

Do and Don'ts in coding

Stefano Borini – Advanced programming in Fortran

Do and Don'ts in coding

Best practices in coding

Programming lore to produce better code

“Better” means:

- More readable
- Less bug prone
- Easier to debug
- Easier to understand
- Easier to discuss
- More reusable
- More uniform
- More pleasant

It has no involvement in:

- Performance
- Overall design
- Algorithmic correctness

This course is aimed at Fortran 95
Concepts are applicable to any language

Focus is on implementation, not on design.

Do and Don'ts in coding

The mastery of a craftsman is in both his experience and his tools

Learn an advanced editor
Vim, emacs, any IDE, your choice
Learn it well

My personal priority list

Fundamental features

- syntax highlight
- keyboard access
- autocompletion
- autoindent

Important features

- folding
- abbreviations
- black-background friendly

Nice to have


- Compile/build/debug integration
- Automatic annotation of FIXMEs

Do and Don'ts in coding

Naming and communication

Do and Don'ts in coding

Name routines and variables with meaningful, expressive names

 `real :: p(:, :)`

Impossible to grep
Likely to require description

 `real :: densityMatrix(:, :)`

Greppable
Self-describing


Check spelling

Nobody wants to use `ClaculateEigenvalues()`

This is particularly important for routines that are part of a common library.
Changes will be hard after people and code depend on them.


Do and Don'ts in coding

Name routines and variables with meaningful, expressive names

 subroutine esp(p,dsto,funvec,ngpb,nder)

Forces to check the routine contents for clues

Forces brain to constantly map concept <-> symbol

 subroutine ElectrostaticPotential(densityMat, orbitalMatSize, potential, numGridPoints, derivativeOrder)

Clear and Self-documenting. Likely saves peeking into the routine.

Forces more natural mapping concept <-> symbol

The name of the routine must respect what it does

```
subroutine  
mulliken()
```


Expectation: computes the mulliken charges

Sad Reality: computes mulliken charges, total charge, dipole, norm of dipole,
stores this info in globals and writes everything into a file.

Do and Don'ts in coding

Named arguments

Use named arguments at call time to enrich meaning, if not self explanatory


 call NewDistributedMatrix(distrOverlap, distrHamiltonian)

 call NewDistributedMatrix(distrOverlap, template=distrHamiltonian)

Particularly important for those arguments that can be accidentally swapped without the compiler reporting it

```
subroutine SetSize(self, width, height)
  <irrelevant> :: self
  integer, intent(in) :: width
  integer, intent(in) :: height
end subroutine
```

 call SetSize(whatever, 100, 200)

 call SetSize(whatever, width=100, height=200)

Do and Don'ts in coding

Order of parameters

Be consistent with the ordering of parameters in a routine

```
subroutine MySub(input, output, optional)
```

Within each group:

- Choose an ordering that sounds natural
- Keep the same ordering for routines that use the same group of data, unless previous point is violated.

In “object oriented” methods, the first argument is always the object.
We use the relatively standardized name “self” as a standard object name.

```
subroutine Solve(self, runConfig, molecule, symmetry, result)
  type (MethodEvaluatorType), intent(inout) :: self
  type (MethodConfigType), intent(in)      :: runConfig
  type (MoleculeType), intent(in)         :: molecule
  type (SymmetryType), intent(in)          :: symmetry
  type (MethodResultType), intent(inout)   :: result
```


Do and Don'ts in coding

Keep it short. Don't go to excessive lengths

 `real :: firstOrderDensityMatrixDerivativeDiagonal(:)`

Accept a compromise:

As short as possible, while keeping clarity.
Four or more words is too much.

Context-clear abbreviations are generally OK (mat, vec, num).


Use other techniques to restrict context scope

No redundancy for data layout / physical type info!

 `real :: chargesVector(:)`


 `real :: charges(:)`


“vector” is relative to the fact that the data is a fortran vector. Redundant.

 `real :: densityMatrix(:, :)`

 `real :: densityMat(:, :)`

The entity is naturally called “density matrix”. In this case “Matrix” or “Mat” is OK.

 `integer :: iCounter`
`real :: raCharges(:)`

 `integer :: counter`
`real :: charges(:)`
`integer :: iAtom, jAtom`
`integer :: runType`
`character(len=128) :: runTypeStr`

Sold under the name of “Hungarian notation”. Generally redundant. In some cases it may be useful to express purpose or to convert data from one form to another.

Do and Don'ts in coding

Consistent naming

Be consistent in your naming usage throughout the program

```
subroutine Whatever()  
    real :: densityMatrix(:, :)  
end subroutine
```

```
subroutine WhateverElse()  
    real :: dMatrix(:, :)  
end subroutine
```

```
subroutine Whatever()  
    real :: densityMatrix(:, :)  
end subroutine
```

```
subroutine WhateverElse()  
    real :: densityMatrix(:, :)  
end subroutine
```

Don't reuse

Generally Bad


```
call GetOverlap(matrix)  
<...>  
call GetHamiltonian(matrix)
```


Worse!

```
call GetOverlap(overlap)  
<...>  
call GetHamiltonian(overlap)
```

Do and Don'ts in coding

Use temporary variables to your advantage to communicate intent

 if (atnum(iatom) == 1 .and. bondnum(iatom) == 0 .and. charge(iatom) > 0.0) then
 <code>
endif

 isLoneProton = (atnum(iatom) == 1 .and. bondnum(iatom) == 0 .and. charge(iatom) > 0.0)
if (isLoneProton) then
 <code>
endif

You may want to create a function. This is an example.

Do and Don'ts in coding

Select case

```
select case (atm)
  case ('C_2')
    <code>
  case ('O_3')
    <code>
  case ('S_2')
    <code>
end select
```

```
character(len=3), parameter :: ATOM_TYPE_C_2 = 'C_2'
character(len=3), parameter :: ATOM_TYPE_O_3 = 'O_3'
character(len=3), parameter :: ATOM_TYPE_S_2 = 'S_2'
```

```
select case (atomType)
  case (ATOM_TYPE_C_2)
    <code>
  case (ATOM_TYPE_O_3)
    <code>
  case (ATOM_TYPE_S_2)
    <code>
  case default
    continue
end select
```

Always provide a default action

In some cases, it can just do nothing (use **continue** for self-documenting)
In some other, it can bail out (reporting a potential bug in the program)

Always use parameters

Catches typos, makes intent clear when the code is a number

Do and Don'ts in coding

Select case

```
select case (cappingPotential)
  case default
    <code>
  case (1) ! ZLY1
    <code>
  case (2) ! ZLY2
    <code>
  case (3) ! DHC1
    <code>
end select
```

Default case should be the **last** one.
Intuition: *"if everything I tried fails, do this"*.

Don't use hardcoded values.
Use the compiler at your advantage to catch typos
and provide a mnemonic.

You can also use integers instead of character strings

```
select case (cappingPotential)
  case (1)
    <code>
  case (2)
    <code>
  case (3)
    <code>
  case default
    <code>
end select
```

```
integer, parameter :: CAPPING_TYPE_ZLY1 = 1
integer, parameter :: CAPPING_TYPE_ZLY2 = 2
integer, parameter :: CAPPING_TYPE_DHC1 = 3
```

! -----

```
select case (cappingPotential)
  case (CAPPING_TYPE_ZLY1)
    <code>
  case (CAPPING_TYPE_ZLY2)
    <code>
  case (CAPPING_TYPE_DHC1)
    <code>
  case default
    <code>
end select
```

Do and Don'ts in coding

Visuals

Do and Don'ts in coding

UPPER CASE vs lower case

ALL CAPS TEXT DECREASES EASE OF READING

All caps text decreases ease of reading

Why? Possible explanations are

- We read words “by shape” and up-downs. Shape is uniform with all caps.
- We read better what we read more often.
- CAPS transmit emphasis and SHOUTING!



Also not good for typing: Either you code with CAPS LOCK or you press shift all the time

Fortran traditionally overuses upper case in old code.

Stop following this tradition
Fix code that uses it

Sources:

- <http://psycnet.apa.org/journals/apl/30/2/161/>
- “Dynamics in Document Design” ISBN-13: 978-0471306368
- <http://www.fonts.com/AboutFonts/Articles/SituationalTypography/AllCaps.htm>

Do and Don'ts in coding

Indentation/syntax highlight

```
IF( LSAME( NORM, 'M' ) ) THEN
  VAL = ZERO
IF( LSAME( UPLO, 'U' ) ) THEN
  K = 0
  DO J = 1, N
    DO I = K + 1, K + J - 1
      VAL = MAX( VAL, ABS( AP( I ) ) )
    ENDDO
    K = K + J
    VAL = MAX( VAL, ABS( REAL( AP( K ) ) ) )
  ENDDO
ELSE
  K = 1
  DO 40 J = 1, N
    VAL = MAX( VAL, ABS( REAL( AP( K ) ) ) )
    DO 30 I = K + 1, K + N - J
      VAL = MAX( VAL, ABS( AP( I ) ) )
    ENDDO
    K = K + N - J + 1
  ENDDO
END IF
ENDIF
```

```
if (lSame( norm, 'M' )) then
  val = zero
  if (lSame( uplo, 'U' )) then
    k = 0
    do j = 1, n
      do i = k+1, k+j-1
        val = max(val, abs(ap(i)))
      enddo
      k = k+j
      val = max(val, abs(real(ap(k))))
    enddo
  else
    k = 1
    do j = 1, n
      val = max(val, abs(real(ap(k))))
      do i = k + 1, k + n - j
        val = max(val, abs(ap(i)))
      enddo
      k = k+n-j+1
    enddo
  endif
endif
```


```
if (lSame( norm, 'M' )) then
  val = zero
  if (lSame( uplo, 'U' )) then
    k = 0
    do j = 1, n
      do i = k+1, k+j-1
        val = max(val, abs(ap(i)))
      enddo
      k = k+j
      val = max(val, abs(real(ap(k))))
    enddo
  else
    k = 1
    do j = 1, n
      val = max(val, abs(real(ap(k))))
      do i = k+1, k+n-j
        val = max(val, abs(ap(i)))
      enddo
      k = k+n-j+1
    enddo
  endif
endif
```


Indentation communicates you the hierarchy of blocks execution

Syntax highlight allows you to spot mistyped entities
and visually classify language tokens vs. variables/routines

Do and Don'ts in coding

Use letter casing at your advantage

 integralmaxvalue
numindependentatoms

 integralMaxValue
numIndependentAtoms

*NB: differently from other languages, Fortran is case insensitive
Foo, fOo, FOO, FoO all refer to the same variable.*

Choose and respect a proper casing convention

but stay consistent with the established one, unless random

- CamelCase
for subroutines (our convention)
Other languages: class names

```
subroutine DipoleMoment()
```

- loweredCamelCase
for variable names
Other languages: routine names/var names

```
integer :: matrixBlockSize = 0
```

- CAPS_AND_UNDERSCORES
for constants
“Universal standard”

```
integer, parameter :: MAX_SHELLS = 4  
real, parameter :: CUTOFF_RADIUS = 10.0
```

- lower_underscored
not used by us
Other languages: var names

Do and Don'ts in coding

Use spacing at your advantage

X `do i = k + 1, k + n - j`

✓ `do i = k+1, k+n-j`

Excessive number of spaces

Loop boundaries are more evident

Spaces

```
subroutine DipoleMoment(masses, coordinates, charges, dipole)
```

```
  real(KREAL), intent(in) :: masses(:)
```

```
  real(KREAL), intent(in) :: coordinates(:, :)
```

```
  real(KREAL), intent(in) :: charges(:)
```

```
  real(KREAL), intent(out) :: dipole(3)
```

Separates declaration of routine args from local vars

```
  integer(KINT) :: nAtoms
```

```
  integer(KINT) :: iAtom
```

```
  real(KREAL) :: CoM(3)
```

Done with declaration

```
  CoM = CenterOfMass(masses, coordinates)
```

```
  nAtoms = size(coordinates, 2)
```

```
  dipole = 0.0_KREAL
```

```
  do iAtom = 1, nAtoms
```

```
    dipole = dipole + charges(iAtom) * (coordinates(:, iAtom) - CoM)
```

```
  enddo
```

“self expressive” block.
dipole initialization is part of the “phrase”

```
end subroutine
```

Use spacing to convey visual and narrative elegance

Do and Don'ts in coding

Fonts

Some fonts are better suited for programming than others

Proportional

different letters have different widths

Fixed

different letters have the same width

Proportional fonts in a terminal can give weird results



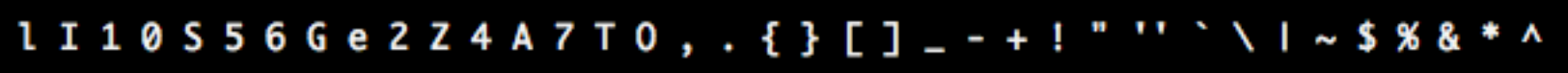
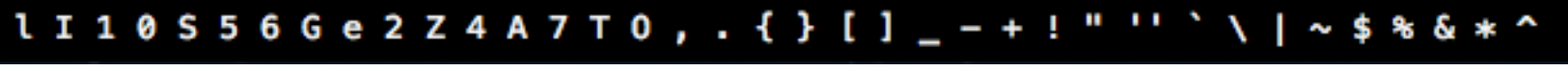
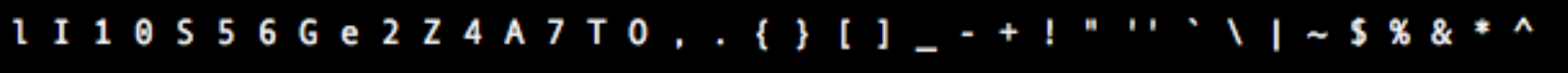
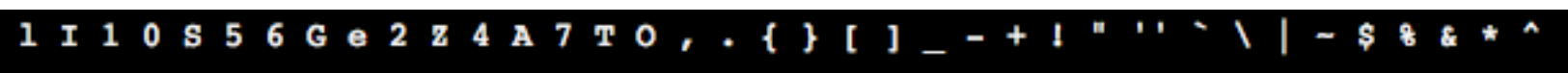
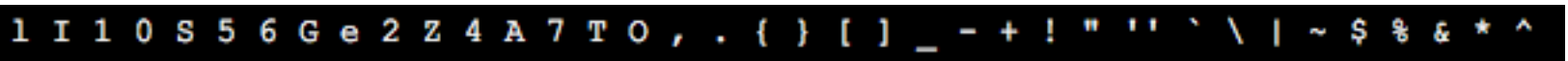
0113653-GettingStartedWithMacros-1.pdf

In non-terminal applications, they are good for documents.
Not good for application where proper spacing and alignment is important

Use Fixed Width fonts for programming

Do and Don'ts in coding

Fonts

Monaco	
Menlo	
Andale	
Courier	
Courier New	

Check difference between:

- lowercase l, uppercase I, number 1
- S and 5
- 6 and G
- 2 and Z
- zero and capital o
- tilde and minus
- dot and comma

Check also:

- parentheses, braces, square brackets
- star (asterisk)
- single, double quotes
- general clarity

For additional fonts, see

<http://www.codinghorror.com/blog/2007/10/revisiting-programming-fonts.html>

<http://hivelogic.com/articles/top-10-programming-fonts/>

Do and Don'ts in coding

Comments

Do and Don'ts in coding

How to comment code

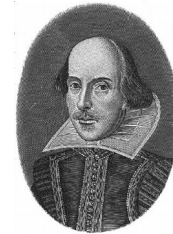
Rule n. 1: Code Tells You How. Comments Tell You Why

Rule n. 2: Comments don't run. Code does.

The best comments are the ones you don't need. Communicate intent with clear code.

Rule n. 3: Excessive commenting is useless.

Brevity forces you to be precise and non-redundant.



Rule n. 4: A wrong comment is worse than no comment at all. *Brevity is the soul of wit.*

Corollary of Rules 2,3,4

Long or excessively formatted comments are likely to be wrong or soon outdated.

Updating them is a hassle. They don't run: does not force people to keep them updated.

Rule n. 5: Keep comments close to the referenced code

Comment for your future self and for others

References:

<http://www.codinghorror.com/blog/2006/12/code-tells-you-how-comments-tell-you-why.html>

<http://www.codinghorror.com/blog/2004/11/when-good-comments-go-bad.html>

Do and Don'ts in coding

Spot the errors!

Ok!

Useless: info for versioning system

Noise: useless ASCII art
non-trivial alignments

Useless: use better var naming.

Wrong: distant from the actual code

Redundant: put eventual clarification
at variable declaration

```
C
C
C Purpose: Numerical calculation of the diagonal elements of the
C exchange-correlation potential matrix for the adaptive
C grid generation.
C
C History: - Creation (31.03.98, MK)
C           (28.01.02, AMK)
C           (21.05.03, AVA)
C
C *****
C
C List of local dimensions:
C
C DSTO: Dimension of Slater typ orbital matrix.
C DVXC: Dimension of exchange-correlation vector.
C
C List of local variables:
C
C LLBATCH: Lower limit of a grid point batch.
C LLGPB : Lower limit grid point of a batch.
C NGPB  : Number of grid points in the current batch.
C ULBATCH: Upper limit of a grid point batch.
C ULGPB : Upper limit grid point of a batch.
C XCVDD  : Diagonal elements of the exchange-correlation matrix.
C
C List of local dynamical fields:
C
C D1HTO: First derivatives of Hermite typ orbitals.
C D1STO: First derivatives of the basis functions.
C HTO  : Function values of Hermite typ orbitals.
C STO  : Function values of the basis functions.
C VGSTO: Work field for GGA exchange-correlation potential.
C
C *****
C
```

Do and Don'ts in coding

How to comment code

- ✓ What a function does, in the minimum amount of text, unless self explanatory
- What each parameter does, unless self explanatory, in three or four words
- What it returns, unless self-explanatory

```
subroutine NewPrivate(self, blockSizes, name)
  ! Creates an invalid (e.g. not defined in its size and blocks) square matrix divided into blocks.
  ! An invalid matrix must be initialized by specifying the block sizes at some point.
  ! It can be specified at constructor call time through the optional blockSizes parameter.
  ! In this case, the matrix will be valid from the start.
  type(BlockMatrixType), intent(out) :: self  !< matrix object
  integer(KINT), intent(in), optional :: blockSizes(:) !< size of each block (columns. Rows are the same)
  character(*), intent(in), optional :: name !< optional name for debugging.
```

- ✓ Algorithmic references

```
subroutine CubicRealSolutions(a,b,c,d, solutions, numSolutions)
  ! Finds the solutions of the cubic  $a*x^3 + b*x^2 + c*x + d = 0$ 
  ! from Numerical Recipes in F77. page 179
```

- ✓ Arcane details that may be unexplainable or non evident from the code by itself

```
! we store the molecule immediately so that the GUI can visualize things.
! It will be overwritten later by the optimized molecule.
call StoreMolecule(self%appCommon%rkFile, self%appCommon%chemicalSystem)
```

- ✓ Unusual units (e.g. different from the internal standard)

```
real :: pressure      ! in atm
```


Do and Don'ts in coding

How not to comment code

Useless comments

```
callCount = 0 ! set callcount to zero  
callCount = callCount + 1 ! increment callcount of one
```

In languages other than English

```
subroutine printHello()  
    ! questa routine stampa un saluto sullo schermo
```

To compensate poor coding practices

```
real :: om ! Inversion angle
```

As dividers in place of proper subroutines

```
! *** Initialize SCF ***  
<lot of code>  
  
! *** Load Hamiltonian ***  
<lot of code>  
  
! *** Load density matrix ***  
<lot of code>  
  
! *** Build core density matrix ***  
<lot of code>  
  
! *** Disable MO exchange ***  
<lot of code>
```



```
call InitializeSCF()  
call LoadHamiltonian()  
call LoadDensityMatrix()  
call BuildCoreDensityMatrix()  
call DisableMOExchange()
```

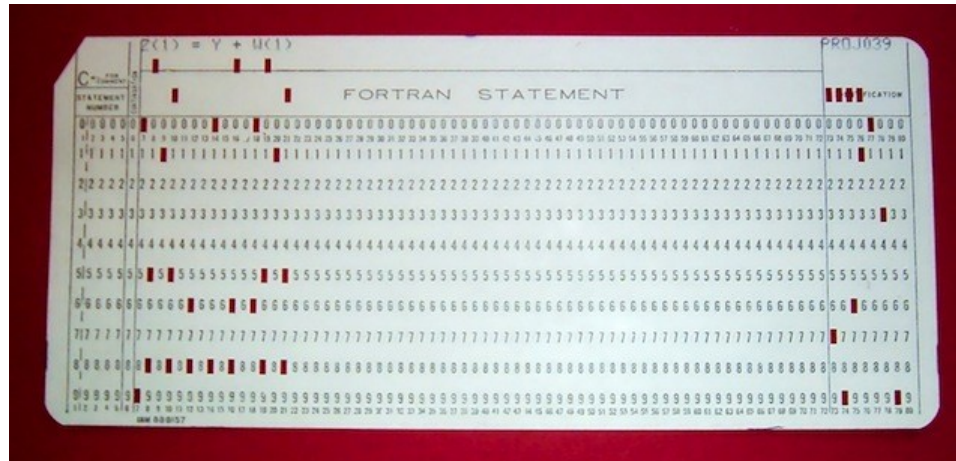
Do and Don'ts in coding

Good programming practices

Do and Don'ts in coding

Implicit

Old Fortran code was typed on punch cards



One line = one card

Less lines = less cards



Do and Don'ts in coding

Implicit

Fortran assumes the type (real, integer, etc...) from the first letter.
When it finds a variable name never seen before

- It assigns a type according to the first letter
- “Slaps a random value” into it (inexact, but net result is the same)
- Use it as-is

Very dangerous

```
integral = 5.0 ; wval = 3.0  
mval = 4.0 ; aval = 2.0  
val = sqrt(2*aval/mval*(wval**2))-2*integral-  
      max(integral*sqrt(integral-mval*3.0*aval)*integral*qval*&  
      exp(wval*2.0/(integral),0.0)-min(integral,0.0)
```

With implicit declaration the mistyped variable contains a random number.
The program will compile, execute, and produce wrong results, possibly
different among computers, independent runs, or code layout.

Always use “implicit none”


The compiler will report undeclared variables and refuse to compile.


Do and Don'ts in coding


Outdated coding style

Don't


 `character*100 foo`


 `real*8 foo`

 `real*8 eta
parameter (eta=0.001d0)`

 `real, dimension(:,,:), allocatable :: h`

Do

 `character(len=100) :: foo`

 `integer :: kreal = kind(0.0d0) ! kind declaration
real(kreal) :: foo`

 `real(kreal), parameter :: eta = 0.0_kreal`

 `real, allocatable :: h(:,,:)`

Do and Don'ts in coding

Resources: use and relinquish

Always deallocate what you allocate

Failing to do so creates a ***memory leak***
memory that cannot be reclaimed until the program is over

```
allocate(foo)
<code>
deallocate(foo)
nullify(foo) ! good practice if foo is a pointer
```

Always close files you open

Data loss may potentially occur if the file is not properly closed.
Number of total open files on the system can be limited.
Additional problems may occur on Network File Systems (NFS)

```
open(unit, file=self%filePath, status='old')

read(unit, '(A)', end=999) tempString

999 close(unit)
```

Do and Don'ts in coding

Misc

Do and Don'ts in coding

No Globals

Avoid global variables

- Express fully visible state of the program as a whole
- Hard to understand where they are actually relevant
- Pollute global namespace
- Their use always involves side effects



Not these....

Some definitions

Namespace: a “container” for visible routines and variables.

The global namespace is the one visible to the program as a whole.

Side effect: when a routine, in addition to returning a value, it also modifies state of the outside world through additional channels.

With Side effects, a program's behavior depends on history. Coupling is increased.

Harder to read/understand/debug: requires knowledge about context, internals and possible histories.

Side effects are not necessarily bad
Some of them should definitely be avoided or limited in scope

Do and Don'ts in coding

“One, Two, Many”

Piraha language has no specific number above two.
Any other amount is “many”.



When is the number of arguments too much?

```
SUBROUTINE XCPOT(RHOALP, RHOBET, RHOTOT, DRHOALP, DRHOBET, DRHOTOT,  
$              ZSPIN, VXALP, VXBET, VXGALP, VXGBET, VCALP, VCBET,  
$              VCGALP, VCGBET, VCG2ALP, VCG2BET, NGPB)
```

Most of these entities are actually conceptually linked together
and should be moved around together.

Use Fortran types to collect entities that belong together

Pass this type around

It can also indicate a function/subroutine that is trying to do too much.

Analyze what the routine is doing, split it into multiple routines

Do and Don'ts in coding

Miller's Law

Humans cannot deal with too much information at once.
We tend to be comfortable with maximum 7 ± 2 entities at once.

Miller, G. A.

"The magical number seven, plus or minus two: some limits on our capacity for processing information".

Psychological Review 63 (2): 81–97 (1956)

DOI:10.1037/h0043158

Compartmentalization

Splitting concepts into (sometimes more or less arbitrary) parts
Enforce thought processes preventing these parts to mix together again

Reduce concepts to hold at once to a bare minimum

Partition well defined areas of competence in your code

Do and Don'ts in coding

Don't pass array sizes to routines
Use the appropriate intrinsic

```
subroutine diagonalize(matrix, nRows, nCols)
  integer, intent(in) :: nRows
  integer, intent(in) :: nCols
  real :: matrix(nRows,nCols)
```

```
subroutine diagonalize(matrix)
  real, intent(in) :: matrix(:, :)

  integer :: nRows, nCols ! potentially useless

  nRows = size(matrix,1) ! potentially useless
  nCols = size(matrix,2) ! potentially useless
```

Note: This works only for routines having an explicit interface!
(e.g. routines contained into modules)

Do and Don'ts in coding

Routines that do too much

```
subroutine mulliken()
```

computes mulliken charges, total charge, dipole, norm of dipole,
stores this info in globals and writes it to a file.

Negative achievements unlocked:

- increases the number of arguments to pass
- makes it harder to reuse in a different context or program
- makes it harder to debug
- makes it longer
- increases the number of variables declared inside the routine
- does not give any hint where the other things are computed
- does not communicate clearly what it **really** does

**Routines should do one simple, easy operation,
that can be expressed with a short, meaningful name.**

Do and Don'ts in coding

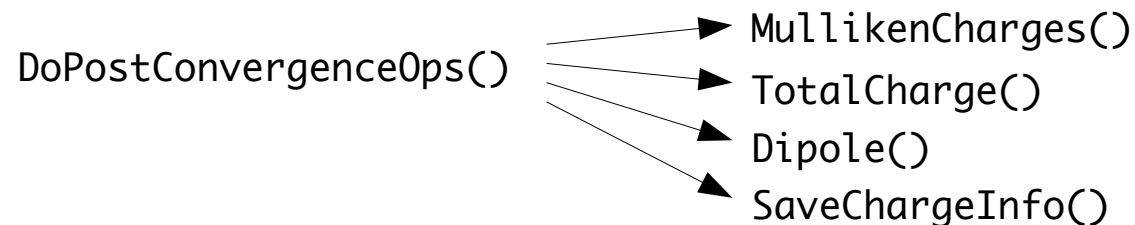
Routines longer than two pages

A long routine is probably doing too much.
Chunk it into smaller routines and call them.

Combine “routines that do” by means of “routines that coordinate”.

Each “Routine that do” should perform one and only one operation

“Routines that coordinate” should perform operations allowing communication and workflow among the “routines that do”.



Do and Don'ts in coding

The IF soup

```
if (len_trim(name) /= 0) then
  call GetMatrixByName(name, matrix, found);
  if (found) then
    if (size(matrix,1) == 1) then
      ! <code 1>
    else
      ! <code 2>
    endif
  else
    <code 3>
  endif
else
  <code 4>
endif
```

```
if (len_trim(name) == 0) then
  <code 4>
  return
endif

call GetMatrixByName(name, matrix, found)

if (.not. found) then
  <code 3>
  return
endif

if (size(matrix,1) == 1) then
  ! <code 1>
else
  ! <code 2>
endif
```

Deeply nested ifs point at bad code organization or excessive complexity

Rework logical statements, flow and eventually extract subroutines.

Do and Don'ts in coding

Crimes against humanity

Do and Don'ts in coding

Don't use goto

Seriously, **don't use it**. I mean it.

Edsger W. Dijkstra

“go to statement considered harmful”

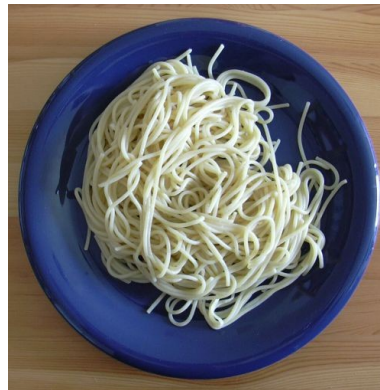
Communications of the ACM - Letters to the editor, Vol. 11 (3), **1968**

DOI: 10.1145/362929.362947

Goto statements can always be replaced with proper structured programming

- do/enddo/exit/cycle
- if/else if/else/endif
- subroutine/end subroutine
- and so on.

Goto makes the code almost impossible to read
“spaghetti code” effect.



Do and Don'ts in coding

Goto: one exception

Only one exception, due to missing language feature

Error Recovery and Rollback

```
subroutine doStuff()  
  logical :: failed  
  
  call doStuffStepOne(failed)  
  if (failed) goto 999  
  
  call doStuffStepTwo(failed)  
  if (failed) goto 998  
  
  call doStuffStepThree(failed)  
  if (failed) goto 997  
  
  ! success  
  return  
  
997    call undoStepTwo()  
998    call undoStepOne()  
999    return  
end subroutine
```


Most recent programming languages use **exceptions** to solve this case with more elegance and flexibility.

Do and Don'ts in coding

Don't collate arrays

 `integer :: bonds(MAX_ATOMS, 0:MAX_BONDS)`

- Hackish
- Non communicative
- Uses a “zero value” to hold counting.
unexpected until one checks the declaration.
- One-of-a-kind exception: limited to integers

 `type BondSetType
 integer :: bonds(MAX_ATOMS, MAX_BONDS)
 integer :: numBonds(MAX_ATOMS)
end type`

- More explicit
- Defines and names a new entity, the BondSet
- Uses the same amount of space
- Same structure can be also used for reals

If you have to keep the first form, at the very least do

```
<... close to bonds declaration ...>  
integer, parameter :: NUM_BONDS_IDX = 0
```

```
<...later...>  
bonds(iAtom, NUM_BONDS_IDX) = 2
```

... but seriously, change it

Do and Don'ts in coding

Don't compare reals for equality!

```
program foo
  real :: r

  r = 0.0
  r = r+0.1
  print *, r, (r == 0.1)
  r = r+0.1
  print *, r, (r == 0.2)
  r = r+0.1
  print *, r, (r == 0.3)
  r = r+0.1
  print *, r, (r == 0.4)
  r = r+0.1
  print *, r, (r == 0.5)
  r = r+0.1
  print *, r, (r == 0.6)
  r = r+0.1
  print *, r, (r == 0.7)
  r = r+0.1
  print *, r, (r == 0.8)
  r = r+0.1
  print *, r, (r == 0.9)
  r = r+0.1
  print *, r, (r == 1.0)
end program
```

```
stefanos-imac:~ borini$ ./a.out
0.10000000    T
0.20000000    T
0.30000001    T
0.40000001    T
0.50000000    T
0.60000002    T
0.70000005    F
0.80000007    F
0.90000010    F
1.0000001     F
```

Reals are **always approximated** by the machine representation.
You can only compare them within a given tolerance.

Integer math is exact. Real math is approximated.

See also:

<http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm>

<http://floating-point-gui.de/>

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Stefano Borini – Advanced programming in Fortran

Do and Don'ts in coding

Don't use reals on cycle loops (compiler should warn)
Use only integers

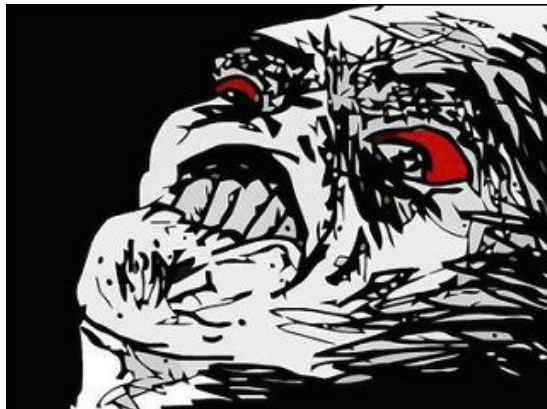
```
real :: distance
```

```
do distance = 1.0, 5.0, 0.1 ! ARGH!!!  
  print *, distance  
enddo
```

```
integer :: i  
real(kreal) :: distance
```

```
do i = 10, 50  
  distance = real(i, kind=kreal)/10.0_kreal  
  print *, distance  
enddo
```

I am serious



Code Evolution

A complex system that works is invariably found to have evolved from a simple system that worked.

The inverse proposition also appears to be true

A complex system designed from scratch never works and cannot be made to work.
You have to start over, beginning with a working simple system.

Gall's Law

“Systemantics: How Systems Really Work and How They Fail”

- ✓ Start simple
- ✓ Have code evolve
- ✓ Direct the change toward better code
- ✓ Keep it working while doing that!

How do we know if the code is still working ?

Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!
Testing! Testing! Testing! Testing! Testing!

Do and Don'ts in coding

Summing up

- ✓ Don't repeat yourself
- ✓ Reduce the burden on the reader's short term memory
- ✓ Take advantage of human senses (color, pattern recognition)
- ✓ Limit scope
- ✓ Aggregate related info
- ✓ Document through code
- ✓ Comment with wisdom
- ✓ Exploit any help the compiler can give
- ✓ Do simple things, then combine them together
- ✓ Principle of least astonishment
- ✓ The fact that you can do something does not mean that you should