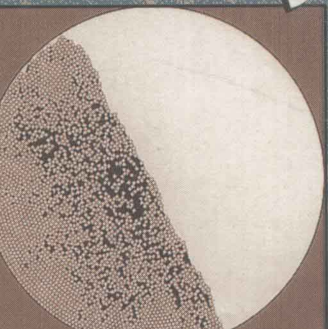
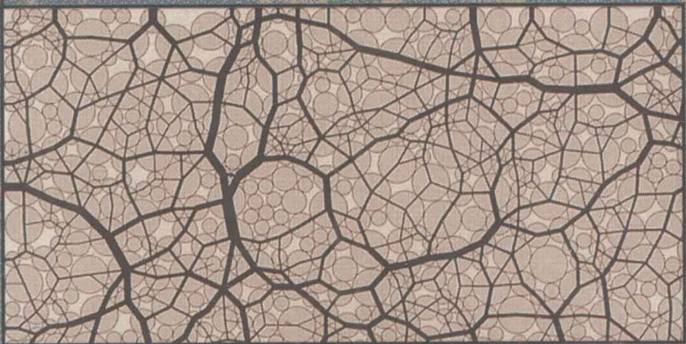
K. H. Hoffmann · M. Schreiber
Editors

Computational Physics

Selected Methods
Simple Exercises
Serious Applications



Including MS-DOS Diskette



Springer

Karl Heinz Hoffmann
Michael Schreiber (Eds.)

Computational Physics

Selected Methods
Simple Exercises
Serious Applications

With 145 Figures, 15 Tables
and a 3.5″ MS-DOS Diskette

Springer

Professor Dr. Karl Heinz Hoffmann
Professor Dr. Michael Schreiber

Institut für Physik
Technische Universität Chemnitz-Zwickau
D-09107 Chemnitz
Germany

Library of Congress Cataloging-in-Publication Data.

Hoffmann, K. H. (Karl Heinz), 1953– . Computational physics : selected methods, simple exercises, serious applications / K. H. Hoffmann, M. Schreiber. p. cm. Includes bibliographical references and index.
ISBN 3-540-60689-0 (Berlin : alk. paper)
1. Physics–Problems, exercises, etc. – Methodology. 2. Numerical calculations. 3. Mathematical Physics. I. Schreiber, Michael, 1954– . II. Title.
QC32.H67 1996 530'.01'13–dc20 96-2830

ISBN 3-540-60689-0 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready copy from the editors using a Springer TEX macro package
Cover design: Erich Kirchner, Heidelberg

SPIN 10525808 56/3144 – 5 4 3 2 1 0 – Printed on acid-free paper

Preface

Computational physics is the field in physics that has experienced probably the most rapid growth in the last decade. With the advent of computers, a new way of studying the properties of physical models became available. One no longer has to make approximations in the analytical solutions of models to obtain closed forms, and interesting but intractable terms no longer have to be omitted from models right from the beginning of the modeling phase. Now, by employing methods of computational physics, complicated equations can be solved numerically, simulations allow the solution of hitherto untractable problems, and visualization techniques reveal the beauty of complex as well as simple models. Many new and exciting results have been obtained by numerical calculations and simulations of old and new models.

This book presents samples of many of the facets that constitute computational physics. Our aim is to cover a broad spectrum of topics, and we want to present a mixture ranging from simple introductory material including simple exercises to reports of serious applications. This is not meant to be an introductory textbook on computational physics, nor is it a proceedings volume of a research conference. This book instead provides the reader with an overview of computational physics, its basic methods, and its many areas of application. Our coauthors lead the reader into new and "hot" topics of research, but the presentation does not require any specific knowledge of the topics and methods. We hope that a reader who has gone through the book can appreciate the wealth of computational physics and is motivated to proceed with further reading.

The topics covered in this book cover a wide spectrum, with a coarse division into "Monte Carlo" type and "molecular dynamics" type chapters. We start with discussing random numbers and their generation on computers. Then these random numbers are used in a variety of applications, which center around "Monte Carlo methods". In these applications the focus is first on classical systems in physics, chemistry, biology, material science, and optimization. Then quantum-mechanical problems are investigated by Monte Carlo procedures. On our way we also encounter quantum chaos and fractal concepts, which are of increasing importance nowadays. The transition from "Monte Carlo" to "molecular dynamics" occurs in the chapter on hybrid methods, which combine elements of both. Then "molecular dynamics" methods are presented, with fluids and solids covered. A chapter on finite-element methods follows, and the two final chapters present principles of parallel computers and associated programming models.

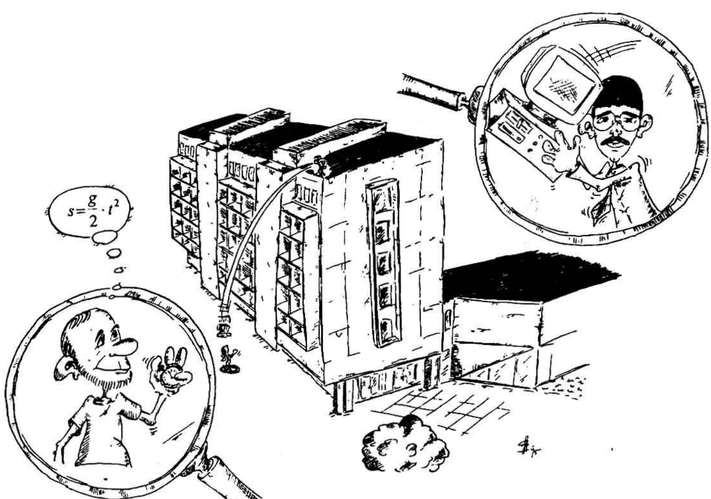
As usual in physics, only active interaction with the matter at hand provides deep insight, and thus we include a diskette that contains sample programs and demonstrations to support the interaction of the reader with the text. The sample programs and demonstrations are selected to provide a glimpse of current research activities, even though the limitations of the available hardware and/or the limited patience of some readers might require a reduction in the dimensionality or size of the application. Also some exercises are included to further foster an active use of this book.

The material in this book is born out of lectures the authors gave at a Heraeus Summer School on computational physics at the Technical University in Chemnitz. The aim of the summer school was the same as the aim of this book: to give a sampler of the field. Due to the gracious funding by the Dr. Wilhelm Heinrich Heraeus and Else Heraeus Foundation the editors (see figure) were able to present two weeks of intense lecturing and "learning by doing" to more than 80 students. We would like to use this opportunity to thank the Heraeus Foundation for making the summer school and this book possible.

But most important we like to thank our coauthors for their contributions to this volume (as well as for their lectures at the summer school). We very much appreciate their willingness to contribute even under the severe limitations that their everyday teaching and research activities (and administrative duties) put on their time. And finally we thank Jörg Arndt, Peter Blaudeck, Andre Fachat, Göran Hanke, Karin Kumm, Sven Schubert, and Peter Späht for their technical help and Springer-Verlag for making this volume a reality.

Chemnitz, December 1995

Karl Heinz Hoffmann and *Michael Schreiber*



With this original answer to the question "How to measure the height of the building of the Institut für Physik in Chemnitz with a computer and a stop watch only?" the editors give a peculiar interpretation of the topic "Physics with a computer".

Contents

* Software included on the accompanying diskette.

Random Number Generation*

Dietrich Stauffer

Institut für Theoretische Physik, Universität zu Köln, D-50923 Köln, Germany
e-mail: `stauffer@thp.uni-koeln.de`

Abstract. The sad situation of random number generation is reviewed: there are no good random numbers. But life has to go on anyhow, and thus we explain how to produce reasonable random numbers efficiently, emphasizing multiplication with 16807 and the Kirkpatrick–Stoll R250 generator.

1 Introduction

Molecular Dynamics and Monte Carlo are the two standard simulation methods of the last decades. Monte Carlo simulations use random numbers to produce random fluctuations. Today, they are no longer made at the roulette tables in Monaco, but on computers. In the good old days, people printed tables of random numbers from which the user could read them off. This, of course, is somewhat tedious when simulating a square lattice of size one million times one million, today's world record [1]. About a decade ago, computer chips became available which produced random numbers through the thermal noise of the electrons, about one number per microsecond. This is not fast enough for many quality applications. Besides, for testing purposes we would like to have *reproducible* random numbers: when we have made a program more efficient without changing the results, we want to run it again and indeed get exactly the same results, and not just roughly the same, within the statistical errors of the Monte Carlo simulations. Moreover, when we switch from one computer to another, we would like again to get the same results: portability is important. Thus special chips using thermal noise are not suitable for this purpose.

Also, the random numbers should be produced quickly since Monte Carlo simulations consume lots of time and we never have enough of it. Thus we need efficient methods, and on many computers it is very slow to call a function or subroutine to produce one random number. Thus a good random number generator should be:

(1) random
(2) reproducible
(3) portable
(4) efficient

Using the built-in random number generator of your computer can make your program inefficient and nonportable. (Seymour Cray knew what he was doing:

* Software included on the accompanying diskette.

his random number generators for the good old CDC series or modern Crays were efficient.) Besides, the user then does not understand what is going on.

Thus we now review why the above criteria are difficult to fulfill and what to do about it, by programming your own random numbers.

2 The Miracle Number 16807

Linear congruential random number generators multiply the last random integer by some big factor, add another integer to it, treat the sum modulo some power of two, and normalize this integer to the interval between zero and unity. This all sounds very complicated, sometimes is presented in this complicated fashion in the literature, and may cause you to give up programming your own random numbers. Thus simply forget these complications and look at the following Fortran or Basic statement, which works for most 32-bit machines:

```
IBM = IBM*16807
```

(fans of Pascal and C should end this line with a semicolon; and enemies of International Bussiness Machines may use a different variable name). If you start with an odd integer for `IBM`, e.g., through `IBM = 2*ISEED-1`, then this single program line should give you, again and again, integers `IBM` distributed randomly between -2^{31} and 2^{31}. Just try it out. Why does it work ?

If you multiply two ten-digit integers, the result will be an integer with about twenty digits and is difficult to obtain by paper and pencil. You may estimate, however, the leading digit correctly without too much effort. On a computer, you may not be able to store more then ten digits for each integer. Then most computers simply throw away, without any error message, the somewhat predictable leading digits and keep only the ten least significant digits. Of course, computers work with binary digits (bits) and not with decimal ones, and with four-byte integers (32 bits) all leading bits beyond the least significant 32 bits are thrown away if the product of two integers has more than 32 bits. In terms of decimal numbers restricted to be at most 999, this would mean that the product of 123 and 899 is not 110577 but merely 577. It is clear that these least significant digits are difficult to predict, that means for a user they look pretty random.

In your youth you have learned that a*b equals b*a, and that the product of two positive numbers is again positive. In linear algebra or quantum mechanics you found out that the first statement was a lie, and now you realize the same for the second statement: `IBM*16807` may be negative even when `IBM` was positive. The reason is that the first (most significant) bit of an integer indicates the sign. Thus before the leading bits of the product were thrown away, the product was positive; but then only the last 32 bits were kept, and the leftmost (most significant) bit may be zero (positive 32-bit number) or one (negative 32-bit number). So, plus times plus is minus, in about half the cases.

Some ancient DEC computers may not have liked this overflow above the 32-bit limit, but otherwise I am not aware of computers where the above Fortran statement causes trouble. Thus we have not only an efficient one-line random number generator, but also a portable one.

If for some reason you want only positive random numbers `IBM`, then you have to add 2^{31} to them if they are negative. This number 2^{31} is too large to handle for the 32-bit computer, but $2^{31} - 1 = 2147483647$ is fine. Thus try

```
IF(IBM.LT.0) IBM = IBM + 2147483647 + 1
```

and it works if the computer is too stupid to find out that you really want to add 2^{31}.

If you want to normalize this number to the interval between 0 and 1, you multiply a positive random integer by $2^{-31} = 4.656612 \times 10^{-10}$. If they are both positive and negative, use

```
Z = 0.5+2.328306 E-10 * IBM
```

to get a random number z between 0 and 1. Of course, this normalization from an integer to a real number costs a lot of computer time; you could do it faster by learning how a floating point number is stored in your computer and then constructing one via bit operations treating the random integer as a bit string for the mantissa.

However, in most cases this normalization is not needed, and you may stay within integer arithmetic. For example, some command `GOTO 1` should be executed with probability p. Normally this is done with

```
IF(Z.LE.P) GOTO 1
```

requiring a random number between 0 and 1. This normalization is avoided by

```
IF(IBM.LE.IP) GOTO 1
```

provided you have defined once (and not millions of times, i.e., for each random number) the variable `IP = 2147483648.0*(2.0*P-1.0)`

```
IF(P.GT.0.999) IP=2147483647
IF(P.LT.0.001) IP=-2147483648
```

which varies between -2^{31} and 2^{31}. Now the computer runs faster. (The last two "if" statements are precautions, seldomly needed, in case rounding errors cause trouble in the conversion to integers if $p = 0$ or $= 1$.)

The number $16807 = 7^5$ is not entirely arbitrary; historically earlier was 65539, and 65549 has also been used. So you may mix them, using in most of your program lines multiplication with 16807, but sometimes also 65539. Do not try to produce different samples just by changing the multiplicator from 16807 to 16809, then 16811, and so on. Also, your `IBM` numbers must always be odd integers; to be safe I start with an integer `ISEED` and then state once `IBM = 2*ISEED-1`, as mentioned already above.

If you simulate at zero temperature (see Sect. 4), then the probabilities are 0, 1/2, and 1 only. With integer random numbers `IBM` varying between -2^{31} and $+2^{31}$ the conditions and Boltzmann integers then have to be formulated exactly as stated above (not `IBM .LT. IP` for example), to avoid a spin flipping when it should not flip. With floating point random numbers you need double precision real*8. This detail may be important for the fraction of frozen spins in spin glasses [7].

3 Bit Strings of Kirkpatrick–Stoll

The principle of Tausworth shift generators has been around for a long time but physicists started to use it mainly after Kirkpatrick and Stoll made it popular in a physics journal [2]. For many years it was regarded as superior to multiplication with 16807; this is no longer true but at least it offers a completely different alternative. It requires bit-manipulation functions which had not yet been standardized before Fortran 90 (they are part of the C language standard) and which were initially demanded by the Pentagon for signal analysis.

Imagine you have two 32-bit integers M and N. Then the exclusive-or operation puts a bit equal to one if and only if the two corresponding bits in M and N are different; otherwise exclusive-or puts this bit to zero. Thus this bit-by-bit exclusive-or `IEOR(M,N)` (Cray called it M.xor.N but unfortunately this did not become the standard) treats 32 bits in parallel and does for each of these bits what a logical operation would do for one bit only with logical (Boolean) variables. Obviously such bit-handling operations can be used in lots of problems where the essential information consists of independent bits, such as in Ising models or cellular automata where it is called multispin coding [3]. Fortran manuals usually hide these tricks in an appendix on the functions which the compiler has stored.

Imagine you have an array of 250 integers N consisting of completely random bits. Then the next integer `N(251)` is produced via `N(251) = IEOR(N(1), N(148))` and generally

```
N(K) = IEOR(N(K-250), N(K-103))
```

where again 250 and 103 are magic numbers which should not be changed. An alternative choice is the simple subtraction

```
N(K) = N(K-250) - N(K-103)
```

but this is less widespread than the exclusive-or method, also called R250.

To work with it we first need 250 random integers. It is not recommended to take the results of `IBM*16807` directly as such integers since the last bits are not random enough; for example the least significant bit is always one since `IBM` is always odd. Instead we set a bit in N equal to one if and only if the result of `IBM*16807` is negative. Thus our 32-bit integers N are initialized through

```
DO K = 1, 250
ICI=0
DO I=1,32
ICI=ISHFT(ICI,1)
IBM=IBM*16807
IF(IBM.LT.0) ICI=ICI+1
ENDDO
N(K)=ICI
ENDDO
```

Here again `ISHFT` is a bit-manipulation function shifting the first argument by one bit to the left. Instead of `ICI=ICI+1`, one could also have used a bit-by-bit or-function `ICI=IOR(ICI,1)`; on most compilers integer and bitstring operations