1. **GAP**, which stands for **Groups, Algorithms and Programming** is a remarkable environment for doing discrete algebra (no limits, please!) – it is of excellent quality, it is very well documented, and it's free.

2. The **GAP** startup depends on your system. Typically under Unix/Linux you type 'gap' after the usual prompt:

   ```
   ~ :  gap
   ```

3. Once in **GAP** you will get a new prompt:

   ```
   gap>
   ```

   GAP is interactive - you type something legal after this prompt and press 'ENTER'. **GAP** then takes your instructions, works on them (perhaps a long time), then responds with an answer, followed by a prompt for the next instruction.

4. Almost every command you type must be followed by a ';' ,which is used to signal the end of your request. For example, here is a typical portion of a **GAP** session:

   ```
   gap> 25 - 13;
   12
   gap> # that was subtraction.  This line, beginning with #, is
   a comment;
   gap> 2^12; # anything following a # is taken as commentary;
   4096
   gap> # Thus comments don't produce output;
   gap> # In fact, comments don't need to end with a semi-colon
   gap> # see??
   gap> # to quit GAP we enter
   gap> quit;
   ```

5. Sometimes **GAP**'s response is suppressed, either by you our the system.

   In particular, you must tell **GAP** to quit, as shown above.

6. **GAP** has a lot of on-line help. Probably the best way is to have at hand a paper copy of the relevant pages of the manual, which is huge.

   Or you may want to open a browser like Netscape. In our system, you should point to the file

   /usr/local/gap4r2/doc/htm/index.htm

7. Finally, you may also seek help from within your GAP session, as below:

```
gap> # I want to find the factors of 123454321
gap> Factor(123454321);
Variable:  'Factor' must have a value
gap> # Hmmm - something went wrong in my command
gap> # so we seek help
gap> # here is how we ask for help on the topic of Factoring
gap> # we can't have a comment after the ?  since GAP would
gap> # interpret that as part of the topic to be searched
gap> ?factor;
Help:  several sections match this topic, type ?2 to see topic
2.
[1] reference:factor groups
[2] reference:factor groups of polycyclic groups - modulo pcgs
[3] reference:factor groups of polycyclic groups in their own
representation
[4] reference:factorcosetoperation
[5] reference:factorization
[6] reference:factorcosetaction for fp groups
[7] reference:factorfreemonoidbyrelations
[8] reference:factorfreesemigroupbyrelations
[9] reference:factorgroup
[10] reference:factorgroupfpgroupbyrels
[11] reference:factorgroupnc
[12] reference:factorgroupnormalsubgroupclasses
[13] reference:factorgrouptom
[14] reference:factorial
[15] reference:factorization (2nd)
[16] reference:factors
[17] reference:factors of univariate polynomial
[18] reference:factorsint
[19] reference:factorssquarefree
gap> # we go after the most likely topic
gap> ?16;
```

Here **GAP** prints out a lot of information, which I don't want to edit into a form acceptable to Latex. Let's simply note the proper command:

```
> Factors( <r> )
```

8. Now back to our session.

```
gap> # O.K. the correct command is Factors;
gap> # by the way UPPER and lower case must be distinguished
gap> # in communicating with GAP
gap> Factors(123454321);
[ 41, 41, 271, 271 ]
gap> # this is actually a list of the prime factors;
gap> # let's check
gap> IsPrime(41);IsPrime(271); # we can have two commands on a
line
true
true
gap> 41*41*271*271;
123454321
gap> # is it obvious that 123454321 is a perfect square?
gap> quit;
```

9. Here is another learning session:

```
gap> # since computers are profoundly stupid, they cannot
gap> # make the sort of casual identifications and
gap> # relabellings that humans do.
gap> # Hence computer algebra syntax is necessarily quite fussy
gap> # Gap also has to worry about allocating its computer
gap> # storage in light of all this
gap> #
gap> # ctrl - D should quit Gap normally
gap> # ctrl - C twice should stop it in the middle of an unfortunate computation
gap> #  whitespace = blanks, tabs, returns are generally meaningless
gap> # to Gap, so are useful for clear writing
gap> 2^3; 2 ^ 3;
8
8
gap> # in the next line I hit return BEFORE the ;
gap> 2
> ^
> 3
> ;
8
gap> # Gap must distinguish names of things in your
gap> # Gap session, e.g. variables, from things
gap> # elsewhere on your computer. E.g. file names
gap> # which, after all, are outside your session, must
gap> # be enclosed in quotes;
gap> #
gap> # For example, sometimes we want to read Gap input from a pre-prepared
gap> # file on our system. Maybe the data is very long or needs easy editing.
gap> # to do this, type and enter  Read("thefilename");
gap> # Or to record your calculations for posterity,
gap> # LogTo("anotherfilename");
gap> #
gap> #
gap> # you can do some editing within your session
gap> # E.g. ctrl-P recovers the last line of input
gap> # ctrl-E moves cursor to the end of line, ctrl - A to beginning
gap> # ctrl-K erases to end of line, etc.
gap> # Try these! See help section on ‘line editing’
gap> # Gap knows various kinds of constants
gap> # Integers:
gap> 134; -18; 2*6; 2^6; (-3)^5;
134
-18
```

```
12
64
-243
gap> Factorial(7);
5040
gap> # it knows truth values = Boolean constants
gap> 2^3 = 3^2; 3^3 = (32 - 5);
false
true
gap> # so you are not allowed to let ‘false’ or ‘true’
gap> # refer to other things;
gap> #
gap> # let’s assign to the variable prue the value 19!
gap> # this is so big I don’t want Gap to print the
gap> # full response, so I suppress output using   ;;
gap> prue := Factorial(19);;
gap> # O.K. let’s see how big that really was;
gap> prue;
121645100408832000
gap> #
gap> # ‘false’ , ‘true’ , ‘quit’, and ‘last’ and other keywords cannot
gap> # be used as variable names
gap> # now try something illegal:
gap> true := Factorial(18);
Syntax error: ; expected
true := Factorial(18);
      ^
6402373705728000
gap> # Gap did what it could - ignore the assignment,
gap> # but compute the factorial
gap> #
gap> true=true;
true
gap> true = false;
false
gap> # let’s abbreviate. We use   :=   with, no space to assign
gap> # some object to an identifier. For example, we assign the
gap> # reserved constant ‘true’ to the identifier ‘T’; ditto for ‘false’
gap> T := true; F:= false;
true
false
gap> (T or F) and (not T);
false
gap> # Gap knows permutations in cycle form;
gap> # E.g. the permutation taking 3->4->1 and back,
gap> # as well as 2->5 and back is
```

```
gap> perm:=(3,4,1)(2,5);
(1,3,4)(2,5)
gap> # of course we can multiply permutations
gap> # and find inverses
gap> perm^3;perm^2;perm^6;
(2,5)
(1,4,3)
()
gap> # so the period is 6 - note the identity ()
gap> quip:=(5,1,4,2);
(1,4,2,5)
gap> quip*perm; perm*quip*perm^-1; # a product and a conjugate
(3,4,5)
(2,4,3,5)
gap> # you can compute the action on an element
gap> 1^quip;
4
gap> # Gap understands elements of finite fields, complex roots of unity
gap> w:=E(3); # this is a primitive third root of unity
E(3)
gap> # namely exp(2*Pi/3);
gap> w^3;
1
gap> w^2+w+1;
0
gap> # However, Gap does not support floating point arrithmetic, i.e.
gap> # real numbers as a computer grasps them; nor does
gap> # Gap do graphics
gap> #
gap> # Gap will do all sorts of lists, set theory, matrices,etc.
gap> # Gap also needs a way to understand characters, namely
gap> # individual keyboard symbols whose mathematical content is
gap> # suppressed or non-existent
gap> 'a'; 'b'='a';
'a'
false
gap> # single characters only; if we want several we need lists

gap> # a variable name, or identifier, is any string of letters
gap> # or numbers, with at least one letter; keywords like `true' or
gap> # `quit' arenot allowed to be variable names.
gap> # Case is important:  x1 and X1 are different
gap> #
gap> # to assign some Gap meaning or object to a variable we enter
gap> #    varname := meaning   , for example
gap> a:=(1,2,3); # a permutation
```

```
(1,2,3)
gap> b:=(1,3,2)^2; # the ‘same’ permutation?
(1,2,3)
gap> a=b;
true
gap> IsIdenticalObj(a,b);
false
gap> # Hmmm: Gap recognizes that a and b are the ‘same’ permutation;
gap> # However, a and b are not identical objects, in the sense that
gap> # the two assignments for a and b pointed to different parts of memory.
gap> # You see, Gap couldn’t know ahead of time that a would equal b, so the
gap> # only reasonable response is to put the two objects in
gap> # different parts of memory
gap> c:=b;
(1,2,3)
gap> IsIdenticalObj(c,b); IsIdenticalObj(c,a);
true
false
gap> # Now, however, Gap has been programmed to understand that c and b
gap> # merely point to the same place in memeory - so c and be really are
gap> # identical from the machine’s strong point of view.
gap> #
gap> # this sort of fussiness is necessary, particularly when ‘=’
gap> # refers to some sort of equivalence relation.
gap> # What happens then in      varname := meaning
gap> # is that the identifier ‘varname’ is attached to, i.e.
gap> # points to the object ‘meaning’. This object may move about in
gap> # memory, but ‘varname’ will go with it, and will always
gap> # point to it --- unless we reassign a new meaning;
gap> a;b;a=b;
(1,2,3)
(1,2,3)
true
gap> a:=193;
193
gap> a; a=b;
193
false

gap> # Lists are  key structures: e.g. matrices are special lists
gap> Pr:=[2,3,5,7]; # a list of the first four primes
[ 2, 3, 5, 7 ]
gap> # To extend our list we ‘append’ another list
gap> Append(Pr,[11,13]);
gap> Pr; Length(Pr);
[ 2, 3, 5, 7, 11, 13 ]
```

```
6
gap> # note how Pr was automatically extended
gap> # say we know that the 8th prime is 19
gap> Pr[8]:=19;
19
gap> Pr;
[ 2, 3, 5, 7, 11, 13,, 19 ]
gap> # notice that Gap left space for the missing 7th element
gap> # such a list with holes is NOT DENSE
gap> Pr[7]:=17;
17
gap> Pr;# this is now a dense list
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> # there are many useful manipulations;
gap> Reversed(Pr);
[ 19, 17, 13, 11, 7, 5, 3, 2 ]
gap> Sum(Pr);Product(Pr);
77
9699690
gap> # a set is a dense list without repetitions;
gap> # this corresponds to how we normally think of sets;
gap> t:=[-3,5,4,5,3,-3,2,5];
[ -3, 5, 4, 5, 3, -3, 2, 5 ]
gap> T:=Set(t);
[ -3, 2, 3, 4, 5 ]
gap> # the elements are ordered in some fashion in a set
gap> # sets and lists can be empty
gap> e:=[]; # the empty list
[  ]
gap> Em:=Set(e); # the empty set - essentially the same thing
[  ]
gap> # The letter 'E' is reserved for use in finite fields
gap> #
gap> S:=Set([2,7,9]);
[ 2, 7, 9 ]
gap> Union(T,S);Intersection(T,S);Union(S,Em);
[ -3, 2, 3, 4, 5, 7, 9 ]
[ 2 ]
[ 2, 7, 9 ]
gap> W:=Set([(1,2,3),-17/4,[],S]);
[ -17/4, (1,2,3), [  ], [ 2, 7, 9 ] ]
gap> # sets and lists can contain all sorts of objects
gap> W;
[ -17/4, (1,2,3), [  ], [ 2, 7, 9 ] ]
gap> W[3];
[  ]
```

```
gap> W[4][3];
9
gap> # a range is an arithmetic progession
gap> R1:=[25,22..-5];
[ 25, 22 .. -5 ]
gap> Elements(R1);
[ -5, -2, 1, 4, 7, 10, 13, 16, 19, 22, 25 ]
gap> J:=[1..20];Elements(J);
[ 1 .. 20 ]
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ]
gap> # we can create lists in many ways
gap> List(J,IsPrime);
[ false, true, true, false, true, false, true, false, false, false, true,
  false, true, false, false, false, true, false, true, false ]
gap> # suppose we want to extract only the primes from [1..20]
gap> Filtered(J,IsPrime);
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> # a matrix is a list of lists of entires from some field or ring
gap> # here is a 2 by 2 matrix of rational numbers
gap> A:=[[3/5,-4/5],[4/5,3/5]];
[ [ 3/5, -4/5 ], [ 4/5, 3/5 ] ]
gap> Display(A);
[ [   3/5,  -4/5 ],
  [   4/5,   3/5 ] ]
gap> DefaultFieldOfMatrix(A);
Rationals
gap> Determinant(A);
1
gap> A^4; # its 4th power
[ [ -527/625, 336/625 ], [ -336/625, -527/625 ] ]
gap> Display(last);
[ [  -527/625,   336/625 ],
  [  -336/625,  -527/625 ] ]
gap> Display(A-A);
[ [  0,  0 ],
  [  0,  0 ] ]
gap> A^0;A^-1;
[ [ 1, 0 ], [ 0, 1 ] ]
[ [ 3/5, 4/5 ], [ -4/5, 3/5 ] ]
gap> Display(last);
[ [   3/5,   4/5 ],
  [  -4/5,   3/5 ] ]
gap> Trace(A);
6/5
gap> w:=E(5); # a primitive 5th root of unity
E(5)
```

```
gap> # lets put the 5th roots on the diagonal
gap> B:=DiagonalMat([1,w,w^2,w^3,w^4]);
[ [ 1, 0, 0, 0, 0 ], [ 0, E(5), 0, 0, 0 ], [ 0, 0, E(5)^2, 0, 0 ],
  [ 0, 0, 0, E(5)^3, 0 ], [ 0, 0, 0, 0, E(5)^4 ] ]
gap> Display(B);DefaultFieldOfMatrix(B);
[ [       1,       0,       0,       0,       0 ],
  [       0,    E(5),       0,       0,       0 ],
  [       0,       0,  E(5)^2,       0,       0 ],
  [       0,       0,       0,  E(5)^3,       0 ],
  [       0,       0,       0,       0,  E(5)^4 ] ]
CF(5)
gap> B^2;
[ [ 1, 0, 0, 0, 0 ], [ 0, E(5)^2, 0, 0, 0 ], [ 0, 0, E(5)^4, 0, 0 ],
  [ 0, 0, 0, E(5), 0 ], [ 0, 0, 0, 0, E(5)^3 ] ]
gap> Display(B^5);
[ [  1,  0,  0,  0,  0 ],
  [  0,  1,  0,  0,  0 ],
  [  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  1 ] ]
gap> Trace(B);
0
gap> # the 5th roots do sum to 0
gap> C:=PermutationMat((1,2,3,4,5),5);
[ [ 0, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 1 ],
  [ 1, 0, 0, 0, 0 ] ]
gap> Display(C);
[ [  0,  1,  0,  0,  0 ],
  [  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  1 ],
  [  1,  0,  0,  0,  0 ] ]
gap> # these two 5 by 5 matrices will generate a group
gap> Grp:=Group(B,C);;
gap> Order(Grp);
125
gap> IsAbelian(Grp);
false
gap> IsSimple(Grp);
false
gap> # finally, let's look at some
gap> # storage issues;
gap> Pr;
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Q:=Pr; # a copy of Pr;
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

```
gap> # let's change the 3rd entry of Q
gap> Q[3]:=5000;
5000
gap> Q;Pr;
[ 2, 3, 5000, 7, 11, 13, 17, 19 ]
[ 2, 3, 5000, 7, 11, 13, 17, 19 ]
gap> # what happened is that  Q:=Pr means that
gap> # Q merely points to the same Gap object as Pr, so
gap> # changing Q also changes Pr; let's restore Pr
gap> Pr[3]:=5;
5
gap> Pr;
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Q:=ShallowCopy(Pr);;
gap> Q;Pr;
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Q[3]:=5000;
5000
gap> Q;Pr;
[ 2, 3, 5000, 7, 11, 13, 17, 19 ]
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> # so now Q points to a new portion of memory.
gap> # this is an issue whenever variable point to objects which
gap> # can be changed
gap> x:=5;
5
gap> y:=x;
5
gap> x;y;
5
5
gap> x:=4;
4
gap> x;y;
4
5

gap> # now for some programming
gap> J:=[1..8]; # a typical index set
[ 1 .. 8 ]
gap> for j in J do Print(j^2); od; # note  the  ;  after the Print command
1491625364964gap>
gap> # well Gap is flexible, so we need formatting commands
gap> # the invisible character "\n" issues a new line command
gap>  for j in J do Print(j, j^2, "\n"); od;
```

```
11
24
39
416
525
636
749
864
gap> # try again
gap> for j in J do Print(j," ",j^2,"\n"); od; # note how the blank space " " appears
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
gap> perms:=[(1,3,2,6,8)(4,5,9), (1,6)(2,7,8), (1,5,7)(2,3,8,6),
> (1,8,9)(2,3,5,6,4), (1,9,8,6,3,4,7,2)];;
gap> Length(perms); # let's multiply these permutations
5
gap> prod:=(); # an empty product equals the identity
()
gap> for p in perms do
>  prod:=prod*p;
> od;
gap> prod; # the product of the permutations
(1,8,4,2,3,6,5,9)
gap> # here is the Sieve of Erathosthenes.
gap> # we'll find all primes less than 1500
gap> # notice that n mod p is the Gap command for the remainder
gap> # upon dividing n by p
gap> 15 mod 5; 15 mod 11; 15 mod 4; 15 mod 21;
0
4
3
15
gap> primes:=[]; # we must have an empty box in which we can put things!
[  ]
gap> numbers:=[2..1500];# this is the range of all integers from 2 to 1500
[ 2 .. 1500 ]
gap> #
gap> for p in numbers do
>  Add(primes,p);
>  for n in numbers do
```

```
>  if n mod p = 0 then
>  Unbind(numbers[n-1]);
>  fi;
>  od;
>  od;
gap> #
gap> Length(primes);
239
gap> # we found 239 primes; to see the last 10
gap> for j in [1..10] do Print(primes[229+j],"\n"); od;
1451
1453
1459
1471
1481
1483
1487
1489
1493
1499
gap> # Note: the function Unbind(numbers[n-1]) deletes
gap> # the element at position n-1 of the list ‘numbers’
gap> # and leaves a hole there
gap> J;
[ 1 .. 8 ]
gap> Elements(J);
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
gap> Unbind(J[5]);
gap> J;
[ 1, 2, 3, 4,, 6, 7, 8 ]
gap> #
gap> #
gap> # we can write our own functions
gap> # for example, we may simply want to abbreviate a wordy Gap command
gap> # say we want to abbreviate the
gap> # SmithNormalFormIntegerMat function
gap> # which produces elemntary divisors on the diagonal of a new matrix
gap> sm:=function(A)
> return SmithNormalFormIntegerMat(A);
> end;
function( A ) ... end
gap> B:=[[12,24,14],[-62,106,-8]];
[ [ 12, 24, 14 ], [ -62, 106, -8 ] ]
gap> Display(B);
[ [   12,   24,   14 ],
  [  -62,  106,   -8 ] ]
```

```
gap> sm(B);
[ [ 2, 0, 0 ], [ 0, 2, 0 ] ]
gap> Display(sm(B));
[ [  2,  0,  0 ],
  [  0,  2,  0 ] ]
gap> # or suppose we want a (useless!)function that replaces each rational
gap> # number by 1 if non-negative, 10 if negative;
gap> jump:=function(x)
>  if x < 0 then
>  return 10;
>  else
>  return 1;
>  fi;
>    end;
function( x ) ... end
gap> jump(19); jump(0); jump(-12/7);
1
1
10
gap> C:=ShallowCopy(B);;
gap> Display(C);
[ [   12,   24,   14 ],
  [  -62,  106,   -8 ] ]
gap> for j in [1..3] do
>  for i in [1..2] do
>  C[i][j]:=jump(C[i][j]);
>  od;
>    od;
gap> Display(C);
[ [   1,   1,   1 ],
  [  10,   1,  10 ] ]
```