

Introduction to Computational Physics

SS 05

Original author of this guide (2002): Reimer Kühn
Institut für Theoretische Physik, Universität Heidelberg
Philosophenweg 19, 69120 Heidelberg
E-Mail: kuehn@tphys.uni-heidelberg.de
Tel.: 06221 – 549436
<http://www.tphys.uni-heidelberg.de/~kuehn/>

with some additions, omissions and editions (2004, 2005) by:

Rainer Spurzem, Astron. Rechen-Inst. Heidelberg
Mönchhofstr. 12-14, 69120 Heidelberg
E-Mail: spurzem@ari.uni-heidelberg.de
Tel.: 06221 – 405230
<http://www.ari.uni-heidelberg.de/mitarbeiter/spurzem/>

Lecturer: R. Spurzem

Contents

1	Motivation	1
1.1	Literature, Sources	2
2	Computers and Numbers	4
2.1	Components of a Computer	4
2.2	Data Representation	7
2.3	Precision, errors, stability	9
3	Practical Hints	12
3.1	Software Engineering	12
3.2	Programming Tools	14
3.2.1	Macros and Directives	15
3.2.2	Pre-Processors and Compilers	16
3.2.3	Makefiles	17
3.2.4	Unix Shells	18
3.2.5	Graphics	19
3.2.6	Mathematica	20
4	Modeling Physics Problems	24
4.1	The Two-Body Problem	24
4.1.1	Elementary Facts	24
4.1.2	Elementary Numerical Solution	25
4.1.3	Leap Frog or Verlet: the next step	27
4.2	Population Dynamics	29
4.3	Interacting Populations	32

5	Linear Algebra	38
5.1	Introduction	38
5.2	Linear Equations	41
5.2.1	Gauß-Jordan Elimination	42
5.2.2	LU-Decomposition	44
5.2.3	Cholesky-Decomposition	47
5.3	Eigenvalue problems	48
5.3.1	Coupled Vibrations	48
5.3.2	Algorithms	55
5.3.3	Linear Variational Methods of QM	63
6	Solving Ordinary Differential Equations (ODEs)	71
6.1	Introduction	71
6.2	Elementary Algorithms	72
6.2.1	Euler-Integration	74
6.2.2	Taylor-Expansion Method	74
6.2.3	Runge-Kutta Methods	75
6.3	Generalized Runge-Kutta Algorithms	79
6.4	Adaptive Step-size Control; Bulirsch-Stoer Algorithm	81
6.5	Duffing-Oscillator as Example	83
6.6	Numerov-Algorithm	87
6.7	Molecular Dynamics	88
6.7.1	Lennard-Jones Systems	90
6.7.2	Integration Methods	92
6.7.3	Measurements – Thermodynamic Functions – Structure	95
6.7.4	Ensembles	98

7	Discrete Dynamical Systems and Chaos	101
7.1	Motivation	101
7.2	Discrete Dynamics	104
7.3	The Logistic Map	107
7.4	Periodically Kicked Rotator	111
8	Random Numbers	119
8.1	Generation of Homogeneously Distributed RNs . . .	120
8.1.1	System-Provided Generators	120
8.1.2	Linear Congruential Generators (LCGs) . . .	121
8.1.3	Portable Generators	123
8.2	Generation of RNs with Prescribed Probability Den- sity	125
8.2.1	Transformation-Method	125
8.2.2	Example: Exponentially Distributed RNs . .	126
8.2.3	Gaussian RNs – Box-Muller Algorithm . . .	127
8.2.4	Rejection-Method	128
9	Monte Carlo Simulation	132
9.1	Monte Carlo Integration: Evaluation of Integrals and Expectations	132
9.1.1	Importance Sampling: Metropolis Algorithmus	134
9.1.2	Random Walk	136
9.1.3	Discrete Systems as Example	138
9.1.4	Aspects of the Simulation	143
9.1.5	Estimation of Errors – The Role of Dynamics and critical Slowing Down	145

1 Motivation

Introduction to use of computers in physics for problems

- that are not solvable analytically
- that are not solvable analytically by elementary means.

Support

- in mathematical analysis (symbolic mathematics, numerics)
- in the analysis of results (statistics, visualization)

Numerically oriented mainly in problems of statistical physics / field theory, astrophysics etc. Computer-aided symbolic math mainly in small-scale problems. Strengths and weaknesses largely complementary.

We shall learn (i) a few important strategies and survival-rules, (ii) and discuss a few elementary algorithms – mainly by way of examples from physics and related areas.

Do use libraries (IMSL, NAG, Numerical Recipes); don't re-invent the wheel. But don't use these tools completely in the “black box mode”. Main emphasis is to teach methods and mature use of available tools.

1.1 Literature, Sources

- Literature
 - *Numerical Recipes in C/Fortran/Pascal*,
W. Press et al. (Cambridge University Press)
 - *Einführung in die Numerische Analysis*,
J. Stoer und R. Bulirsch (Springer)
 - *Physics by Computer*,
W. Kinzel und G. Reents (Springer)
 - *Computational Physics*,
M. Thijssen (Cambridge University Press)
 - *Mathematica*,
S. Wolfram (Cambridge University Press)
 - *The Art of Computational Science*,
P. Hut and Jun Makino (<http://www.artcompsci.org>)
 - *Perspectives of Nonlinear Dynamics*,
E. Atlee Jackson, Cambridge Univ. Press
 - *Order and Chaos in Dynamical Astronomy*,
G. Contopoulos, Springer
 - *A practical guide to computer simulations*,
Alexander K. Hartmann (<http://arxiv.org/abs/cond-mat/0111531>)
- Algorithms in the web (public domain, source codes)
 - <ftp://ftp.uni-stuttgart.de>
 - <http://www.netlib.org>
 - <http://www.gams.nist.gov>
 - <http://www.gnu.org>

- Commercial Libraries:
 - NAG (numerical)
 - IMSL (numerical)
- Symbolic Mathematics:
 - MAPLE
 - MATHEMATICA
 - MATLAB

2 Computers and Numbers

2.1 Components of a Computer

- **CPU:** active element
(instruction-set, register architecture & cycle-time)
 - executes programs stored in memory
 - words in memory contain instruction code and address of operands
 - program counter: special register, contains address of next executable instruction
 - normally just incremented, exception: conditional execution, loops, goto's etc.
- **register:** special memory integrated into CPU for data and addresses (1 register = 1 word)
 - number of registers typically small (8 ... 64)
- **memory:** fast mass storage for programs and data, hierarchically organized
 - Cache (close to processor and very fast, mostly on CPU chip)
 - normal memory (RAM) connected via FSB (front side bus) 533 MHz \approx 2 GB/s
 - approx. size today (2004): 512 MB - several GB for PCs, \approx 1 TB = 1024 GB for mainframes
- **chipset and data bus:** controls CPU-memory and memory-PCI bus connection (e.g. E7505, i870, 440BX)
 - FSB (Front Side Bus) CPU-memory connection (North Bridge)

– PCI (Periph. Component Interconnect) memory - external connection e.g. graphics card, hard disk, network (South Bridge)

- **external connection:** hard disk storage, other devices (CD/DVD/tape), network connection, e.g. via PCI-X bus 133 MHz \approx 500 MB/s

- **I/O:** monitor, keyboard, mouse

Example 2.1:

scalar product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

compute and store in data register D2. Assembler-Code:

```
CLR      D2          clear data register 2
MOVE.W  998,D0       dimension n of vectors in 998 and 999 (2 byte)
MOVE.L  #1000,A1    starting address 1000 of a to addressreg. 1
MOVE.L  #2000,A2    starting address 2000 of b to addressreg. 2
LOOP:   MOVE.W  (A1)+,D1  a-component from (A1) to datareg. 1, increment A1
        MUL     (A2)+,D1  mult. b-component from (A2) with a component ->D1
        ADD.W  D1,D2     sum in D2
        SUB.W  #1,D0     reduce dimension
        BNZ   LOOP      to LOOP, if things left to do
```

MOVE.W MOVE.L means moving word or long word (2 words).

Timing:

```
ADD.W   D0,D1        4 cycles
ADD.W   (A1),D0      8 cycles (memory access needed)
ADD.L   (A1),D0      14 cycles (memory access needed twice)
ADD.L   123456,D0    22 cycles
MULU    D0,D1        70 cycles (max.) for unsigned integer mult.
```

2.2 Data Representation

Units

- **bit**: binary digit, smallest information unit $\{0,1\}$, $\{a,b\}$, $\{\text{low, high}\}$ voltage.
- **byte**: 1 byte = 8 bit, 1kB = 1024 byte , 1MB = 1024kB, 1 GB = 1024 MB etc.
- **word**: machine dependent today mostly 32 bit (Intel Pentium, AMD), or 64 bit (Intel Itanium, AMD Opteron)

Data representation usually bit-wise or binary. Smallest addressable unit in memory: 1 byte. Computer word of length n can encode 2^n different objects.

Numerical

- **CHARACTER** (Letters numbers and special characters) coded in a byte; for character codes (e.g.. ASCII)

$$\text{char} \in \{0, \dots, 255\}$$

- **INTEGER** (integer numbers) For computer word of length

n :

(i) pos. int. number $N \in \{0, 1, 2, \dots, 2^n - 1\}$

(ii) int. number: $Z \in \{-2^{n-1}, \dots, 0, 1, 2, \dots, 2^{n-1} - 1\}$

e.g.: $n = 16$

$$-32768 \leq Z \leq 32767 \quad 0 \leq N \leq 65535$$

e.g.: $n = 32$

$$-2147463648 \leq Z \leq 2147463647 \quad 0 \leq N \leq 4294967295$$

- **FLOATING POINT** (Decimals) Organization depends on word length. For **32 bit**:

$$\begin{aligned}
 f &= (d_0, d_1, d_2, \dots, d_{23}) \cdot 2^e \\
 &\equiv (d_0 \cdot 2^0 + d_1 \cdot 2^{-1} + d_2 \cdot 2^{-2} + \dots + d_{23} \cdot 2^{-23}) \cdot 2^e .
 \end{aligned}$$

with e an 8 bit integer, $e = e_0 - 127$, and $e_0 = 0$ and $e_0 = 255$ represent $f = 0$ and $f = \infty$; so for finite non-zero f $-126 \leq e \leq 127$. Convention: $d_0 \equiv 1$ for normalization. thus d_0 free as sign bit.

- largest representable f

$$f_{\max} = (1, 1, 1, 1, \dots, 1) \cdot 2^{127} \simeq 3.40 \cdot 10^{38} .$$

- smallest representable f

$$f_{\min} = (1, 0, 0, \dots, 0) \cdot 2^{-126} \simeq 1.18 \cdot 10^{-38}$$

- machine precision ε_m : smallest Increment of mantissa
 $\varepsilon_m = 2^{-23} \simeq 1.19 \cdot 10^{-7}$, consequence:

$$1 + \varepsilon = 1$$

for $\varepsilon < \varepsilon_m$!

For **64 bit** words: bits d_0, \dots, d_{52} for mantissa and 11 remaining bits for exponent. This gives $f_{\max} \simeq 1.8 \cdot 10^{308}$, $f_{\min} \simeq 2.23 \cdot 10^{-308}$ and $\varepsilon_m \simeq 2.2 \cdot 10^{-16}$

Consequences: Numerical mathematics is not mathematics

- Not every sum representable (same for products, differences, quotients)
- Can have

$$a + b = a$$

for $b \neq 0$.

- addition commutative but not necessarily associative. For $\varepsilon < \varepsilon_m$ we have

$$-1 + (1 + \varepsilon) = 0$$

but

$$(-1 + 1) + \varepsilon = \varepsilon$$

- **Rounding errors !**

2.3 Precision, errors, stability

Computation with rounding errors ε_r in each floating-point operation. Estimate: $\varepsilon_r = \pm\varepsilon_m$, (accumulation *diffusively*, N_{op} steps)

$$\varepsilon_{\text{tot}} = \mathcal{O}(\sqrt{N_{\text{op}}} \varepsilon_m)$$

At 10^8 floating-point operationen/sec in a program of 1 h CPU-time

$$\varepsilon_{\text{tot}} \sim 6 \cdot 10^5 \varepsilon_m$$

With $\varepsilon_m \simeq 10^{-7}$ this can be insufficient !

Estimate could even be overly optimistic for two reasons:

- (i) rounding errors can be much larger than relative order ε_m !
- (ii) errors can be amplified!

Example 2.2

Solution of $ax^2 + bx + c = 0$:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If $ac \ll b^2$, so $b^2 - 4ac = b^2$ due to finite machine prec., computer gives $x_1 = 0$ instead of

$$x_1 = \frac{-b + b\sqrt{1 - 4ac/b^2}}{2a} \simeq \frac{-b + b(1 - 2ac/b^2 + \dots)}{2a} \simeq -\frac{c}{b}$$

which can be $\mathcal{O}(1)$!

Example 2.3

Compute integral

$$y_n = \int_0^1 dx \frac{x^n}{x+a}$$

Seemingly clever idea: recursive evaluation via

$$y_n = \int_0^1 dx \frac{x^{n-1}(x+a) - ax^{n-1}}{x+a} = \frac{1}{n} - ay_{n-1}$$

and initial condition $y_0 = \ln[(1+a)/a]$. For $a > 1$ this is an *unstable algorithm* (error amplification). Double-precision arithmetic gives for $a = 5$: $y_{22} \simeq -0.192$, .. $y_{27} \simeq 624.5$, i.e. blatant nonsense (Details depend on compiler machine ..).

Way out: backward iteration is stable ! Error in (unknown initial condition vanishes after sufficiently many backward steps.

- \Rightarrow : Survival-rule 1: keep stability in mind. Obvious or mathematically correct (even elegant) things can produce utter nonsense (and fast).
- \Rightarrow : Survival-rule 2: See whether your results make sense. Check computation in simpler limiting cases for which you know the results.

3 Practical Hints

This section was inspired by “A practical guide to computer simulations”, by A.K. Hartmann, H. Rieger, <http://arxiv.org/cond-mat/0111531>, and is a short extraction of important pieces from there.

3.1 Software Engineering

Never just start writing code! Think:

- What is the input?
- Which results you want to obtain?
- Which quantities are needed for further analysis?
- Can you define objects, stand-alone modules?
- Foresee later extensions (data space, names!)
- Which parts of the algorithm can be solved by standard procedures, e.g. from libraries?

Designing and Testing:

- Data Structure
- Task Structure

- Provide error messages and short documentation printout for wrong or no inputs. Think about the innocent user. Provide messages of progress.
- Documenting the code. Brief comments are better than no comments. Long comments are even better. Headers for sub-routines and functions.
- Testing the code. Do test for known solutions. Check plausibility of results (order of magnitude, sign, steady behaviour).

Object-Oriented Software Development:

- Objects and Methods
- Data Capsuling

Object Oriented Programming Languages (like e.g. C++) are not equivalent to Object Oriented Programming!

Traditional Code can be written in C++, and object-oriented programming style can be used in classic languages such as Fortran,C.

Programming Style:

- Split the code into modules, in different files.

- Separate data structures in header files (.h).
- Use meaningful names which relate to objects or their function. Do not use too short names.
- Use indentation and proper line spacing to ease code reading
- Avoid goto-jumping
- Do use global variables only with great care and only if it is a huge economical gain, generally avoid them

Survival Rule 3: one letter variable names plus zero comments = spaghetti code!

3.2 Programming Tools

Elements of Programming:

- Source Code with Macros (.cc .C .F) \Leftarrow **C-Preprocessor**
- Clean Source Code (.cc .c .f) \Leftarrow **Compiler**
- Header Files (.h .f) (User and System)
- Object Code (.o) \Leftarrow **Loader or Linker**
- Object Libraries, Shared Object Libraries (.a .so) (User and System)
- Executable File (e.g. a.out)

- Running the Code (./a.out , take care of path!)
- Using Input / Output Redirection (./a.out ; in ; out)

3.2.1 Macros and Directives

Macros are processed in the pre-processing stage of a compiler. It is defined by

```
#define   name  definition
```

Macros can be used in a very powerful way, very much like a small programming language processed by the compiler pre-processor. A often used simple function of macros is to define them via a compiler option (see below) and then e.g.:

```
#ifdef SINGLE
```

```
(1)  ...
```

```
#endif
```

```
#ifdef PARALLEL
```

```
(2)  ...
```

```
#endif
```

If the compiler is invoked with the `-D` option, like `gcc -DSINGLE ...` the Macro is defined such that the first portion of code is compiled only.

Survival Rule 4: Use Macros with care. Debugging and error search may be more complicated, because not all of your code is actually compiled!

3.2.2 Pre-Processors and Compilers

The C-Preprocessor (`cpp`) can be used also for Fortran-Programs, to process Macros. It is usually included in standard Compilers such as `gcc`, `g++`, `g77`, `cc`, `c++`, `f77`, `f90`. Some Unix Compilers from special companies can be useful (e.g. Intel or Pallas Compilers, `ifc`, `pgf`). Many Computer vendors have their own compiler versions and names (IBM: `xlf...`). Modern compilers automatically include the loader or linker step, if not told otherwise.

Compilers provide a large number of options, of which the most important are:

- `-c` Compile, but do not load/link.
- `-o` name of executable file, to avoid `a.out`
- `-lname` search for `libname.a` in search path
- `-L` define directories to search for libraries
- `-I` define directories to search for header files
- `-D` define pre-processor macro
- `-On` define code optimising level

- **-fast** fastest possible optimisation (machine and compiler dep.)
- **-g** useful for debugging (**generally conflicts with -O!!**)

3.2.3 Makefiles

make is a magical Unix command, which evaluates rules from a file named **Makefile**. With **make -fname** you can use any other file name rather than **Makefile**.

Makefiles are unique in the sense that they do not support sequential logics. First all rules of a Makefile are evaluated and then action taken. The order of rules has little relevance. Makefiles are best studied by starting from simple examples. They assist in keeping track of a large number of files and subroutines. By evaluating the dates of last changes they are acting only on recently changed objects.

A Makefile rule is coded by

```
target: sources  
⟨TAB⟩ command(s)
```

The first line defines the dependencies. The second and possible continuation lines **MUST** start with a ⟨TAB⟩

An example:

```
simulation.o: simulation.c simulation.h
<TAB> cc -c simulation.c
```

It means that whenever `simulation.c` or `simulation.h` have been changed `simulation.o` has to be recompiled. By typing just the command `make` in the UNIX line this job will automatically be done. A Makefile can use variables (similar to shell variables), and can contain several targets, e.g.

```
CC=gcc
simulation.o: simulation.c simulation.h
<TAB> $(CC) -c simulation.c
myprog: simulation.o
<TAB> $(CC) -o simulation.out simulation.o
```

By typing `make myprog` the Makefile first checks whether `simulation.o` needs an update, and then creates a new `simulation.out` if necessary. Here the use of a variable `CC` has been demonstrated, which can be used to select different compilers at the beginning of a Makefile. Variables from Unix shells can be inherited to Makefiles, but note the different syntax.

3.2.4 Unix Shells

On top of the Unix Operation System (usually called the kernel program) various other so-called shell programs are running, which provide convenient functions for the user. The most common ones

are **bash** (Bourne Again Shell) and **tcsh** (C like shell). Both shells provide useful commands, a script programming language and so-called in-line command editing functions. Unfortunately the syntax of everything differs between them (variable usage, aliases, program statements for scripts...). When typing **ps** you can find out what is your present shell.

3.2.5 Graphics

A most easy to start yet powerful graphics program is called **gnuplot**. You can do a virtually infinite number of tasks with it, it has a good online help, and it can be used to plot mathematical functions, plot data from files, do arithmetic operations with data, convert into different file formats, even to create movies...

Here only some basic functionalities are demonstrated to give a start. After typing **gnuplot** in the Unix shell command window one gets the **gnuplot>** prompt:

```
gnuplot> plot sin(x)
```

plots the function with some default values for the data ranges.

```
gnuplot> plot 'name' using 1:2 with lines
```

plots the first and second column of data from a file.

```
gnuplot> plot 'name' u 1:2 w l
```

is a shortcut.

```
gnuplot> plot 'name' u 1:($2+$3) w l,
```

```
'' u :(sqrt($4)) w p
```

plots from the file the sum of column 2 and 3 as a function of column 1 (with a line) and plots the square root of column 4 as a function of line number, from the same file.

```
gnuplot> plot 'name' u 1:2 w l, x**2
```

plots the data from the file and the function x^{**2} .

```
gnuplot> save 'name'
```

saves all parameters in a gnuplot command file, which then can be used by

```
gnuplot> load 'name'
```

to reproduce exactly what one had before. A few useful examples of parameter set commands (many more to be found in

```
gnuplot> help
```

are:

```
gnuplot> set logscale x
```

```
gnuplot> set nologscale y
```

```
gnuplot> set xrange [0:100]
```

```
gnuplot> set ylabel "string"
```

3.2.6 Mathematica

Mathematica is a convenient symbolic algebra software package, which also can do extensive numerical calculations (arbitrary precision, but not fast) and good graphics. Mathematica can be called by **math** to get an inline command prompt (like in gnuplot), or by **mathematica** to get a collection of X-Windows (Graphical User Interface, GUI). In these X-Windows there are command functions, help functions, and a command window in which one can write and evaluate expressions. There are books and literature on Mathematica, the help facility is extended, and there is a lot of online information at <http://mathworld.wolfram.com/>.

The command window is used to type some tasks or operations, and evaluate them, for which we give here just a very small set of examples:

```
Solve[x^3-10x^2+21x-10==0,x]
```

By pressing **SHIFT+ENTER** the expression is evaluated. Other examples:

```
Plot[Sin[x], {x, 0, Pi}]
```

```
Integrate[x^n, {x, 0, 1}]
```

```
Solve[x^2+x+1==0,x]//N
```

The last example first solves the quadratic equation analytically, and then evaluates the complex numbers with limited precision (**N=numeric**), as it would be done by a normal computer code.

With

```
n=2
```

```
Clear[n]
```

values can be assigned to variables; after setting **n=2** any occurrence of *n* will be automatically substituted by the value of 2, until the variable is cleared with the **Clear** command. Note that “=” assigns values to variables, while “==” is used to define mathematical equations.

Rather than typing Mathematica commands, templates from the X-Command Panel can be selected with the mouse, and filled interactively. With the Pull-Down Menu File you can save the results obtained so far in a file (.nb), which is called a Mathematica Notebook. This can be printed nicely, sent to someone else by e-mail, and reused in Mathematica later. Here are some more complex examples how to create solutions of the Volterra-Lotka system, and how to create the plots shown above.

```
In[1]= NDSolve[{u'[t]==u[t](1-v[t]),v'[t]==0.5 v[t](u[t]-1),u[0]==1,v[0]==3},
              {u,v},{t,0,40}]
Out[1]= {u -> InterpolatingFunction[{{0., 40.}}, <>],
         v -> InterpolatingFunction[{{0., 40.}}, <>]}
In[2] = Plot[{Evaluate[u[t] /. %1], Evaluate[v[t] /. %1]}, {t,0,40}]

[...]

In[nn] = ParametricPlot[ {{Evaluate[{u[t], v[t]} /. %1],
                          {Evaluate[{u[t], v[t]} /. %2]}, {Evaluate[{u[t], v[t]} /.
%3]},
                          {Evaluate[{u[t], v[t]} /. %4]}}}, {t,0,40}]
```

Explanation: the **NDSolve** command creates an object, which contains the numerical interpolation of the solution of the ODE given to **NDSolve**. This object has to be evaluated, before it can be plotted. While **Plot** creates a standard function plot, **Parametric Plot** can be used to plot the trajectories of the Volterra-Lotka system. After [...] I have assumed, that four different objects have been created before by **NDSolve** for different values of the initial condition $v(0) = 2, 3, 4, 5$. The number after the percent sign is used in the **ParametricPlot** line to refer to these different solution, by setting %n equal to the Out[n] number of the previous lines in the mathematica notebook.

Further useful mathematica commands for matrix operations and linear algebra:

```
M={{ 0,1 },{1,2}}
```

```
v={1,4,7,6,5,3}
```

creation of an object containing a 2x2 matrix or a 6 element vector

```
M = Table[1/(i+j-1),{i,10},{j,10}]
```

create an object containing a 10x10 matrix, whose elements are given by the formula $1/(i + j - 1)$.

```
Evaluate[Eigenvectors[M]]//N
```

numeric evaluation of eigenvectors of M

```
Eigenvalues[M]
```

get eigenvalues of M

```
eigen = Eigenvalues[M]
```

```
v2=eigen[[2]]
```

create object **eigen** containing eigenvalues of M and then take second element of **eigen** and assign it to variable **v2**

```
Inverse[M]
```

```
Norm[M]
```

```
M.Inverse[M]
```

```
M.v
```

```
v.v
```

diverse matrix operations such as getting the inverse, the norm, matrix multiplication of M with its inverse matrix, matrix-vector multiplication, scalar product.