```
for(i=0; i<100000; i++) {
  if(a > b) {
    a = func1(c,d,i*10);
  }
  else if(a < b) {
    a = func2(e,f,i*10);
  }
  else {
    a = func3(g,h,i*10);
  }
}
```

```
if(a > b) {
  for(i=0; i<1000000; i+=10) {
    a = func1(c,d,i);
  }
}
else if(a < b) {
  for(i=0; i<1000000; i+=10) {
    a = func2(e,f,i);
  }
}
else {
  for(i=0; i<1000000; i+=10) {
    a = func3(g,h,i);
  }
}
```

Scientific Programming

LECTURE 6: OPTIMIZATION

WHY YOUR CODE MAY NEED OPTIMIZATION?

- The code is too slow, like fundamentally slow
- The code does not reach the precision required
- The code does not fit into the memory
- In any other case focus on functionality and let compilers handle optimization

HOW TO GO ABOUT OPTIMIZATION?

- Start by writing a clean, well structured and well documented code
- Debug this code to make sure it works and does what it is supposed to do
- Only now you can start optimization
- Optimization comes in multiple levels: statement (local), block (e.g. loops) and subroutines

LOCAL OPTIMIZATION

- Assign names to constants and use those names throughoutout the code. E.g.
  `REAL(8), PARAMETER :: Pi=3.1415D0`
- In every assignment statement try to combine variables and constants of the same type
- For integer power operation:
  use **`y = x**6`** instead of **`y = x**6.0`**
- Use intrinsic functions when possible:
  `A=SQRT(SQRT(B))` is faster than **`A=B**0.25`**
- Assign repeated operations to a temporary variable or at least keep the sequence of such operations; instead of
  `a=atan(33./(1.-y**2))+(1.-y)*b/2.*(1.+y)`
  use:
  ```
  onemy2=1.-y*y
  a=atan(33./onemy2)+onemy2*b/2.
  ```

LOCAL OPTIMIZATION

- Operation timing:
  integer math
  - multiplication
  -- division
  --- addition
  ---- subtraction
  ----- power
  ------ elementary functions
  (min/max, sqrt, exp/log, trigonometry)
- This means that **`amean=(a1+a2+a3+a4)*0.25`** will be faster than **`amean=(a1+a2+a3+a4)/4.`**
- Do integer math instead of the floating point whenever it is possible

BLOCK OPTIMIZATION (LOOPS)

- If you can make the loop header create all necessary indices, do it
- Pre-compute all parts not changing inside the loop
- If you use accumulators, do not touch them until the end of the loop

```
p=0.0; x=1.0; y=0;
for(i=s=0; i<n; i++,s+=10) {p=p+func(x,y,i);
x=y; y=s;}
```

- Help memory pre-fetching:

```
for(i=s=0; i<n;)
{
  p=p+func(x,y,i++); x=y; y=s; s+=10;
  p=p+func(x,y,i++); x=y; y=s; s+=10;
  p=p+func(x,y,i++); x=y; y=s; s+=10;
  p=p+func(x,y,i++); x=y; y=s; s+=10;
}
```

- Avoid if- and goto-statements inside the loop

AVOIDING IF AND GOTO STATEMENTS

```
      if(x.gt.3) goto 11
      a=11+b+c+d
      b=c
      c=d
      d=x
      goto 12
11    a=12+b+c+d
      b=x
      c=b
      d=c
12    ...
```

```
if(x.gt.3) then
   a=11+b+c+d
   b=c
   c=d
   d=x
 else
   a=12+b+c+d
   b=x
   c=b
   d=c
 endif
```

WHEN IF IS UNAVOIDABLE

```
    subroutine qroots(a,b,c,x1,x2,flag)
    ...
c
c Finding roots of a quadratic equation
c
    b1=0.5*b/a
    det=b1*b1-c
    if(det.lt.0.) then
      x1=0.
      x2=0.
      flag=2
      return
    else if(det.eq.0.) then
      x1=-b1
      x2=x1
      flag=1
      return
    endif
    det=sqrt(det)
    x1=-b1-det
    x2=-b1+det
    flag=0
    return
    end
```

At least try to minimize the work in each case

ULTIMATE SPEED OPTIMIZATION

- Find the best algorithm for your task and look for its professional implementation. For example, the LAPACK implementation of the LU decomposition beats your simple-minded Gauss elimination by a factor of 3÷5×N/logN
- Even if you do not find a readily-available library look around for the source code of a better algorithm: Marquardt-Levenberg is vastly superior to the gradient search method in optimization problems of a moderate size.

TRADING MEMORY FOR SPEED

Tabulate complex functions in the initialization section of the code.

```
Function longa(x)
real xx(10000),yy(10000),yy2(10000)
integer i
logical first
save first,xx,yy,yy2
data first/.true./
if(first) then
  first=.false.
  do i=1,10000
     xx(i)=i*0.01-50
     yy(i)=sin(xx(i))**2+exp(-(xx(i)/10.)**2)
  enddo
  call spline_init(xx,yy,yy2)
endif
call spline_interp(xx,yy,yy2,x,longa)
return
end
```

OPTIMIZING FOR PRECISION

- Find where precision is lost: subtraction/addition of comparable or very different numbers
- Try to fix the problem by increasing precision
- Try to fix the problem by centering variables

```
xm=mean(xx)
call spline_init(xx-xm,yy,yy2)
call spline_interp(xx-xm,yy,yy2,x-xm,y)
```

- Create generic interface to your tools

GENERIC INTERFACE

```
MODULE SPLINES
    INTERFACE SPLINE_INIT
      MODULE PROCEDURE SPLINE_INIT8, SPLINE_INIT4
    END INTERFACE
    INTERFACE SPLINE_INTER
      MODULE PROCEDURE SPLINE_INTER8, SPLINE_INTER4  END
  INTERFACE
    INTERFACE BEZIER_INIT
      MODULE PROCEDURE BEZIER_INIT8, BEZIER_INIT4
    END INTERFACE
    INTERFACE BEZIER_INTER
      MODULE PROCEDURE BEZIER_INTER8, BEZIER_INTER4
    END INTERFACE
  CONTAINS
        SUBROUTINE SPLINE_INIT8(X,Y,Y2)
  !
  !  Computes second derivative approximations for cubic spline
  !
        IMPLICIT NONE
        REAL (KIND=8) :: Y(:),Y2(:)
        REAL (KIND=8) :: X(:)
        INTEGER       :: N,I
        REAL (KIND=8) :: U(SIZE(X)),SIG,P,YY1,YY2,YY3
   ...
```

OPTIMIZATION FOR MEMORY

- Recycle arrays and variables
- If you use elements of an array recurrently (e.g. element **`i`** is only used after element **`i-1`** and before element **`i+1`**) the whole array may be replaced by a single variable
- Use less memory consuming algorithms: conjugate gradients instead of Newton optimization
- Quite often reducing memory means reducing performance but not always: if you can squeeze the whole memory of a computationally heavy routine into cache memory it will run much faster!

CONCLUSIONS

1. Start by writing a clear and well debugged code
2. Run a few tests – this will be your reference
3. Identify parts that represent bottlenecks. It is a good idea to separate these into subroutines/modules
4. Concentrate on optimization of only crucial parts
5. Start by find the best algorithm.
6. Next, help the compiler of doing small code restructuring
7. Finally, us the compiler optimization flags to do automatic optimization. Verify the optimized code against the reference version.

NEXT LECTURE: MIXING LANGUAGES

The lecture is on Monday October 17th at 10:15.

Don’t forget the Homework Part II