip them.

Problem 1. By hand or using a package such as Maple, find the eigenvectors and eigenvalues of this matrix. Compare with those for the Ising problem. *If you choose to diagonalize the matrix by hand, note that it is block diagonal.*

Notice the symmetry in every solution: under interchange of the spins, the wavefunction is either invariant or else changes sign. We could have seen this from the original Hamiltonian, (10); it is a consequence of Bloch's or Floquet's theorem or more deeply of Nöther's. We call the symmetric solution momentum 0, the antisymmetric momentum π. The Hamiltonian never mixes the two momentum sectors, so we can solve them separately. The momentum-0 sector is spanned by the three vectors

$$\begin{aligned} |s_1\rangle &= |\uparrow_1\uparrow_2\rangle = |1\rangle \\ |s_2\rangle &= \frac{1}{\sqrt{2}}(|\uparrow_1\downarrow_2\rangle + |\downarrow_1\uparrow_2\rangle) = \frac{1}{\sqrt{2}}(|2\rangle + |3\rangle) \\ |s_3\rangle &= |\downarrow_1\downarrow_2\rangle = |4\rangle \quad , \end{aligned} \tag{15}$$

while the antisymmetric space is "spanned" by the single vector

$$|a_1\rangle = \frac{1}{\sqrt{2}}(|\uparrow_1\downarrow_2\rangle - |\downarrow_1\uparrow_2\rangle) = \frac{1}{\sqrt{2}}(|2\rangle - |3\rangle) \quad . \tag{16}$$

The Hamiltonian in the basis of the symmetric space is

$$H_s = \begin{pmatrix} \langle s_1|H|s_1\rangle & \langle s_1|H|s_2\rangle & \langle s_1|H|s_3\rangle \\ \langle s_2|H|s_1\rangle & \langle s_2|H|s_2\rangle & \langle s_2|H|s_3\rangle \\ \langle s_3|H|s_1\rangle & \langle s_3|H|s_2\rangle & \langle s_3|H|s_3\rangle \end{pmatrix} = \frac{J}{4}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad , \tag{17}$$

while in the antisymmetric space

$$H_a = \left(-\frac{3J}{4}\right) \quad . \tag{18}$$

These two matrices are extremely easy to diagonalize: verify that the results agree with what you found in problem 1. Note furthermore the spin values of the three degenerate symmetric eigenstates, which are ground states for ferromagnetic ($J < 0$) interactions. The H-eigenstates $|s_1\rangle$, $|s_2\rangle$, and $|s_3\rangle$ are also eigenstates of S^z_{total} (verify this!) with eigenvalues $+1$, 0, and -1.[8]

The fact that H_s is already diagonal tips us off to another symmetry in the problem: as noted before, H never alters S^z_{total}. Therefore we can split the symmetric space further into $S^z_{\text{total}} = +1$, $S^z_{\text{total}} = 0$, and $S^z_{\text{total}} = -1$ subspaces, each one-dimensional and therefore completely trivial. Note the overall effect: we have reduced the problem from that of diagonalizing one 4×4 matrix to that of four independent 1×1 "matrices." Since matrix diagonalization is in general an $\mathcal{O}(N^3)$ process, the former problem is $4^3/4 = 16$ times harder numerically. As we go to larger chains, the difference becomes enormous: without symmetries we simply could not calculate beyond a few sites.

2. More sites

The degenerate ground state of the two-spin antiferromagnetic ($J > 0$) *Ising* (classical) system included $|2\rangle = |\uparrow_1\downarrow_2\rangle$ and $|3\rangle = |\downarrow_1\uparrow_2\rangle$. These are not eigenstates of the quantum Hamiltonian,

[8] It takes not much more work to confirm that they are also all eigenstates of the operator $S^2_{\text{total}} = (\mathbf{S}_1 + \mathbf{S}_2) \cdot (\mathbf{S}_1 + \mathbf{S}_2)$, all with eigenvalue 2. You may recall that the combination of two spins-½ can have either spin $s = 0$ or $s = 1$, and that the eigenvalue of $S^2_{\text{total}} = s(s+1)$. Our three states, therefore, lie all in the $s_{\text{total}} = 1$ sector and represent the three possible m quantum numbers for a spin 1. We'll not pursue this much further in this course, but it's nice to see it working.

(10), and their energy expectation value $\langle 2|H|2\rangle = \langle 3|H|3\rangle = -J/4$ is larger than the ground-state energy, $-3J/4$. Each state, $|2\rangle$ or $|3\rangle$, appears antiferromagnetic, but the peculiar nature of quantum mechanics permits a lower energy when the two combine antisymmetrically. Interestingly, the symmetric "antiferromagnetic" configuration $(1/\sqrt{2})(|2\rangle + |3\rangle)$ has a high energy.

It comes as no surprise, therefore, that our first naïve guess of the ground state of a large or an infinite chain of antiferromagnetically coupled spins, the Néel state $|\uparrow\downarrow\uparrow\downarrow\ldots\rangle$, is wrong. Of course, we see straight away that the S^+S^- terms in (10) mean the Néel configuration is not an eigenstate.[9]

The surprise is the magnitude of the problem. Consider first a simulation of $N = 36$ classical vector spins.[10] We would need to store two angles, real numbers, for each spin. The interaction energy between neighboring spins i and $i+1$, $\mathbf{S}_i \cdot \mathbf{S}_{i+1}$, is the cosine of their relative angle. In some detached sense, the classical problem is larger than the quantum, for the classical spins can assume any of infinitely many configurations, the quantum only finitely many. Actually solving the classical case (which we can do by hand, but let's imagine it numerically) is, however, a straightforward and fast exercise in minimization. In one approach, we start with a random configuration of the spins, then pick a spin (at random). Its neighbors put it in a potential, which we quickly minimize by changing that spin's two angles. We repeat the exercise until the spins stop changing. If we double the number of sites, there are twice as many angles to twiddle, and the problem takes twice as long.[11]

Contrast this to the spin-½ quantum case, where at first we ignore symmetry. Each spin can point either up or down, two choices. With N sites, we have 2^N possible configurations. To diagonalize the resulting matrix takes of order $(2^N)^3$ operations; for $N = 36$, this is $2^{108} \sim 10^{32}$, which on the world's fastest computer would take longer than the age of the universe.[12]

Yet we *can* learn a great deal about the 36-site system. Our only hope is to reduce the number of configurations. That is where symmetry comes in. Let's reverse the order of symmetries from our two-site example and try $S^z_{\rm total}$ first. Our N-site chain (36 for concreteness) can have a total spin equal to any integer from $-N/2 = -18$ to $+N/2 = +18$, 37 sectors. The ± 18 cases are pretty easy but not of interest. Indeed, for an even number of sites, we need not consider any but the $S^z_{\rm total} = 0$ sector. Just as we saw in the two-site case, a state with some other $S^z_{\rm total}$ quantum number will be a member of a multiplet of states, all with the same $S^2_{\rm total} = s(s+1)$, energy, and other interesting properties but with $S^z_{\rm total}$ ranging from $-s$ to $+s$.[13]

---

[9] Bethe (1931) calculated the true ground state of an infinite spin-½ antiferromagnetic chain: his elegant and very useful technique goes well beyond the level of the present course and is restricted to one-dimensional problems, while the numerical approach we follow here can be extended to two and three dimensions.

[10] These spins have a fixed length but can point anywhere on the surface of a sphere. Do not confuse them with classical Ising spins, which, like quantum spins ½, take only two values.

[11] Of course, we wouldn't really solve this problem this way. However, we might imagine a more complicated classical problem in two or three dimensions where spins have bonds to close neighbors, not just nearest, some of which are ferromagnetic, others antiferromagnetic. Then such an approach becomes plausible. Add temperature and anisotropy, and it becomes necessary.

[12] We've overstated the case by assuming, impossibly, the use of dense matrices. Even sparse-matrix techniques do not suffice. That's just 36 spins. Yet nature manages 10^{23} without difficulty. Thoughts such as these lead naturally to contemplation of a quantum computer, one whose spin elements would be applied to problems equally difficult for classical logic.

[13] In equations (15) and (17), we had a triplet with $s = 1$, $S^2_{\rm total} = s(s+1) = 2$, and energy$= J/4$, while $S^z_{\rm total}$ took values ± 1 and 0. If we need to know the degeneracy of a given state $|x\rangle$ of energy E, we need merely calculate its total spin using $S^2_{\rm total}|x\rangle = s(s+1)|x\rangle$. The degeneracy is the

Let's measure the size of the reduced Hilbert space. Instead of 2^N configurations, we now count only those chains that have total spin projection $S^z_{\rm total} = 0$. We have $S^z_{\rm total} = 0$ if and only if the total number of up spins equals the total number of down spins. Think of the N sites as N boxes. We can fill $N/2$ of them with up spins; the other $N/2$ will (by default) end up with down spins. The order in which we fill half the boxes with up spins doesn't matter, so the appropriate combination is

$$\binom{N}{N/2} = \frac{N!}{(N/2)!^2} \quad ; \tag{19}$$

for $N = 36$, this is about 10^{10}, more than seven times smaller than what we had before but still too big.[14]

Next, we consider translational symmetry. Strictly speaking, a chain of 36 spins doesn't have any, but we're really interested in macroscopic, "infinite" chains. Such a chain differs from the one at hand most notably in lacking endpoints. What happens in a finite chain in the vicinity of the dangling ends does not usually reflect the physics we wish to understand. Therefore, we seek a way to get rid of the endpoints. The most typical technique links them up so that the chain becomes a circle. No point is any closer to the "end" than any other. It isn't a perfect simulation of an infinite chain, but it's often better than what we had.

It also has a translational symmetry we can use to reduce further the size of the Hilbert space. The Hamiltonian

$$\begin{aligned} H &= J \sum_{i=0}^{N-1} (\mathbf{S}_i \cdot \mathbf{S}_{i+1}) \\ &= J \sum_{i=0}^{N-1} (S^z_i S^z_{i+1} + \frac{1}{2}[S^+_i S^-_{i+1} + S^-_i S^+_{i+1}]) \end{aligned} \tag{20}$$

(with the understanding that site N is the same as site 0) is invariant under translation by one unit to the right. This implies a translational symmetry of the eigenvectors, too. Take for example a six-site chain. The state

$$|\uparrow_1\uparrow_2\downarrow_3\uparrow_4\downarrow_5\downarrow_6\rangle \tag{21}$$

should not appear in our basis, since it lacks the full translational symmetry of the Hamiltonian. However, the combination of states

$$\begin{aligned} &\frac{1}{\sqrt{6}}\,(|\uparrow_1\uparrow_2\downarrow_3\uparrow_4\downarrow_5\downarrow_6\rangle + |\downarrow_1\uparrow_2\uparrow_3\downarrow_4\uparrow_5\downarrow_6\rangle + |\downarrow_1\downarrow_2\uparrow_3\uparrow_4\downarrow_5\uparrow_6\rangle \\ &+|\uparrow_1\downarrow_2\downarrow_3\uparrow_4\uparrow_5\downarrow_6\rangle + |\downarrow_1\uparrow_2\downarrow_3\downarrow_4\uparrow_5\uparrow_6\rangle + |\uparrow_1\downarrow_2\uparrow_3\downarrow_4\downarrow_5\uparrow_6\rangle) \quad , \end{aligned} \tag{22}$$

should appear. The first term in (22) is (21). The second term is the result of starting with (21) and translating the whole chain one step to the right; similarly, the third term is the result of letting the translation operator act on the second. Eventually, we come back where we started.

number of states $-s,\ -s+1,\ \ldots,\ s-1,\ s$, or $2s+1$. To get the density of states at energy E, we would add $2s+1$ for each state $|x\rangle,\ |x_2\rangle,\ \ldots$ lying in the $S^z_{\rm total} = 0$ spin sector and having energy E.

[14] We'll mention briefly that we could look at the different $S^2_{\rm total}$ sectors one at a time. The largest of these, $s = 0$, has a Hilbert space (for $N = 36$) another factor of 19 smaller than the $S^z_{\rm total} = 0$ sector. As a practical matter, a single-s subspace is awkward to deal with. The technique outlined by Ramasesha and Soos (reference supplied on request) applies well to one-dimensional chains when all one wants is the ground-state energy but loses its advantages in higher-dimensional problems, problems with long-range interactions, and when one needs eigenvectors, not just energies.

Unlike (21), (22) is invariant under translation. The numerical factor in front is just normalization, so that the dot product of the transposed version of (22) with itself is 1.

Now the symmetry of a wavefunction is a little more complicated than that of a Hamiltonian: because a measurable quantity is given by the bracket

$$\langle v|\text{operator}|v\rangle = (\text{row-vector-}\mathbf{v}^{*\mathrm{T}})(\text{matrix-operator })(\text{column-vector-}\mathbf{v}) \quad , \tag{23}$$

the wavefunction $\mathbf{v}$ is determined only up to a complex factor of unit length, most generally

$$e^{i\theta} \tag{24}$$

for any real number θ. Thus, while (22) is invariant under a translation of one unit to the right, the action of translation by one unit to the right on a general basis vector may multiply the vector by a number of the form (24). With only six sites, θ is restricted to $2\pi n/6$ for integers n. Because it derives from a translational symmetry, θ is called the momentum.[15] From the original state (21), therefore, we construct, in addition to (22) ($n = 0$), basis states for three additional momenta,

$$\begin{aligned} |\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow, n\rangle = \frac{1}{\sqrt{6}}\Big(&|\uparrow_1\uparrow_2\downarrow_3\uparrow_4\downarrow_5\downarrow_6\rangle + e^{n\pi i/3}|\downarrow_1\uparrow_2\uparrow_3\downarrow_4\uparrow_5\downarrow_6\rangle + e^{2n\pi i/3}|\downarrow_1\downarrow_2\uparrow_3\uparrow_4\downarrow_5\uparrow_6\rangle \\ &+e^{3n\pi i/3}|\uparrow_1\downarrow_2\downarrow_3\uparrow_4\uparrow_5\downarrow_6\rangle + e^{4n\pi i/3}|\downarrow_1\uparrow_2\downarrow_3\downarrow_4\uparrow_5\uparrow_6\rangle + e^{5n\pi i/3}|\uparrow_1\downarrow_2\uparrow_3\downarrow_4\downarrow_5\uparrow_6\rangle\Big) \quad , \end{aligned} \tag{25}$$

for $n = 1,\ 2,\ 3$. Momenta $4\pi/3$ and $5\pi/3$ are the complex conjugates of $2\pi/3$ and $\pi/3$. Since the spin chain has a mirror symmetry about the origin, the physics of negative momenta looks just like the physics at positive momenta.[16]

The zero-momentum sector contains the largest number of basis vectors, the others fewer, since not every starting spin configuration, *e.g.*, $|\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\rangle$, admits every momentum. Let's see how far translational symmetry has reduced the size of our Hilbert space. The configuration (21) we used as our example has a period equal to the length of the chain (six): it doesn't coincide with itself until translated by six units. If every configuration had period equal to the chain length, N, the reduction in Hilbert-space size would be a factor of N. Since some configurations have periods less than N, the reduction isn't quite so large. However, it is still very significant. Remember that matrix diagonalization is an order N^3 process, so any way we can manage to reduce the Hilbert space is welcome.

We will use one more symmetry, one that works only because we are restricting ourselves to $S^z_{\text{total}} = 0$. Consider the parity operator, P, which flips all the spins in a configuration. Parity is very much like a translation by half the length of the chain: since $P^2 = 1$, its only eigenvalues are $e^{0\pi i}$ and $e^{1\pi i}$, or ± 1. The ket (25) is not an eigenvector of P. To construct such an eigenvector, we act with P on (25):

$$\begin{aligned} |\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow,\ n,\ p\rangle &= \frac{1}{\sqrt{2}}\left(|\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow,\ n\rangle \pm P|\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow,\ n\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow,\ n\rangle \pm |\downarrow\downarrow\uparrow\downarrow\uparrow\uparrow,\ n\rangle\right) \quad , \end{aligned} \tag{26}$$

[15] Let X be the translation operator. Then if $X|v\rangle = e^{i\theta}|v\rangle$, we say that $|v\rangle$ has momentum θ. The relationship between θ and momentum as ordinarily construed is the subject of solid-state physics.

[16] In a problem with an external magnetic field (which breaks time-reversal symmetry), we have to consider these conjugate momentum sectors, too.

where the eigenvalue p of P is ± 1. Note (from equation (25)) that this expression has twelve terms.

Since the energy is the same for a flipped as for an unflipped configuration, we can find simultaneous eigenvectors of H and of P, and H will not mix sectors of different parity. In fact, we can find simultaneous eigenvectors of translation (momentum), energy, S^z_{total}, and parity.[17] The use of parity symmetry results in a further modest reduction in the size of the Hilbert space.

Now that we have constructed a symmetry-reduced Hilbert space (for $S^z = 0$, a particular momentum, and parity plus or minus one), we need to build a Hamiltonian matrix. Let us label the various kets in the reduced space, of which (26) serves as an example, $|0\rangle'$, $|1\rangle'$, $|2\rangle'$, *etc.*, and the various spin configurations of which they are composed $|0\rangle$, $|1\rangle$, *etc.* There are substantially more unprimed kets than primed. The Hamiltonian in the primed basis has an element in row i, column j given by $H'_{ij} = {}'\langle i|H|j\rangle'$. Most entries will be zero, so it's not efficient to consider every pair of i and j.[18] Rather, we consider each ket j in turn, then act with the Hamiltonian (20) on each of the spin configurations in the state $|j\rangle'$. The $S^z_k S^z_{k+1}$ term in the dot-product expansion (remember that k is a site index) will either add or subtract $J/4$ times a coefficient from the j-j^{th} diagonal term of the matrix H' for each unprimed ket in the expansion of $|j\rangle'$, depending on whether the umprimed ket has spins up-up or down-down, or up-down or down-up, on the sites k and $k+1$. The coefficient of $\pm J/4$ is the absolute value squared of the coefficient with which the unprimed ket appears in the expansion of the primed ket.

Evaluating contributions from the off-diagonal terms $S^+_k S^-_{k+1}$ and $S^-_k S^+_{k+1}$ goes faster with a trick. Because we deal with only one set of symmetry numbers (momentum and parity, while spin-z always is zero) at a time, an unprimed ket that appears in the expansion of one primed ket will never appear in any other (in this momentum-parity sector). In shorthand, let's call the expansion of a primed ket in terms of unprimed kets the *orbit* of the first unprimed ket in the expansion. (It resembles an orbit because under the action of translation and parity it comes back to the same place.) Then *orbits do not intersect.* Now when we consider the action of the off-diagonal elements on each of the unprimed kets in the j^{th} orbit, we check to see if the resulting unprimed ket is the *first* element of some new orbit (which may be the same orbit). If it is, we know by symmetry that the off-diagonal terms acting on our current orbit will generate the *whole* new orbit. If the state is not the first element of some orbit, we ignore it; it's in some orbit certainly, but the off-diagonal operators acting on some other ket in the current orbit will give the first element of that orbit, so we'll get it eventually. Naturally, we need to keep track of coefficients throughout.

3. Determining Degeneracy

To characterize the entire spectrum of energies, we must indicate not only what energies are possible but also with what degeneracy each energy occurs. One way to do this (for n even) is to diagonalize separately the Hamiltonians for $S^z = 0$, $S^z = 1$, $S^z = 2$, ..., $S^z = n/2$; every energy that appears in each of these sectors other than $S^z = 0$ appears a second time in the corresponding negative S^z sector. However, because every multiplet of states includes a representative in $S^z = 0$, it suffices to compute just this sector if we also have a way of determining in *which* multiplet each state lies.

Let E be an energy eigenvalue of the $S^z = 0$ Heisenberg Hamiltonian; assume that it is not degenerate in the $S^z = 0$ sector, that is, that the $S^z = 0$ matrix has no other equal eigenvalue. We'll relax this assumption momentarily. The corresponding eigenvector, $|\psi\rangle$, must simultaneously be an eigenvector of H, S^z, and S^2:

[17] Such operators are said to commute: expressed as matrices, X representing translation, $HPXS^z_{\text{total}} = PS^z_{\text{total}}HX$ *etc.* This is *not* true of matrices or quantum operators in general, as we have already seen with spin components S^x and S^y.

[18] In a more sophisticated treatment, we would implement H' as a sparse matrix, storing only the non-zero values and diagonalizing with the Lanczos technique.

$$\begin{aligned} H|\psi\rangle &= E|\psi\rangle \\ S^z|\psi\rangle &= 0 \\ S^2|\psi\rangle &= s(s+1)|\psi\rangle \quad . \end{aligned} \tag{27}$$

Recall that S^z and S^2 are defined as sums over all sites:

$$\begin{aligned} S^z &= \sum_{\text{sites } i} S_i^z \\ S^\pm &= \sum_{\text{sites } i} S_i^\pm \\ S^2 &= S^{x2} + S^{y2} + S^{z2} \\ &= S^{z2} + \tfrac{1}{2}(S^+S^- + S^-S^+) \\ &= S^{z2} + S^-S^+ \quad , \end{aligned} \tag{28}$$

where the last step uses the fact that $|\psi\rangle$ lies in the $S^z = 0$ sector.

As previously noted, the state $|\psi\rangle$ is part of a $(2s+1)$-fold-degenerate multiplet of states in S^z sectors from $-s$ to $+s$, where s is the S^2 quantum number in the last line of (27). If we have also restricted the solution to a particular momentum (section 2) other than 0 or π, there will also be a factor of two degeneracy for momentum and a similar factor of two for parity, if we've implemented it.

Because the S^2 operator contains terms like $S_i^+S_j^-$ for every pair of sites i and j, not just nearest neighbors, we would rather avoid having to apply it to $|\psi\rangle$. Instead, note that

$$\langle\psi|S^2|\psi\rangle \;=\; s(s+1) \;=\; \langle\psi|S^{z2}|\psi\rangle + \langle\psi|S^-S^+|\psi\rangle \;=\; 0 + \langle S^+\psi|S^+\psi\rangle \quad , \tag{29}$$

where, since $S^z|\psi\rangle = 0$, the new ket $|S^+\psi\rangle$ must lie in the $S^z = 1$ sector but is not normalized. Therefore, to find s, we let S^+ act on ψ, take the norm of the resulting ket, and solve a quadratic equation for s.

We have assumed that $|\psi\rangle$ is non-degenerate in the given S^z (and possibly momentum) sector, but we shall not always be so fortunate. If we find eigenvectors $|\psi_0\rangle$, $|\psi_1\rangle$, …, $|\psi_{m-1}\rangle$, all with energy E, it is *possible* but highly unlikely that computer will have picked the m vectors to be eigenvectors of S^2. Rather, there exist m orthogonal linear combinations of the kets $|\psi_k\rangle$ that *are* eigenvectors of S^2 with various eigenvalues. Since all the $|\psi_k\rangle$ share energy eigenvalue E, these m linear combinations will, too. We need to *rediagonalize* the m vectors with respect to the operator S^2 to find these linear combinations. In the actual calculation, we shall use the simple extension of (29), $\langle\phi|S^2|\psi\rangle = \langle S^+\phi|S^+\psi\rangle$ for $|\phi\rangle$ and $|\psi\rangle$ in the $S^z = 0$ sector.

We didn't have to rediagonalize in the special case, $m = 1$, of a non-degenerate eigenvector $|\psi\rangle$. In this case as well as in the general case, there are guaranteed to exist simultaneous eigenvectors of H, S^z, and S^2. In this case as well as in the general case, there exist m independent linear combinations of m vectors that are such simultaneous eigenvectors. However, when $m = 1$, there's only one "linear combination," and it's just $|\psi\rangle$.

4. Correlation functions

We have already seen that the ferromagnetic problem ($J < 0$) has the classically expected ground state, $|\uparrow\uparrow\ldots\uparrow\rangle$. How are we going to find this in the $S^z = 0$ sector, however? *No* state in this sector, after all, has any net magnetization: $\langle\text{state}|S^z_{\text{total}}|\text{state}\rangle = 0$. In the two-site case, there were three ferromagnetic ground states, (15), of which $|s_2\rangle$ fell in the $S^z = 0$ sector. While

it had no net magnetization in the $\hat{\mathbf{z}}$ direction, it did have magnetization in the $\hat{\mathbf{x}}$ direction. For a chain with many sites, the ferromagnetic state would seem rather more difficult to identify. The problem, however, is easily overcome. Instead of looking for net magnetization in any direction, we examine the correlation between two distant spins at sites i and j. Since the chain is translationally invariant, this must be a function only of the difference between the two sites:

$$C(i,j) = C(|i-j|) = \langle \text{ground state} | \mathbf{S}_i \cdot \mathbf{S}_j | \text{ground state} \rangle \quad . \tag{30}$$

Such a function is called a site-site (or spin-spin) correlation. If C is zero, it means the two sites are completely independent. So long as the sites i and j are distinct, the correlation (30) will be some number between $-3/4$ and $+1/4$.

As an example, consider the two-site ferromagnetic ground state $|s_2\rangle$ from equation (15). We calculate

$$\begin{aligned} C(1,2) &= \langle s_2 | S_1^z S_2^z + \tfrac{1}{2}[S_1^+ S_2^- + S_1^- S_2^+] | s_2 \rangle \\ &= \tfrac{1}{2}\left(\langle\uparrow\downarrow| + \langle\downarrow\uparrow|\right)\left[S_1^z S_2^z\left(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle\right) + \tfrac{1}{2}S_1^+ S_2^-\left(0 + |\downarrow\uparrow\rangle\right) + \tfrac{1}{2}S_1^- S_2^+\left(|\uparrow\downarrow\rangle + 0\right)\right] \\ &= \tfrac{1}{2}\left(\langle\uparrow\downarrow| + \langle\downarrow\uparrow|\right)\left[\left(-\tfrac{1}{4}|\uparrow\downarrow\rangle - \tfrac{1}{4}|\downarrow\uparrow\rangle\right) + \tfrac{1}{2}|\uparrow\downarrow\rangle + \tfrac{1}{2}|\downarrow\uparrow\rangle\right] \\ &= \tfrac{1}{2}\left(\tfrac{1}{4}\langle\uparrow\downarrow|\uparrow\downarrow\rangle + \tfrac{1}{4}\langle\downarrow\uparrow|\downarrow\uparrow\rangle\right) \\ &= 1/4 \quad . \end{aligned} \tag{31}$$

The positive sign tells us, as we expected, that the two spins point in the same direction; that the magnitude is as large as possible tells us that they *always* point in the same direction in the ground state.

Exercise. Following the calculation (31), show that the spin-spin correlation for the state $|a_1\rangle$ in equation (16) is $-3/4$, indicating a perfect antiferromagnetic correlation.

The correlation is a function of distance. It seems reasonable to say that if $\lim_{x\to\infty} C(x) \neq 0$, the chain has *long-range order*, since no matter how far away the two spins, what happens on one site tells us what's happening on the other site. If, on the other hand, $C(x)$ is non-zero only for small x but goes to zero in the limit of a large distance, we would say that the chain has only *short-range order*.[19] When looking for antiferromagnetic correlations, we need to account for the fact that the sign of C will change depending on whether the distance between two sites is odd or even; the total number of sites in the chain should additionally be even for an antiferromagnetic chain unless we're interested in studying effects of frustration.[20]

We can never actually take the limit $x \to \infty$, but for any finite chain of N sites, we can graph C for $x = 1,\ 2, \ldots,\ N/2$. (Why can we go only so far as $N/2$?) We can then fit the resulting curve to a form

$$C_N(x) = C_N(\infty) + a_{_N} e^{-x/r_0} \quad , \tag{32}$$

where $C_N(\infty)$ is the long-range correlation and r_0 the characteristic decay length. We graph our best fits for $C_N(\infty)$ against N and fit *that* graph to a similar form to estimate $C_\infty(\infty)$. With only a few points, we shall need not only the best estimate of $C_\infty(\infty)$ but also a statistical estimate of its range. If $C_\infty(\infty)$ is further away from zero than our uncertainty in its measurement, the chain would appear to have long-range order.

5. Conclusions

Once we have diagonalized a spin chain, we have an enormous number of things to calculate, beginning with the energy density of states (a smoothed version of a scatter plot with one data

[19] Sometimes one makes a finer distinction based on the speed with which the correlation decays: $\lim_{x\to\infty} C(x)/e^{-x/r_0} = \text{constant}$ is short-range ordered (exponentially decaying), but $\lim_{x\to\infty} C(x)/x^{-\alpha} = \text{constant}$ is *critical* (algebraically decaying).

[20] A technical word, not a description of the student's efforts.

point at the energy of each eigenvalue) and the dispersion relation giving energy as a function of momentum. With enough spins, we must abandon dense matrices and go to Lanczos techniques. We could also go to a higher dimension than one and consider further-than-nearest-neighbor bonds or random bonds. These possibilities all bring up the issue of frustration, in which the classical problem's bonds cannot all be satisfied simultaneously. Alternatively, we could consider spins greater than one-half and Haldane's results on spin order in such chains.[21]

We should not leave you with the idea of exact diagonalization as the only approach to collections of spins. We have already mentioned the Bethe-Ansatz ground state as an exact solution. Sometimes one can construct a Hamiltonian for which an exact solution is known, then expand in some parameter to learn about a wider class of problems. Spin-wave theory, mean-field theory, the density-matrix renormalization group, Monte Carlo, and variational approaches, among others, have all been applied to spin chains.

[21] For numerical evidence supporting Haldane's conjecture, read Golinelli, Jolicoeur, Lacaze, "Finite-lattice extrapolations for a Haldane-gap antiferromagnet," *Phys. Rev. B***50** (1994) 3037-3044. For a theoretical discussion, see Auerbach, *Interacting Electrons and Quantum Magnetism*, Springer 1994.

Physics Z-5156, Fall 2006

Supplementary notes on changing bases

or why is it called "diagonalization"?

To answer a student's request for more details on how to change from one basis to another and to make more explicit some of our assumptions, here are some notes on quantum mechanics as linear algebra. For a thorough explanation of the different spin bases and how they are related by rotations in real space, see volume III from Feynman's *Lectures in Physics*.

If a system is in the i^{th} eigenstate $|x^{(i)}\rangle$ of an operator X, it has no overlap with any other eigenstate $|x^{(j)}\rangle$ ($j \neq i$), so we write $\langle x^{(j)}|x^{(i)}\rangle = 0$. Conversely, it overlaps with 100% probability with itself, $\langle x^{(i)}|x^{(i)}\rangle = 1$. We abbreviate this $\langle x^{(j)}|x^{(i)}\rangle = \delta_{ij}$, where δ is the Kronecker delta, 1 if its subscripts are equal, 0 otherwise. These rules, the completeness of eigenstates, and the superposition principle let us represent quantum mechanics in the language of linear algebra, with $|x^{(i)}\rangle$ being a column vector with (in this basis) all zeroes except for a 1 in the i^{th} position. The quantum-mechanical rules of time evolution (which we have not discussed) and conservation of probability require the elements of a state vector to take on complex values. Since the probability of a state $\boldsymbol{\psi}$ must be real, we define the inner product so that the row vector is complex conjugated. Thus $\langle\psi|\phi\rangle = \boldsymbol{\psi}^\dagger\boldsymbol{\phi} = \boldsymbol{\psi}^{*\mathrm{T}}\boldsymbol{\phi}$, where * represents complex conjugation, $^{\mathrm{T}}$ transposition (column vector $\leftrightarrow$ row vector), and $^\dagger = {}^{*\mathrm{T}}$ is called Hermitian conjugation.

Consider an operator X with eigenvectors $|x^{(i)}\rangle$ and associated eigenvalues $x^{(i)}$. The expectation value of X in an eigenstate $|x^{(i)}\rangle$ is by definition just the eigenvalue, $x^{(i)}$. This suggests writing for the expectation value

$$\langle x^{(i)}|X|x^{(i)}\rangle = x^{(i)}\langle x^{(i)}|x^{(i)}\rangle = x^{(i)} \quad ,$$

or in vector language,

$$\mathbf{x}^{(i)\dagger}X\mathbf{x}^{(i)} = \mathbf{x}^{(i)\dagger}(x^{(i)}\mathbf{x}^{(i)}) = x^{(i)} \quad .$$

(I put vectors in **boldface**, using subscripts for Cartesian indices and superscripts in parentheses to identify basis elements, *e.g.*, $^{(i)}$, or the basis in general, *e.g.*, $^{(X)}$ for the X-eigenvector basis. I leave off the basis superscript where the statement is independent of basis or where the basis is specified in the text.) Quite independently of basis, the matrix X must be Hermitian, meaning $X = X^\dagger$, or its expectation value will not always be real. Since in the eigenstate basis the row vector $\mathbf{x}^{(i)\dagger}$ and the column vector $\mathbf{x}^{(i)}$ each contains only one non-zero element in the i^{th} position, the X-basis representation of X must be diagonal and real:

$$X_{ij}^{(X)} = \delta_{ij}x^{(i)} \quad .$$

What if we represent X instead in the basis of the eigenstates of Y, in which $\mathbf{y}^{(i)}$ has only one non-zero element but now $\mathbf{x}^{(i)}$ may have many? In the current problem, think of X as the Hamiltonian (energy) operator, which in the energy basis is diagonal, but we know how to calculate it only in the basis of eigenstates of $Y = S^z$. We would like the eigenvalues $x^{(i)}$ and the Y-basis eigenstates of X, $\mathbf{x}^{(i)(Y)}$.

First expand the j^{th} X-basis state in terms of the Y basis states:

$$\mathbf{x}^{(j)} = \sum_i \mathbf{y}^{(i)}C_{ij} \quad .$$

A general state vector $\boldsymbol{\phi}$ which in the X basis had elements $\phi_i^{(X)}$ (*i.e.*, $\boldsymbol{\phi} = \sum_i \phi_i^{(X)}\mathbf{x}^{(i)}$, or $|\phi\rangle = \sum_i \phi_i^{(X)}|x^{(i)}\rangle$) will in the Y basis look like

$$\boldsymbol{\phi} = \sum_j \phi_j^{(X)}\mathbf{x}^{(j)} = \sum_{ij} \phi_j^{(X)}C_{ij}\mathbf{y}^{(i)} = \sum_i \phi_i^{(Y)}\mathbf{y}^{(i)}$$

where

$$\phi_i^{(Y)} = \sum_j C_{ij}\phi_j^{(X)} \quad ,$$

or

$$\boldsymbol{\phi}^{(Y)} = \mathrm{C}\boldsymbol{\phi}^{(X)} \quad .$$

The matrix C, serving as a Rosetta Stone for converting from the X to the Y basis, must be unitary: $\mathrm{C}^\dagger\mathrm{C} = 1$. Otherwise, we'd get a different answer for $\langle\psi|\phi\rangle$ depending on whether we calculated it as $\boldsymbol{\psi}^{(X)\dagger}\boldsymbol{\phi}^{(X)}$ or as $\boldsymbol{\psi}^{(Y)\dagger}\boldsymbol{\phi}^{(Y)} = (\mathrm{C}\boldsymbol{\psi}^{(X)})^\dagger(\mathrm{C}\boldsymbol{\phi}^{(X)}) = \boldsymbol{\psi}^{(X)\dagger}\mathrm{C}^\dagger\mathrm{C}\boldsymbol{\phi}^{(X)}$.

We can convert X^X in the X basis to X^Y in the Y basis by surrounding it on either side with matrices that undo the conversion we just effected in the bra (row vector) and ket (column vector):

$$X^Y = \mathrm{C}X^X\mathrm{C}^\dagger$$

so that

$$\boldsymbol{\psi}^{(Y)\dagger}X^Y\boldsymbol{\phi}^{(Y)} = \boldsymbol{\psi}^{(X)\dagger}X^X\boldsymbol{\phi}^{(X)}$$

for any vectors ψ and ϕ.

As suggested before, we are interested this week in letting X be the Hamiltonian H and Y the z projection of spin, S^z. We know H in the S^z basis, and we wish to find its diagonal representation in its own basis. In effect, we need to find the similarity matrix C. From the definition of C above, we see its ij^{th} element as nothing more than the amount of the i^{th} vector in the new basis present in the j^{th} vector in the old. Equivalently, it is the overlap between these two vectors. Equivalently again, it is the projection of the old basis vector onto the new. *In other words, the columns of* C *are the eigenvectors of* X (i.e., H) *in the* Y (S^z) *basis.*

We can go through exactly the same derivation in bra-ket notation. Here, the amount C_{ij} of Y-basis vector $y^{(j)}$ in the X-basis vector $x^{(i)}$ is written

$$C_{ij} = \langle\boldsymbol{y}^{(i)X\dagger}|\boldsymbol{x}^{(j)X}\rangle = \langle\boldsymbol{y}^{(i)Y\dagger}|\boldsymbol{x}^{(j)Y}\rangle \quad , etc.$$

Now the ij^{th} element of X in the Y basis is

$$X_{ij}^{(Y)} = \langle y^{(i)}|X|y^{(j)}\rangle \quad ,$$

since $\langle y^{(i)}|$ in the Y basis is a row vector with 1 in the i^{th} place and all else 0, while $|y^{(j)}\rangle$ in the Y basis is a column vector with 1 in the j^{th} place. We can write

$$\begin{aligned} |y^{(j)}\rangle &= \sum_{j'} |x^{(j')}\rangle C^*_{j'j} = \sum_{j'} |x^{(j')}\rangle(\langle x^{(j')}|y^{(j)}\rangle) \\ &= \left(\sum_{j'} |x^{(j')}\rangle\langle x^{(j')}|\right)|y^{(j)}\rangle \quad . \end{aligned}$$

Now you know what people mean when they say "$(\sum_{j'} |x^{(j')}\rangle\langle x^{(j')}|) = 1$." It works only if the sum is over a complete set of states (not just some of them) and if there are no other factors that depend on the summation index (j').

We do the same thing to the bra to find

$$\begin{aligned} X^{(Y)}_{ij} &= \langle y^{(i)}|X|y^{(j)}\rangle = \sum_{i'j'} \langle y^{(i)}|x^{(i')}\rangle\langle x^{(i')}|X|x^{(j')}\rangle\langle x^{(j')}|y^{(j)}\rangle \\ &= \sum_{i'j'} \langle y^{(i)}|x^{(i')}\rangle\langle x^{(j')}|y^{(j)}\rangle x_{i'}\delta_{i'j'} \\ &= \sum_{i'} \langle y^{(i)}|x^{(i')}\rangle\langle x^{(i')}|y^{(j)}\rangle x_{i'} \\ &= \left[\mathrm{C}X^{(X)}\mathrm{C}^\dagger\right]_{ij} \quad , \end{aligned}$$

or

$$\mathrm{C}^\dagger X^{(Y)}\mathrm{C} = X^{(X)} \quad \text{(diagonal)}.$$

We give LAPACK the Hermitian matrix $X^{(Y)}$ (the Hamiltonian in the S^z basis), and it uses something clever, like LU factorization, to return the matrices C and $X^{(X)} = \mathrm{diag}(x_0, x_1, \ldots, x_{n-1})$.

Physics Z-5156, Fall 2006

Chapter 9. Molecular Dynamics on a Parallel Computer

A good starting reference is `comp-gas/9303002` (on `arxiv.org`): D.M. Beazley and P.S. Lomdahl, "Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5," *Parallel Computing* **20**, 173–195 (1994). This and a couple of other papers by the same authors are also available at `citeseer.ist.psu.edu`: S.J. Zhou, D.M. Beazley, P.S. Lomdahl, B.L. Holian, "Large-Scale Molecular-Dynamics Simulations of Three-Dimensional Ductile Failure," *Phys. Rev. Lett.* **78**, 479–482 (1997); and B.L. Holian, P.S. Lomdahl, S.J. Zhou, "Fracture Simulations via Large-Scale Nonequilibrium Molecular Dynamics," *Proc. of CECAM Workshop, Lyon, July 15–19, 1997*. For a general overview of realistic simulation of fracture, read F.F. Abraham, J.Q. Broughton, N. Bernstein, and K. Kaxiras, *Computers in Physics* **12**, 539 (1998), publicly available at `http://www.aip.org/cip/pdf/featnd98.pdf`. Chapters on classical molecular dynamics and parallel computing in Thijssen, *Computational Physics*, should also be helpful, particularly on the Verlet integration method.

We shall integrate Newton's equations for a modified Lennard-Jones pair potential between particles to see how a crack in a two-dimensional solid propagates with time. Our simulation will be less realistic than that described in the references, but the principles are the same.[1] To make efficient use of available computer resources, we shall use the Message-Passing Interface, MPI, on IRCE, a 50-node Beowulf cluster managed by the Research-Oriented-Computing group within USF's office of Academic Computing. The programming environment for the cluster is essentially the same as that used on current-model supercomputers such as the Cray XT3 and the IBM Blue Gene (see documentation in Packet 3).

This is a group project. After our first class discussion, students will meet to determine overall strategy for the project, to subdivide the project into manageable pieces, and to delegate responsibilities. They should meet several more times over the three weeks to coordinate efforts and to ensure the project is on track. In the last week, the group will submit a report on methods and results.

[1] In particular, the Lennard-Jones potential isn't really justified except perhaps in noble-gas solids. The dependences of bonds on direction and local electronic density require three-body and higher-order terms. For insulators, see, for example, F.H. Stillinger and T. Weber, *Phys. Rev. B***31**, 5262 (1985) and J. Justo, M.Z. Bazant, E. Kaxiras, V.V. Bulatov, and S. Yip, *Phys. Rev. B***58**, 2539 (1998).