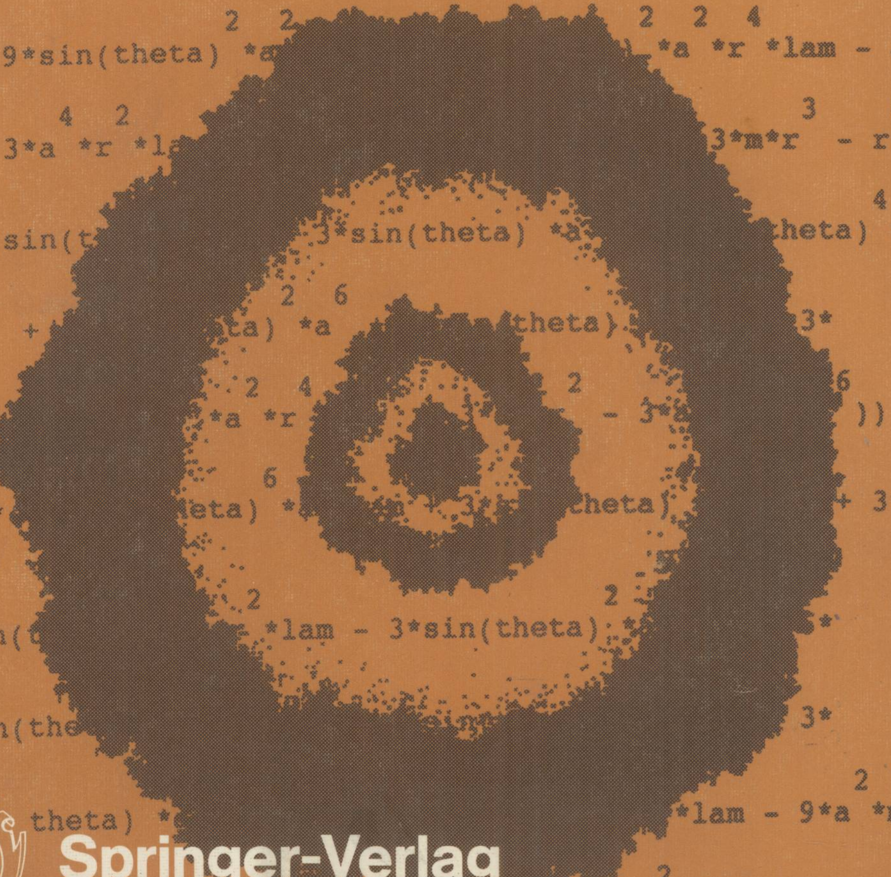
D. Stauffer F. W. Hehl
V. Winkelmann J. G. Zabolitzky

Computer Simulation and Computer Algebra

Lectures for Beginners



Springer-Verlag

TP301
S798

D. Stauffer F. W. Hehl
V. Winkelmann J. G. Zabolitzky

Computer Simulation and Computer Algebra

Lectures for Beginners

With 7 Figures

Springer-Verlag Berlin Heidelberg New York
London Paris Tokyo

Professor Dr. Dietrich Stauffer
Professor Dr. Friedrich W. Hehl
Dipl.-Phys. Volker Winkelmann*
Institut für Theoretische Physik, Universität Köln, Zülpicher Straße 77,
D-5000 Köln 41, Fed. Rep. of Germany
*Rechenzentrum, Universität Köln, Robert-Koch-Straße 10,
D-5000 Köln 41, Fed. Rep. of Germany

Professor Dr. John G. Zabolitzky
Kontron Elektronik GmbH, Breslauer Straße 2,
D-8057 Eching, Fed. Rep. of Germany

ISBN 3-540-18909-2 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-18909-2 Springer-Verlag New York Berlin Heidelberg

Library of Congress Cataloging-in-Publication Data. Computer simulation and computer algebra : lectures for beginners / D. Stauffer ... [et al.]. 1. Computer simulation. 2. Algebra-Data processing. I. Stauffer, Dietrich. QA76.9.C65C656 1988 001.4'34-dc19 88-19977

Printing: Weihert-Druck GmbH, D-6100 Darmstadt
Binding: J. Schäffer GmbH & Co. KG., D-6718 Grünstadt
2156/3150 - 543210

Preface

Computers play an increasingly important role in many of today's activities, and correspondingly physicists find employment after graduation in computer-related jobs, often quite remote from their physics education. The present lectures, on the other hand, emphasize how we can use computers for the purposes of fundamental research in physics.

Thus we do not deal with programs designed for newspapers, banks, or travel agencies, i.e., word processing and storage of large amounts of data. Instead, our lectures concentrate on physics problems, where the computer often has to work quite hard to get a result. Our programs are necessarily quite short, excluding for example quantum chemistry programs with 10^5 program lines. The reader will learn how to handle computers for well-defined purposes. Therefore, in the end, this course will also enable him to orient himself in computer-related jobs.

The first chapter deals mainly with solutions of the Newtonian equation of motion, that force equals mass times acceleration, which is a precursor to the molecular dynamics method in statistical physics. The second chapter considers, by means of several examples, another method for statistical physics, Monte Carlo simulation. These two chapters deal with numbers, the traditional territory of computers. In contrast, analytic formula manipulation, such as $(a+27b^3-4c)^5 = a^5+135a^4b^3-\ldots$, is taught in the last chapter and is important, for instance, in analytic integration or for evaluating expressions in Einstein's general theory of relativity.

All chapters try to convince readers to write their own computer programs for their own needs; it is not our aim that the reader buys software that requires the typing of only a few numbers before the results are produced, since then the students will only be amazed at the competence of the authors. Our aim is to teach them to program at least as well by themselves.

We have taught this course at various universities: repeatedly in Cologne, but also in Minneapolis and Antigonish. Prospective readers should have access to a computer (micro, mainframe, ...) to run their programs, either in batch or in interactive mode. For the first two sections, they should have about 2 years of university physics education whereas the computer algebra course can be understood by any freshman. The languages used here are Fortran, for number crunching, and Reduce, for analytic formula manipulation. Reduce is explained here in detail, and Fortran (summarized in an appendix) can be easily understood even if the reader knows only Basic or Pascal. Numerous high school students, who had never attended a university course, were able to write programs for parts of this course.

The authors come from different backgrounds (nuclear physics, solid state physics, and relativity) and have different programming styles: STRENGTH THROUGH DIVERSITY. Each author agrees, however, that the reader should not trust the advice of the other authors. We thank D. Cremer, C. Hoenselaers, T. Pfenning and W. Weiss for their help in the course and H. Quevedo for TEX-assistance. The Eden clusters in the cover picture were produced by R. Hirsch.

Cologne, April 1988

D. Stauffer, F.W. Hehl
V. Winkelmann, J.G. Zabolitzky

Contents

1. Computational Methods in Classical Physics

John G. Zabolitzky, KONTRON Electronics, 8057 Eching, West Germany

1.1 Preface

It is the aim of this chapter to enable the readers to implement solutions to problems in the physical sciences with a computer program, and carry out the ensuing computer studies. They will therefore be shown a few basic numerical methods, and the general spirit for mapping physics problems onto a computational algorithm. It is advisable to spend some time actually implementing the exercises proposed, since is only by so doing that one may learn about, and get a feel for, the spirit of scientific computing. Examples are given using the FORTRAN 77 language and the UNIX operating system. The graphics interface used is that of the SUN workstation.

1.2 Motion of a Classical Point–Like Particle

The first few examples will deal with problems in classical Newtonian mechanics, in particular with the motion of a single classical point–like particle, described by Newton's law,

$$\mathbf{F} = m\mathbf{a} \qquad \text{(Force = mass * acceleration)}. \tag{1}$$

F and **a** may be taken to have the dimensions of the system under consideration, i.e., if the particle is moving in three–dimensional space, **F** and **a** will be three–vectors, and the particle coordinates are labelled **r** . The derivatives of these coordinates with respect to time are

$$\text{velocity:} \qquad \mathbf{v} = \frac{d\mathbf{r}}{dt} \, , \tag{2}$$

$$\text{acceleration:} \qquad \mathbf{a} = \frac{d^2\mathbf{r}}{dt^2} = \frac{d\mathbf{v}}{dt} \, . \tag{3}$$

The force **F** of (1) is the total force acting on the particle, that is the (vector) sum of all individual forces acting on the particle. Some examples of such individual forces are

$$\text{constant gravitational field} \qquad \mathbf{F} = m\mathbf{g} \, , \tag{4}$$

$$\text{general gravitational field} \qquad \mathbf{F} = \nabla\Phi(r_{12}) \, , \tag{5}$$

$$\text{potential } \Phi = G\frac{m_1 m_2}{r_{12}} \, , \tag{6}$$

$$\text{friction} \qquad \mathbf{F} = k\left(\frac{-\mathbf{v}}{|v|}\right) v^{\alpha} \, . \tag{7}$$

In (7) k is some suitable constant, the expression in parentheses is a unit vector in the direction opposite to the current velocity, and $v = |v| = |\mathbf{v}|$ is the magnitude of the velocity. The exponent α can take on a number of values depending upon the type of friction involved.

Equation (7) is not an exact model since the exponent should really depend on the velocity as well, though this is not considered here as the deviation is small.

Example. Let us consider the movement of a particle in constant gravitational field, (4). Using (4) in Newton's law, (1), yields

$$m\mathbf{g} = m\mathbf{a} \qquad \text{or} \qquad \mathbf{g} = \mathbf{a}, \tag{8}$$

which may not be too surprising. As a differential equation, (8) becomes

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{g} \qquad \text{or} \qquad \frac{d^2y}{dt^2} = -g, \qquad \frac{d^2x}{dt^2} = 0, \tag{9}$$

where the coordinate vector $\mathbf{r} = (y, x)$ has been written using its two components y (elevation) and x (horizontal position).

We have now reduced the (physics) problem of calculating the motion of a point–like particle to the (mathematical) problem of solving a differential equation. In all generality, (1) may be considered as relating a known function, the force given in terms of position, velocity, acceleration etc., to the acceleration (or second derivative of the coordinate). So the general form of (1) is

$$\frac{d^2\mathbf{r}}{dt^2} = \frac{1}{m}\mathbf{F}(\mathbf{r},\ \mathbf{v}, ...). \tag{10}$$

This equation does <u>not</u> specify the path, i.e. the function $\mathbf{r}(t)$, uniquely. This is because we have not specified any initial conditions, or end conditions, or boundary conditions. So far we have only specified a <u>family</u> of functions (= set of all solutions of (10) for specfic F). We need to find the number of parameters required to select one unique solution out of this set. Formally integrating (10) twice,

$$\mathbf{r}(t) = \int_{t_0}^{t} \mathbf{v}(\tau)d\tau + \mathbf{r}_0 \tag{11}$$

$$\mathbf{v}(\tau) = \frac{1}{m}\int_{\tau_0}^{\tau} \mathbf{F}(\tau')d\tau' + \mathbf{v}_0\ , \tag{12}$$

it is seen that we need two constant vectors, $\mathbf{r}_0$ and $\mathbf{v}_0$, to specify the solution completely, that is, initial position and initial velocity of the particle. In two dimensions, these would be four numbers, in three dimensions six. Equivalently, one could ask: What is the path terminating at a given velocity and position?, i.e, integrate backwards in time; or one could ask: What is the path passing through $\mathbf{r}_0$ at $t = 0$ and through $\mathbf{r}_1$ at $t = t_1$? This latter problem would be a boundary value problem (instead of solution and first derivative given at one point, solution given at two different points) and will be discussed in part 4 of this chapter.

Example. Continuing the above example, the solution – carrying out integrations (11) and (12) – is simple since the motion in the x and y directions is independent. The two differential equations of second order for the two scalar coordinates are [(9)]

$$\frac{d^2y}{dt^2} = -g; \qquad \frac{d^2x}{dt^2} = 0. \tag{13}$$

After the first integration we have

$$\frac{dy}{dt} = v_y(t) = -gt + v_y(0); \qquad \frac{dx}{dt} = v_x(t) = v_x(0), \tag{14}$$

with the initial conditions $v_y(0)$ and $v_x(0)$ defining the velocity vector at time $t = 0$. The second integration yields

$$y(t) = -\frac{g}{2}t^2 + v_y(0)t + y(0); \qquad x(t) = v_x(0)t + x(0). \tag{15}$$

In (15) we have the complete solution, given the four initial conditions $x(0)$, $y(0)$, $v_x(0)$, $v_y(0)$. Here we have of course recovered the well–known fact that n differential equations of order m need $n * m$ initial conditions to have a unique solution.

In the general case (10) will not have a closed–form solution which could be derived by analytical means. Let us therefore consider numerical solution to (10). As before, I will substitute the velocities as additional variables, which yield a system of (in general coupled) differential equations of the first order:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \qquad \text{and} \qquad \frac{d\mathbf{v}}{dt} = \frac{1}{m}\mathbf{F}(\mathbf{r}, \mathbf{v}, ...). \tag{16}$$

For simplicity, first consider just a single equation, $y' = f(t, y)$, where the prime is shorthand for d/dt. Given y at some t_0 , we want to find $y(t)$ a little time later, at $t = t_0 + \Delta t$. That is, we break the problem we want to solve (find complete path, i.e. function $y(t)$) into a number of smaller, less difficult problems: find just a little piece of the path. Furthermore, we do not require to know the path at all arbitrary points, but we ask only for the coordinates at a few selected, discrete points in time. This is the first and essential task in order to make any problem tractable numerically: we have to discretize the variables involved in the problem, here the time t. The time t will not run over a continuum of real values, but assume only values out of a given, finite set . In our case this set will be a one–dimensional mesh of time points, $\{t | t = t_0 + i\Delta t, \quad i = 0, 1, 2, ..., max.\ \}$. By repeatedly stepping the current solution through one time interval Δt we may proceed to any finite time desired.

Given y at t_0 (that is, numbers given for these quantities), we can evaluate the function $f(t, y) = y'$ to find the value of y' at that point. We are now left with the task to extrapolate from our current knowledge about the function $y(t)$ to find the value $y(t_0 + \Delta t)$. Since $y(t)$ is the solution of a differential equation that is assumed to exist, we can also assume that $y(t)$ possesses a Taylor expansion around $t = t_0$, i.e.,

$$y(t) = y(t_0) + y'(t_0)(t - t_0) + \frac{1}{2}y''(t_0)(t - t_0)^2 + ... \tag{17}$$

If the time step is made small enough (if $\Delta t = t - t_0$ is small enough) the second and all higher derivative terms in (17) may be neglected. In other words, in a sufficiently small neighbourhood any function may arbitrarily well be approximated by a linear function and in that case we have all the knowledge to calculate $y(t_0 + \Delta t)$:

$$y(t_0 + \Delta t) = y(t_0) + y'(t_0)\Delta t. \tag{18}$$

This method is called the linear extrapolation or Euler method. It is quite obvious that for a finite time step Δt errors of the order Δt^2 are generated: the numerical treatment will not produce an exact solution, but the true solution may be approximated as well as one likes by making Δt small enough. This shows a general feature of numerical methods: the higher the accuracy desired of the results, the more computational work is required, since with smaller time steps Δt a larger number of steps are necessary to traverse a given time interval. The Euler stepping method obviously can be applied repeatedly, and we require only the initial values in order to do the first step. Errors are generated at each step and may build up as we go along. We know from (17) that errors are proportional to the second derivative of the solution – essentially the curvature. So we know where to look for problems! Wherever our final solution is to have large curvature, numerical errors may possibly have come in.

Let us now generalize the method for one function to coupled systems of differential equations like (16). At time t_0, all the right hand sides may be calculated from the known function values. The function values are known either as initial values, or from the last time step. We can therefore write down a Taylor series for each of the components of (16), and proceed with each component independently as we did for the single equation case. The reason is quite simple: all the couplings between the various equations are contained exclusively within the right hand sides, and are simply computed with the known values at the current time point. After that, the equations are really independent and their solution does not pose any more problems than the solution of a single equation.

Algorithm: Linear Extrapolation or Euler Method. A set of ordinary differential equations is written as

$$\frac{dy_i(t)}{dt} = f_i(t, y_1, y_2, ..., y_n), \quad i = 1, ..., n \; . \tag{19}$$

At an initial time $t = t_0$ all function values are provided as initial values, and the function values at time $t + \Delta t$ are calculated from

$$y_i(t + \Delta t) = y_i(t) + f_i(t, y(t)) * \Delta t, \quad i = 1, ..., n. \tag{20}$$

The method is applied repeatedly to integrate out to large times, giving as error proportional to the second derivative times Δt^2 . Two points should be noted:

1. Higher–order ordinary differential equations may be transformed into the form of (19) by substituting new variables for the higher derivatives, in exactly the same way as substituting the velocity in our physical example.
2. Quite clearly this is the simplest method possible for solving differential equations. More involved methods with higher accuracy will be discussed later.

Problem No. 1: Throw the Ball !

Consider the two–dimensional ($2d$) coordinate system y(up)$-x$(right). At the origin of this coordinate system a ball of mass 1kg is thrown with velocity v and angle theta with respect to ground. Gravitational acceleration is taken to be 10 m/sec^2 and the frictional force due to movement in a viscous medium is given by

$$kv^{1.87}, \quad \text{with} \quad k = 0.5 \text{ kg/sec (m/sec)}^{-0.87} \; .$$

Where is the ball at $t = 2$ sec it is thrown with a velocity of 70 m/sec at an angle of 44 degrees ? (Hint: use smaller v first!) Write a subroutine to calculate the path of flight. The subroutine is to be called with three arguments:

```
subroutine nstep (dt, n, y) ,
dimension y (4, n) ,
```

where dt is the timestep to be used and n is the number of timesteps to be taken plus one. The array y holds initial conditions as well as the solution, so that y (all, 1) are the initial conditions, y(all, 2) are positions and velocities after one timestep, ..., y(all, n) are positions and velocities after (n-1) timesteps. The four components are then assigned as

y(1,t) = y-coordinate	y(2,t) = x-coordinate
y(3,t) = y-velocity = y'	y(4,t) = x-velocity = x' .

Use the linear extrapolation method to obtain the trajectory of the ball.

Input: dt, n, y(all, 1) (corresponding to t=0)
Output: y(all, 2...n) (corresponding to t=dt,...,(n-1)*dt)

Theory.
We have Newton's law, $\mathbf{F} = m\mathbf{a}$. Acceleration $\mathbf{a}$ is the second derivative of the coordinates with respect to time, $\mathbf{a} = \mathbf{x}''$. We therefore have

$$\frac{d^2\mathbf{x}}{dt^2} = \frac{1}{m}[-mg\mathbf{e}_y - kv^{1.87}\mathbf{e}_v],$$

where $\mathbf{x}$ is a $2d$ vector of current position, the left hand side is therefore the $2d$ acceleration vector, m is the mass of the ball, g is the gravitational acceleration, $\mathbf{e}_y$ is a unit vector in the y direction, k is the constant of friction, v is the magnitude of the current velocity, and $\mathbf{e}_v$ is the unit vector in the direction of current velocity. The first term in brackets is the gravitational term, the second term comes from the friction. The direction of the gravitational force is in the negative y direction, the direction of the frictional force is opposite to that of the velocity.

Implementation.
Using the above constants and equation, it is straightforward to write down the right hand side of the derivative vector, $y' = ...$, $x' = ...$, $y'' = ...$, $x'' = ...$ as derivatives of y, x, y', x'. We therefore have the problem in the canonical form and can use the algorithm given above. Collect your subroutine(s) in any file with extension .f. (e.g., myball.f). The procedure to translate, load and execute the program is execball. You therefore type "execball myball" .

```
jgz%
jgz% cat execball
f77 -o $1  -f68881 -O $1.f  /u/jgz/cpc/scaff1.o -lcore77
                                  -lcore -lsunwindow -lpixrect -lm
$1
jgz%
jgz%
```

Provided Scaffold for Problem # 1.

```
      subroutine init (dt,y,tmax,nmax,n)
      dimension y(4)
c
c this subroutine initializes the ball fly problem
```

```
c   by obtaining user input
c
    2 write (*,*) 'enter end-time'
      read (*,*) tmax
      if (tmax .le.  0.0) then
            write (*,*) 'illegal end-time, must be > 0'
            goto 2
      endif
    1 write (*,*) 'enter time step'
      read (*,*) dt
      if (dt .le.  0.0) then
            write (*,*) 'illegal time step, must be > 0'
            goto 1
      endif
      if (dt .gt.  tmax) then
            write (*,*) 'illegal time step, > tmax'
            goto 1
      endif
      n=tmax/dt+0.1+1.
c added 1 for initial t=0 storage
      if (n .gt.  nmax) then
            write (*,*) 'too many time steps'
            goto 1
      endif
c
    3 write (*,*) 'enter velocity'
      read (*,*) v
      if (v .le.  0.0) then
            write (*,*) 'illegal velocity, must be > 0'
            goto 3
      endif
    4 write (*,*) 'enter angle in degrees'
      read (*,*) angdeg
      if (angdeg .le.  0.0 .or.  angdeg .ge.  90.0) then
            write (*,*) 'illegal angle, must be > 0 and < 90'
            goto 4
      endif
      angrad=angdeg*3.141592654/180.0
c
      y(1)=0.0
      y(2)=0.0
      y(3)=v*sin(angrad)
      y(4)=v*cos(angrad)
c
      return
      end
      program ball
c
c solves ball fly problem
c
      parameter (nmax=10000)
      dimension y(4,nmax)

c get input
    1 call init (dt,y,tmax,nmax,n)
c
c document input
      write (*,*) 'solving ball fly problem for dt=',dt
      write (*,*) '                     up to tmax=',tmax
      write (*,*) '            resulting in nsteps=',n
      write (*,*) '         for initial velocities=',y(3,1),y(4,1)
c
c call problem solution code
      call nstep (dt,n,y)
c
c write out results
      write (*,234) (n-1)*dt,(y(i,n),i=1,4)
  234 format (' at tmax=',f10.3,'  y=',f15.6,'  x=',f15.6,/,
     *         19x,             ' vy=',f15.6,' vx=',f15.6)
c
c draw graph of flight path
      call plobal (y,n)
      goto 1
      end

        subroutine plobal (y,n)
c plot results from ball problem
        dimension y(4,n),xx(10000),yy(10000)
        do 1 i=1,n
        xx(i)=y(2,i)
        yy(i)=y(1,i)
    1   continue
c call standard plotting routine to do the nitty--gritty
        call plotfu(xx,yy,n,1,1,-.5,10.5,-3.,5.)
        return
        end

jgz% cat solv1.f
      subroutine derivs (t,y,dydt)
      dimension y(4), dydt(4)
c
c this subroutine computes the right-hand- sides for
c   ball fly problem
c variables are  y(1)=y  y(2)=x  y(3)=y'  y(4)=x'
```

```
c r.h.s are y'=y'  x'=x'  y''=-g -y'cv**.87  x''=0 -x'cv**.87
c
c first, compute velocity
      v=sqrt(y(3)**2+y(4)**2)
      cv87=0.5*v**0.87
c
      dydt(1)=y(3)
      dydt(2)=y(4)
      dydt(3)=-10.0 -y(3)*cv87
      dydt(4)=0.0   -y(4)*cv87
c
      return
      end

      subroutine tstep (t0,dt,y0,y)
      dimension y0(4), y(4), dydt(4)
c
c this subroutine steps the vector y through one time step,
c   from t0 to t0+dt
c
      call derivs (t0,y0,dydt)
      do 1 i=1,4
    1 y(i)=y0(i)+dt*dydt(i)
c
      return
      end

      subroutine nstep (dt,n,y)
      dimension y(4,n)
c
c this subroutine solves the ball fly problem for n-1 time steps
c   given initial conditions at t=0 in y(*,1)
c
      t=0.0
      do 1 i=2,n
      call tstep (t,dt,y(1,i-1),y(1,i))
    1 t=t+dt
c
      return
      end
```

```
jgz% cat examp1.out
 solving ball fly problem for dt=    1.00000e-03
                       up to tmax=   2.000000
              resulting in nsteps=  2001
             for initial velocities=   48.62609   50.35379
 at tmax=     2.000  y=      -0.201220  x=       7.995113
                    vy=      -4.889576 vx=       0.309135
```

```
      solving ball fly problem for dt=    3.00000e-04
                             up to tmax=   2.000000
                    resulting in nsteps=  6667
                 for initial velocities=   48.62609   50.35379
     at tmax=      2.000   y=        -0.183019  x=        8.009531
                          vy=        -4.888783 vx=        0.310836
   jgz%
```

It is seen clearly that the numerical result depends upon the stepsize used, as is to be expected from the previous discussion. The difference gives some indications of the numerical error in the final result.

1.3 Short Course in FORTRAN Programming Methodology

You want to obtain good solutions to a problem as fast as possible and there are a number of programming methods which will help you to do so. The most important consideration is to keep your thoughts clean, modular and hierarchical. The only way humans can solve complex problems is by means of breaking them down into smaller ones. This is applied recursively until you finally reach the level of trivialities: the problem is solved. In exactly the same way you should construct your programs: define blocks which attack a particular task. In order to solve some well–defined task, one such block will have to call upon other well–defined tasks. These blocks of work should be made into FORTRAN subroutines (or functions). As a rule no subroutine should be more than about one (with an absolute maximum of two) pages of FORTRAN. The less skilled you are, the smaller the subroutines should be. For a beginner 5–10 lines of code is a reasonable size.

One subroutine should correspond to one small, well–defined piece of work which may be trivial or elementary; in this case, we have a lowest–level subroutine in front of us which does not call any others to do some work for it. On the next level, more complex tasks may be accomplished by another small routine calling some lower–level routines to do logical pieces of work. This partitioning should be done on a basis of logical connectednes: keep things together which belong together, and do not mix together different tasks.

This information about subroutines applies equally well to data structures: the lowest level is the machine word, usually a floating–point real number or an integer. These individual objects may be grouped together into one–dimensional arrays.

element	array
a(1)	dimension a(100)
....	do 1 i=1,100
a(100)	1 a(i)=1./i

Of course, you only group together in an array data that is logically connected, like components of a vector, or values of a function at a number of points, like the $1/x$ function in above example. In the first problem the four elements of the solution vector are grouped together:

$$y,\ x,\ v_y,\ v_x \longrightarrow y(1),\ y(2),\ y(3),\ y(4).$$

At a <u>fixed</u> time t, these four data elements define a state of our system (the ball in this case).