# Mapping and modeling Earth Science Data

## Some brief notes on computing and programming

**Thorsten Becker**

University of Southern California, Los Angeles

Università di Roma TRE, June 2012

# Computing fundamentals

At a low level, a computer stores information in the binary system, *i.e.* in bits that can hold the values of either zero or one. You can then use a byte (8 bits) to encode numbers from 0 to $2^8 - 1 = 255$ using the binary system. For floating point or larger integers, more memory is required. A single precision float take up four bytes and is accurate up to $\sim 5 \cdot 10^{-7}$, a double precision float up to $\sim 5 \cdot 10^{-15}$. With a 32 bit operating system, the largest number you can represent is $2^{31} - 1 \sim 2.1$ billion. (Aside: this might seem like a big number but is not, as it corresponds roughly to a bit more then $800^3$ resolution.)

- → A numerical representation of a float will always be approximate (only integers are exact). This means to not test for $x == 0$ (equal to zero) but $abs(x) < \varepsilon \quad (abs(x) = |x|)$ where $\varepsilon$ depends on implementation.
- → The detailed storage depends on the hardware, "big endian" *vrs.* "small endian"
- → Some mathematical operations that are theoretically valid will lead to large round off errors. *e.g.* $\cos^{-1}(x)$ for small $x$, subtracting large numbers from each other.
- → The memory requirements for a float vector will be half of that of a double.

# Memory

1 MB (megabyte) corresponds $1024 \times 1024$ bytes; 1 GB = 1024 MB. As of 2008, your PC will have likely have at least $\sim$ 2 GB of Random Access Memory or RAM (as opposed to hard drive space) meaning you can store how many floats and doubles? To increase the available memory, one can use formerly called "supercomputers". Those consist these days mainly of

**Distributed memory machines** *e.g.* $200 \times 2 \times$quadcore (8 Central Processing Units or CPUs) $\times 8$ GB RAM machines which need specially designed software to make use of parallelism, *e.g.* Message Passing Interface or MPI.

**Shared memory machines** This is the more expensive, old school approach where several CPUs can share a larger than normal (*e.g.* 256 GB) memory. Compilers can sometimes help make your code make use of "parallelism", *i.e.* having the computational time decrease by using more than one core or CPU. Right now, typical PCs can be considered shared memory (multi-core, *i.e.* CPU) machines.

# Elements of a computer program

```
% This is the main program. Notice the '%' symbol - it means this line is
% a comment and will be ignored at run time.
i = 0;    % assign integer variable for loop
n = 100;    % some number of elements
x = zeros(n,1);   % allocate and initialize a vector x[] with n elements
y = 1;
for i = 1:n     % loop from i = 1, 2, ..., n
  x(i) = y^2;     % assign some value
  y = y+2;      % increment variable
end       % close loop
% notice the statements inside the loop are indented.
i = 1;
while (i <= n)      % different loop construct
  x(i) = mysin(x(i));     % function call
  i = i+1;
  printf("%g\n", x(i));     % output statement
end

% This is the subroutine or function 'mysin'
function result = mysin(xloc)
  result = sin(xloc);
% Note that this subroutine will not know the main programs
% variables,  they are "local".
```

# Programming philosophy

1. *Modularize* and test for robustness.

   - Break the task down into small into small pieces that can be reused within the same program or in another program
   - Test each part well before using it in a larger project to make code more robust.
   - ensure that each subroutine gives error messages, in case non sensible input arguments are given.
   - do not ignore compiler warnings

# Programming philosophy

2. Strive for *portability*

Don't use special tricks/packages that might not be available on other platforms.

3. *Comment*

   - Add explanatory notes for each major step, strive for a fraction of comments to code $\geq 30\%$

   This will help re-usability, should you or someone else want to modify the code later.

4. Use "*structures*", avoid globals

   - If variables are needed in several subroutines, do not use "global" declaration, but pass a structure that contains a set of variables.

5. *Avoid unnecessary computations*

   See below for common speed up tricks.

# Programming philosophy

6. *Visualize* you intermediate results often (But don't print it all out in color!)

   Bugs in the code can often be seen easily when output is analyzed graphically, and may show up as, *e.g.*

   - lines being wiggly when they should be smooth
   - solutions being skewed when they should be symmetrical
   - etc.

Object oriented programming forces you to follow rules 1 & 4 (not so much 2). Editors and advanced development environments (such as the Matlab DE) help with 3 & 6.

# *Programming:* Philosophy

- strive for **transparency** by commenting and documenting the hell out of your code
- create **modularity** by breaking tasks into reusable, general, and flexible bits and pieces
- achieve **robustness** by testing often
- obtain **efficiency** by avoiding unnecessary computational operations (instructions)
- maintain **portability** by adhering to standards

D. Knuth: The art of computer programming

# *Programming:* Traditional languages in the natural sciences

- *Fortran:* higher level, good for math
  - F77: legacy, *don't use* (but know how to read)
  - F90/F95: nice vector features, finally implements C capabilities (structures, memory allocation)
- *C:* low level (*e.g.* pointers), better structured
  - very close to UNIX philosophy
  - structures offer nice way of modular programming, see Wikipedia on C
- I recommend F95, and use C happily myself

# *Programming:* Some Languages that haven't completely made it to scientific computing

- *C++*: object oriented programming model
  - reusable objects with methods and such
  - can be partly realized by modular programming in C
- *Java*: what's good for commercial projects (or smart, or elegant) doesn't have to be good for scientific computing
- Concern about portability as well as general access

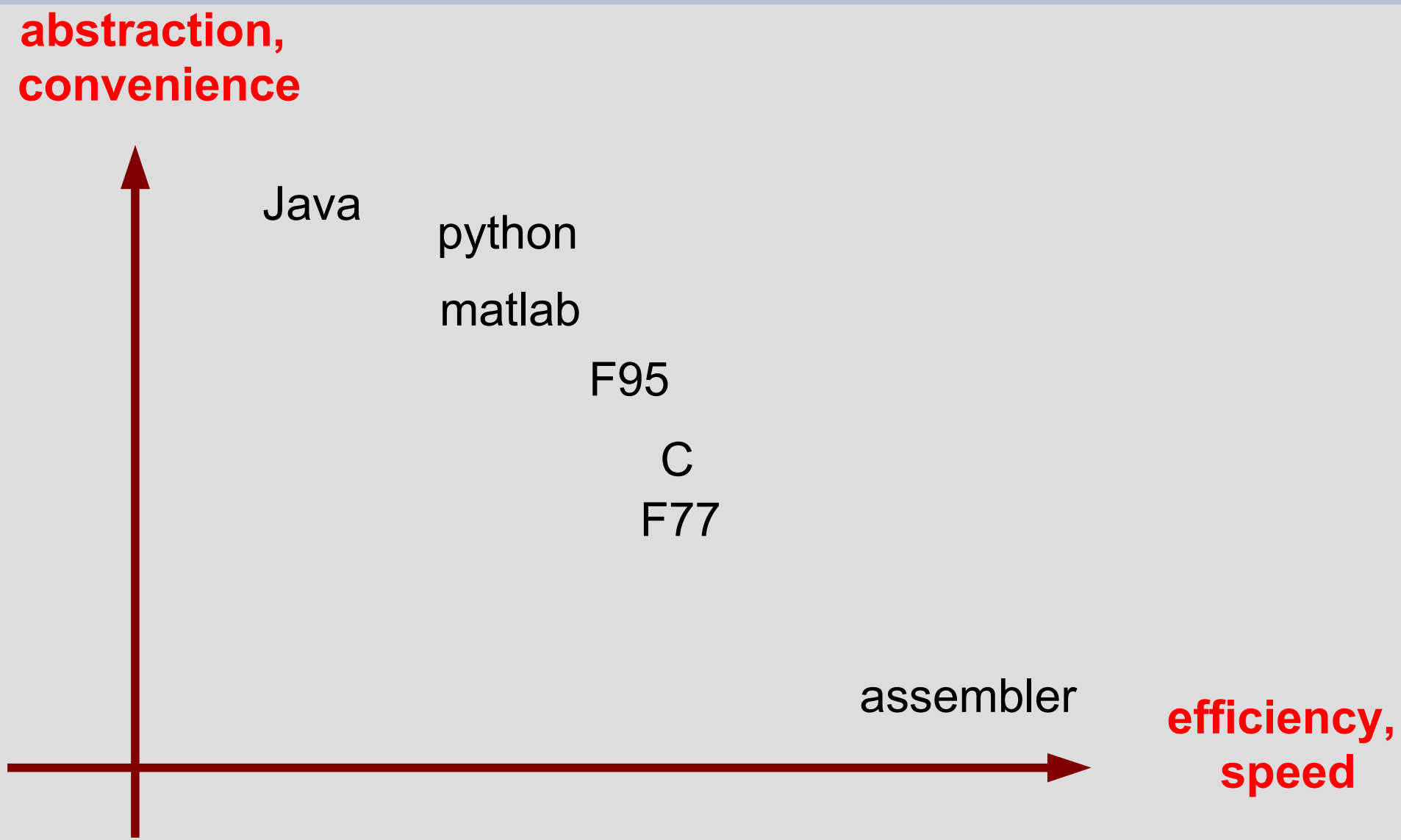
# *Programming:* **Compromises**

- Python
  - Object oriented
  - Interpreted
  - Interfaces easily with F90/C
  - Numerous scientific packages

# *Programming:* Other interpreted, high-abstraction languages for scientists

- Bash, awk, perl, python: script languages
  - Bash: A shell can be scripted
  - AWK: interpreted C, good for simple data processing
- Matlab (or octave)
  - language is some mix between F77 and C, highly vectorized
  - might be good enough for your tasks
  - interpreted (slower), but can be compiled
- IDL: visualization mostly, like matlab
- Mathematica: symbolic math mostly

# *Programming:*
# Trade-offs for scientific computing



# *Programming:* Most languages need to be compiled to assembler language

- $(F77) $(FFLAGS) -c main.f -o main.o
- there are standards, but implementations differ, especially for F77/F90/F95, not so much for C
- the compiler optimization flags, and the choice of compiler, can change the execution time of your code by factors of up to ~10, check out some example benchmarks for F90
- the time you save might be your own
- don't expect -O5 to work all the time

# AWK programming

- In many ways, like C
- Very useful for small computations, in particular when operating on ASCII tables
- Based on line by line processing

```
echo 5 6 7 | gawk '{print($2*$3)}'
```

will return 42

```
#
# calculate the standard deviation of the col column,
# fast (and inaccurate) if fast is set to unity, default is slow
# if col is not set, uses col=1
#
# $Id: standarddev.awk,v 1.5 2012/06/23 22:20:39 becker Exp becker $
BEGIN {
    sum = sum2 = 0.0;n = i = 0; # initialize summations
    if(col==0)                  # default is to use first column
        col = 1;
}
{
    if((NF>=col)&&(substr($1,1,1)!="#")&&(tolower($col)!="nan")){ # we can use this line
        if(fast){                       # fast, inaccurate way
            sum += $col;sum2 += $col * $col;n++;
        }else{                  # slow way
            n++;x[n]=$col;sum += $col;
        }
    }
}
END {
    if(n > 1){
        if(fast){
            std = sqrt ((n * sum2 - sum * sum) / (n*(n-1)));
            # mean would be  sum / n;
            printf("%.10g\n",std)
        }else{
            mean = sum / n;sum2 = 0.0;
            for(i=1;i<=n;i++){
                x[i] -= mean;
                sum2 += x[i]*x[i];
            }
            printf("%.10g\n",sqrt(sum2/(n-1)));
        }
    }else{
        print("NaN");
    }
}
```

# Additional material

# *Tuning:*
# How to design efficient code

Even better, in Matlab (and languages such as FORTRAN90) you can *vectorize*, *i.e.* write symbolically for a vector **x**

```
x = x + 5; % x here can be a matrix or a vector
```

if you want to add a scalar to each element, or

```
x = x .* y
```

for the example above. Matlab internally takes take that the looping is taking care of in the most efficient way. This can make a huge difference, vectorize whenever you can in.

3. *Avoid if statements* as much as possible. For example, if this test

```
if(debug == 1) % evaluating this expression will take time
  % do this
else
  % do that
end
```

if optional and usually zero, comment it out using pre-processor directives. *I.e.* in C, you would write the code like so

```
#ifdef DEBUG
    % code here for debugging version of program
#else
    % code here for the regular version of program
#end
```

and compile the program with or without

```
gcc -DDEBUG
```

depending on if you want those statements to be executed when the program runs.

4. *Pre-compute* common factors to avoid redundant computations.
   For example, instead of

```
for i = 1:n
  x(i) = x(i)/180*pi;
end
```

   It is better to do

```
fac = pi/180;
for i = 1:n
  x(i) = x(i)*fac;
end
```

   because it entails one less division per step. In Matlab, it's better still to use the vectorized version, `x=x.*fac`.

5. *Share the code*!

   The more eyes, the less bugs, and the better the performance.

# *Tuning:*
# How to design efficient code

6. Use *hardware optimized packages* for standard tasks, *e.g.*

   - LAPACK for linear algebra
     This package is available highly optimized for several architectures.
   - FFTW for FFT,
     an automatically adapting package.

   Different hardware makes certain chunks of memory sized ( "cache" ) operations highly efficient (see, *e.g.* *Dabrowski et al.*, 2008, as used later in class).

7. *Use version control*!

   Use version control packages (such as subversion, RCS) during code development, as this might safe you an immense amount of time when you're trying to track down where and when that bug crept into the code.

# *Tuning:*
# How to design efficient code

1. *Avoid reading and writing* intermediate steps *to "file"*, *i.e.* on the hard drive (Input/Output or IO) if at all possible.

2. Use nested loops that are sorted by the fastest/major index, because memory access is faster that way. The storage depends on the computer language (C *vrs.* FORTRAN). *e.g.* in Matlab, you would write

```
for i = 1:n      % increment i across all rows (slow index)
  for j = 1:m      % row i computations across all columns j first
    x(i,j) = x(i,j) * y(i,j);
  end
end
```

to multiply x elements by those of y and NOT the other way around,

```
for j = 1:m      % row i computations across all columns j first
  for i = 1:n      % this will make things jump around in memort
                   % and slow things down
   x(i,j) = x(i,j) * y(i,j);
 end
end
```

# *Tuning:*
# How to design efficient code

- watch array ordering in loops x[i*n+j]
- avoid things like:
  - f1=cos(x)*sin(y);f2=cos(x)*cos(y);
- use BLAS, LAPACK
- experiment with compiler options which can turn *good* (readable) code into *efficient*
- look into hardware optimized packages
- design everything such that you can run in parallel ($0^{th}$ order) on one file system

# *Tuning:* Tuning programs

- use compiler options, *e.g.* (AMD64 in brackets)
  - GCC: `-O3 -march=pentium4 (-m64 -m3dnow -march=k8) -funroll-loops -ffast-math -fomit-frame-pointer`
  - ifc: `-O3 -fpp -unroll -vec_report0`
  - PGI: `-fast -Mipa (-tp=amd64)`
- use profilers to find bottlenecks
- use computational libraries

# *Tuning:* **Matlab**

- basically all actual computations within matlab are based on LAPACK, ARPACK and other library routines etc.
- Mathwork added a bunch of convenient wrappers and neat ways to do things vectorially
- plus plotting and GUI creation
- Matlab *might* be all you need

# *Tuning:* Using standard subroutines and libraries

- many tasks and problems have been encountered by computational scientists for the last 50 years
- don't re-invent the wheel
- don't use black boxes either
- exceptions: I/O, visualization, and complex matrix operations

# *Algorithms:* Collection of subroutines: Numerical recipes

- THE collection of standard solutions in source code
- usually works, might not be best, but OK
- NR website has the PDFs, I recommend to buy the book
- criticism of numerical recipes
- website on alternatives to numerical recipes
- use packages like LAPACK, if possible
- else, use NR libraries but check
- MATLAB wraps NumRec, LAPACK & Co.

# *Tuning:* Linear algebra packages, examples

- EISPACK, ARPACK: eigensystem routines
- BLAS: basic linear algebra (*e.g.* matrix multiplication)
- LAPACK: linear algebra routines (*e.g.* SVD, LU solvers), SCALAPACK (parallel)
- parallel solvers:  MUMPS (parallel, sparse, direct), SuperLU
- PETSc: parallel PDE solvers, a whole different level of complexity

# *Tuning:* Examples for some other science (meta-)packages

- netlib repository
- GNU scientific library
- GAMS math/science software repository
- FFTW: fast fourier transforms
- **hardware optimized solutions**
  - ATLAS: automatically optimized BLAS (LAPACK)
  - GOTO: BLAS for Intel/AMD under LINUX
  - Intel MKL: vendor collection for Pentium/Itanium
  - ACML: AMD core math library

# *Shells:* Cluster job control

- on large, parallel machines one typically runs batch schedulers or queing systems
- this allows distributing jobs and utilizing resources efficiently
- PBS
  - **qsub** myjob.exe -tricky_options -q large
  - **qstat** | grep $USER
  - **pbstop**
  - **qdel** job-ID

# Writing and compiling a C program

**Example project:** main.c

```
/*
example program to compute the sin and cosine
of x values in degrees read in from stdin

Thorsten Becker, July 2005, twb@usc.edu

$Id: main.c,v 1.2 2005/07/30 19:57:34 becker Exp $

*/
#include "mysincos.h"

int main(int argc, char **argv)
{
  COMP_PRECISION x;
  int n;
  if(argc != 1){
    /*
       for all non-zero number of command line arguments,
       print a help page
    */
    fprintf(stderr,"%s\ncompute sin and cos from x values [deg] read from stdin\n",
            argv[0]);
    exit(-1);
  }
  fprintf(stderr,"%s: reading x values in degrees from stdin\n",
          argv[0]);
  n = 0;
  while(fscanf(stdin,SCAN_FMT,&x) == 1){
    /*
       compute and print values
    */
    fprintf(stdout,"%11g %11g\n",
            mysinf_deg(x),mycosf_deg(x));
    n++;                              /* increment counter */
  } /* end while loop */
  fprintf(stderr,"%s: computed %i pairs of sines/cosines\n",
          argv[0],n);

  return 0;                           /* normal end */
}
```

# Example project: mysincos.h

```
/*
   definitions and headers for example program to compute the sin and
   cosine of numbers read in from stdin

   Thorsten Becker, July 2005

   $Id: mysincos.h,v 1.1 2005/07/30 19:44:18 becker Exp becker $

*/

#include <stdio.h>
#include <math.h>

/*
   precision setting
*/
#define DOUBLE_PRECISION        /* use double precision?
                                   undefine else
                                */

#ifdef DOUBLE_PRECISION
#define COMP_PRECISION double
#define SCAN_FMT "%lf"
#else  /* single precision */
#define COMP_PRECISION float
#define SCAN_FMT "%f"
#endif

/* constants */
#define ONEEIGHTOVERPI 57.295779513082320876798154814105


/*
   function and subroutine declarations
*/
COMP_PRECISION mysinf_deg(COMP_PRECISION);
COMP_PRECISION mycosf_deg(COMP_PRECISION);
```

# Example project:
myfunctions.c

```
/*

   functions and subroutines for example program to compute the sin and
   cosine of numbers read in from stdin

   Thorsten Becker, July 2005, twb@usc.edu

   $Id: myfunctions.c,v 1.1 2005/07/30 19:44:15 becker Exp becker $

*/
#include "mysincos.h"

/*
   compute the sin of a value in degres
   input: xdeg [deg]

   output: return value

*/
COMP_PRECISION mysinf_deg(COMP_PRECISION xdeg)
{
  COMP_PRECISION xrad;
  xrad = xdeg/ONEEIGHTOVERPI;    /* convert to radians */
  return sin(xrad);              /* rerturn sin */
}
/* compute the cos of a value in degres */
COMP_PRECISION mycosf_deg(COMP_PRECISION xdeg)
{
  COMP_PRECISION xrad;
  xrad = xdeg/ONEEIGHTOVERPI;
  return cos(xrad);
}
```

# *Building:* How to compile the example project

```
becker@jackie:~/dokumente/teaching/unix/example >
ls
bin/  main.c  makefile  myfunctions.c  mysincos.h  objects/  RCS/

becker@jackie:~/dokumente/teaching/unix/example >
cc main.c -c

becker@jackie:~/dokumente/teaching/unix/example >
cc -c myfunctions.c

becker@jackie:~/dokumente/teaching/unix/example >
cc main.o myfunctions.o -o mysincos -lm

becker@jackie:~/dokumente/teaching/unix/example >
echo 0 90 -90 | mysincos
mysincos: reading x values in degrees from stdin
         0          1
         1 6.12303e-17
        -1 6.12303e-17
mysincos: computed 3 pairs of sines/cosines
```

# Building:

**Automating the build process with make:**
makefile

```
#
# makefile for mysincos package
#
# $Id: makefile,v 1.1 2005/07/30 20:06:07 becker Exp becker $
#
# architecture
ARCH = i686
# object file directory
ODIR = objects/$(ARCH)/
# binary directory
BDIR = bin/$(ARCH)/
# header files
HDR = mysincos.h
# needed objects
OBJS = $(ODIR)/main.o $(ODIR)/myfunctions.o
# libraries
LIBS = -lm
# main targets
all: $(OBJS) mysincos
# clean up
clean:
        $(RM) $(ODIR)/*.o
# binary
mysincos: $(BDIR)/mysincos
# main program rule
$(BDIR)/mysincos: $(OBJS)
        $(CC) $(OBJS) -o $(BDIR)/mysincos $(LIBS)
# make directories
dirs:
        if [ ! -s ./objects/ ]; then mkdir objects;fi;
        if [ ! -s $(ODIR) ];then mkdir $(ODIR);fi;\
        if [ ! -s ./bin/ ];then mkdir bin;fi;\
        if [ ! -s bin/$(ARCH)/ ];then mkdir bin/$(ARCH);fi;
#
# general rules to generate objects
#
$(ODIR)/%.o: %.c  $(HDR)
        $(CC) $(CFLAGS) $(INCLUDES) -c $< -o $(ODIR)/$*.o
```

# *Building:* Building with make

```
becker@jackie:~/dokumente/teaching/unix/example > make dirs
if [ ! -s ./objects/ ]; then mkdir objects;fi;
if [ ! -s objects/i686/ ];then mkdir objects/i686/;fi;\
if [ ! -s ./bin/ ];then mkdir bin;fi;\
if [ ! -s bin/i686/ ];then mkdir bin/i686;fi;
becker@jackie:~/dokumente/teaching/unix/example > make
icc -no-gcc -O3  -unroll -vec_report0 -DLINUX_SUBROUTINE_CONVENTION  -c main.c \
-o objects/i686//main.o
icc -no-gcc -O3  -unroll -vec_report0 -DLINUX_SUBROUTINE_CONVENTION  -c myfunctions.c \
-o objects/i686//myfunctions.o
icc objects/i686//main.o objects/i686//myfunctions.o -o bin/i686//mysincos -lm
becker@jackie:~/dokumente/teaching/unix/example > make
make: Nothing to be done for `all'.
becker@jackie:~/dokumente/teaching/unix/example > touch main.c
becker@jackie:~/dokumente/teaching/unix/example > make
icc -no-gcc -O3  -unroll -vec_report0 -DLINUX_SUBROUTINE_CONVENTION  -c main.c \
-o objects/i686//main.o
icc objects/i686//main.o objects/i686//myfunctions.o -o bin/i686//mysincos -lm
```

# *Building:* Version control

- **RCS**, **SCCS**, **CVS**, **SVN**: tools to keep track of changes in any documents, such as source code or HTML pages
- different versions of a document are checked in and out and can be retrieved by date, version number etc.
- I recommend using RCS for everything

# *Building:* RCS example

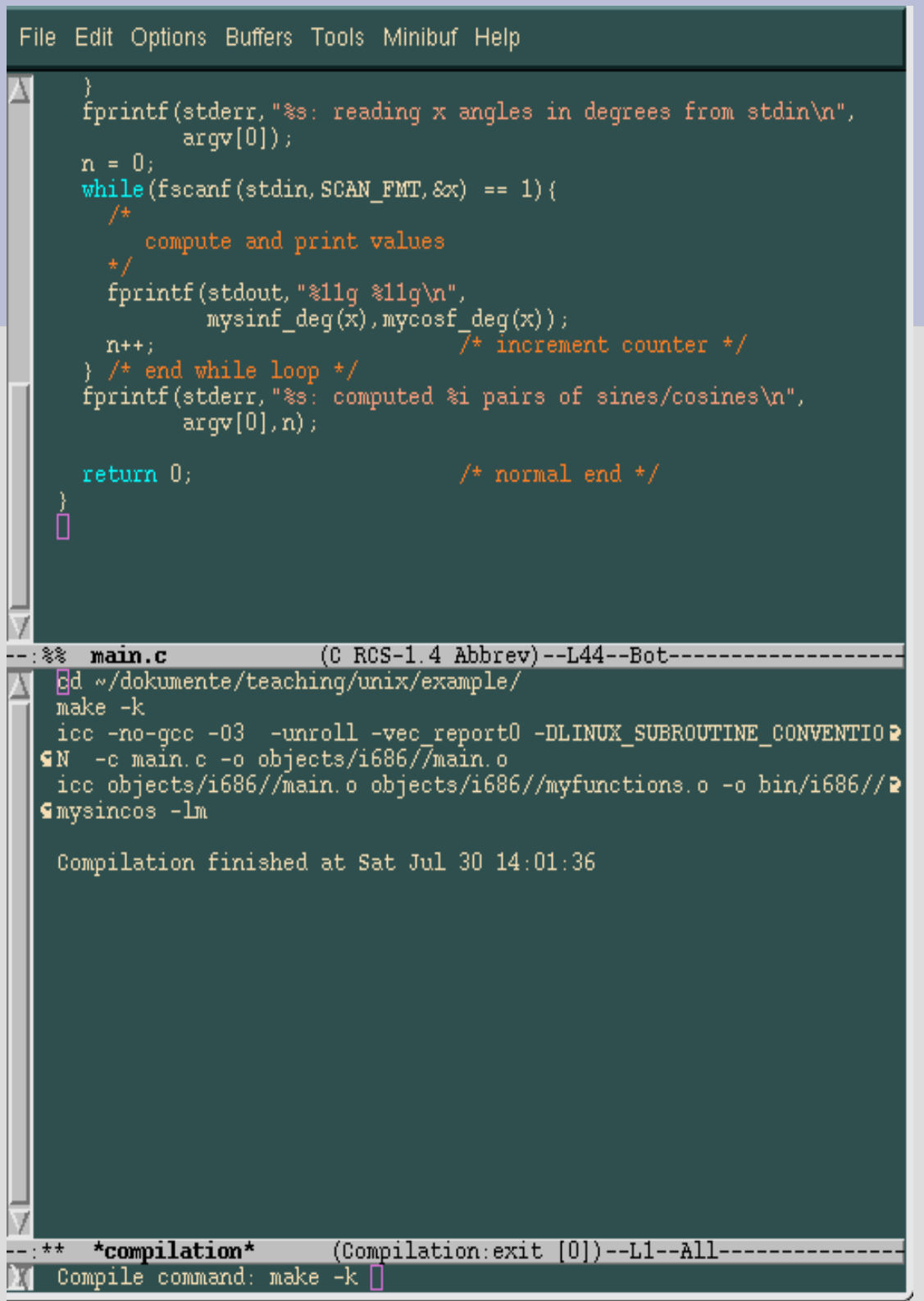
```
becker@jackie:~/dokumente/teaching/unix/example > co -l main.c
RCS/main.c,v  -->  main.c
revision 1.2 (locked)
done

becker@jackie:~/dokumente/teaching/unix/example > emacs main.c

becker@jackie:~/dokumente/teaching/unix/example > ci -u main.c
RCS/main.c,v  <--  main.c
new revision: 1.3; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> corrected some typos
>>
done

becker@jackie:~/dokumente/teaching/unix/example > rcsdiff -r1.2 main.c
===================================================================
RCS file: RCS/main.c,v
retrieving revision 1.2
diff -r1.2 main.c
9c9
< $Id: main.c,v 1.2 2005/07/30 19:57:34 becker Exp $
---
> $Id: main.c,v 1.3 2005/07/30 20:42:07 becker Exp $
27c27
<   fprintf(stderr,"%s: reading x values in degrees from stdin\n",
---
>   fprintf(stderr,"%s: reading x angles in degrees from stdin\n",
```

# *Building:*
# Version control is worth it

- small learning curve, big payoff
- EMACS can integrate version control, make, debugging etc. consistently and conveniently
- opening files, checking in/out can be done with a few keystrokes or menu options

# *Building:*
# EMACS modes

- EMACS is just one example of a programming environment
- *e.g.* there is a **vi** mode within EMACS
- dotfiles.com on .emacs

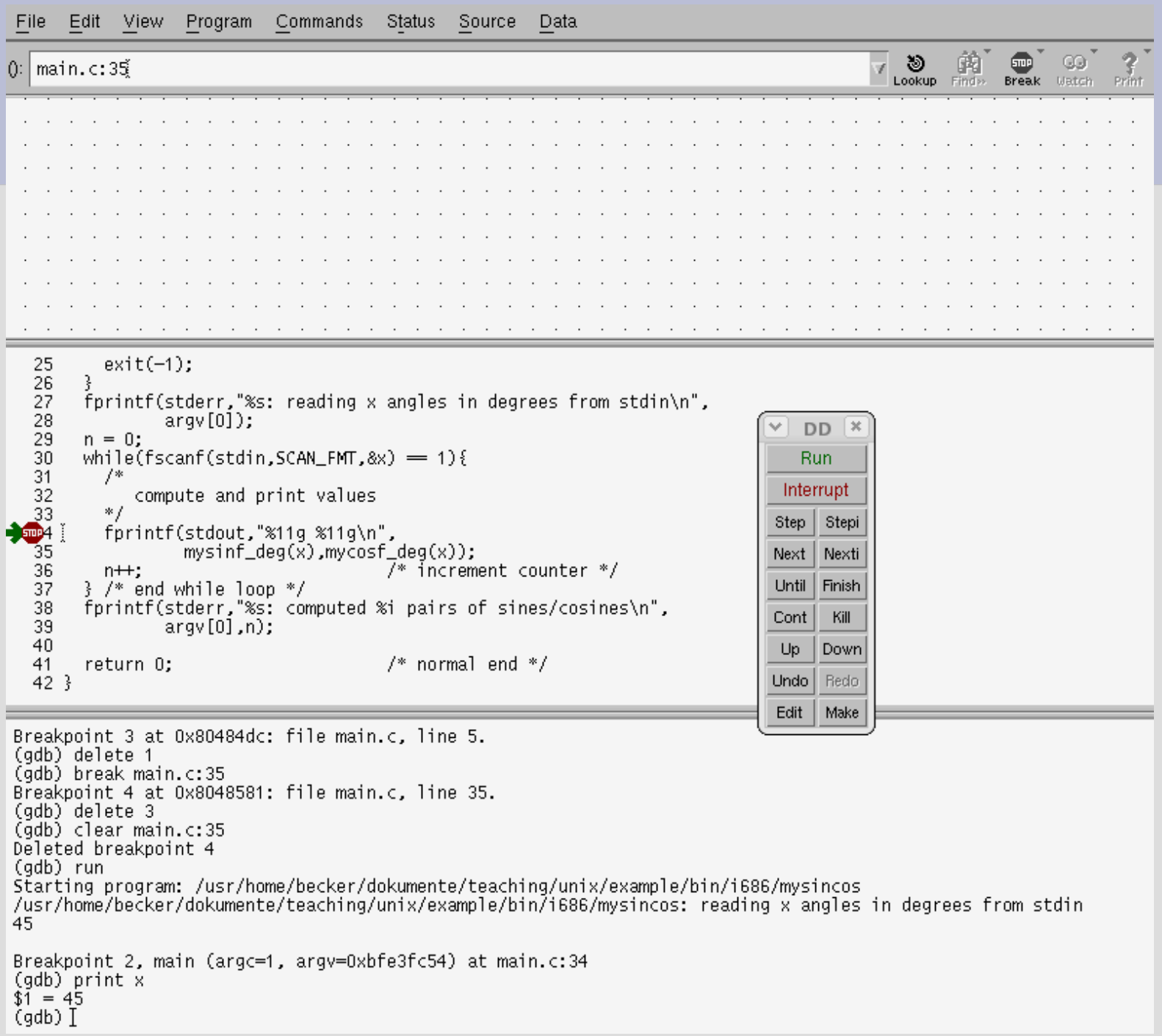


# *Building:* Debugging

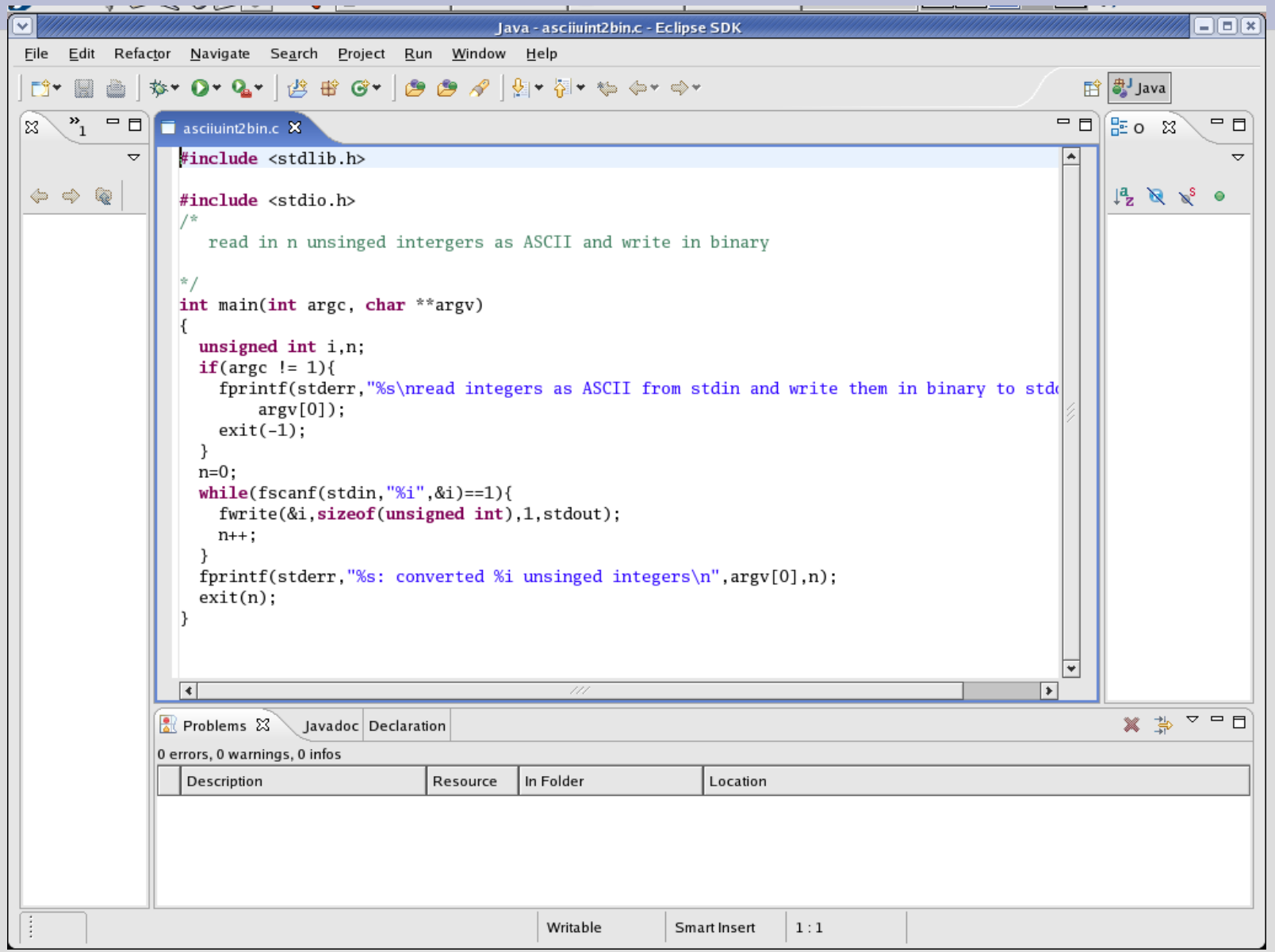
- put in extra output statements into code (still the only option for MPI code, kind of)
- use debuggers:
  - compile without optimization: **cc** -g main.c -c
  - **gdb**: command line debugger
    - **gbg** bin/x86_64/mysincos
    - (gdb) run
    - after crash, use *where* to find location in code that caused coredump, etc.
  - visual debuggers: ddd, photran, etc.

# Building:

# ddd



# Building: Eclipse environment (see also Code warrior etc.)



# *Tuning:* Calling F90 from C

- some subroutines are Fortran functions which you might want to call from C
- this works if you pointerize and flip
  - call func(x) real*8 x as func(&x) from C
  - storage of x(m,n) arrays in Fortran for x(i,j) is x[j*m+i] (fast rows) instead of x[i*n+j] (fast columns)
- C x[0,1,2,...,n-1] will be x(1,2,...,n) in Fortran
- don't pass strings (hardware dependent)
- BTW: Fortran direct binary I/O isn't really binary