Problem. Run the following program several times (with i = 1,2,3,4,5) and explain the results.

```
program test_int
    implicit none
    integer :: m,i
    m = 2147483645
    do i=1,10
        print*,'i=',i,': m+i=',m+i
    end do
end program test_int
```
Run the program several times, with i = 1,2,3,4,5 and explain the results.

Problem. Run the following program and explain the results.

```
program test
    real :: a, p
    double precision :: b
    print*, 'please provide p:'
    read*, p
    b = (1.99d0+p)*(2.d0**127)
    print*, b
    a = b
    print*, a
    print*, 'normal end here !'
end program test
```
Problem. Long summations may lead to roundoff error accumulation, and the total result can be quite inaccurate.

Consider the sequence (in this order) \(x_1 = 1, x_2 = x_3 = 2^{-1}, \) etc. such that there are \(2^p\) consecutive terms of value equal to \(2^{-p}\). Consider the sum containing \(p = 0\) through \(p = 7\) groups. Clearly the exact value of the sum is 8 (since each group of terms adds up to 1).

Write programs to:

- compute the sum directly (adding \(x_1\) to \(x_2\), the result to \(x_3\) etc.) in single precision;
- compute the sum in double precision;
- compute the sum in single precision but start with the last element (elements are sorted in increasing order);
- use Kahan’s formula (single precision).

Kahan’s summation algorithm tries to compensate for the roundoff errors of each summation stage:

```fortran
sum = 0.0
carry = 0.0
do j=1,N
   y = carry + x(j)
   tmp = sum + y
   carry = (sum-tmp) + y
end do
sum = tmp
sum = sum + carry
```

Comment on how accurate are the results. Repeat the experiments for a sequence containing groups from \(p = 0\) to \(p = 9\). The exact sum is now 10. How accurate are the results?

Explain in detail the mechanism by which Kahan’s summation works.

Problem. Run the following two programs and explain the results:

```fortran
program test
   implicit none
   real :: x
   x = 1.0/2.0
```

if ( (2.0*x) .eq. 1.0 ) then
    print*, 'Correct'
else
    print*, 'Funny'
end if
end program test

and

program test
    implicit none
    real :: x
    x = 1.0/3.0
    if ( (3.0*x) .eq. 1.0 ) then
        print*, 'Correct'
    else
        print*, 'Funny'
    end if
end program test

Problem.

program test
    real :: a=1.0, b=-1.0E+8, c=9.999999E+7
    read :: d, r1, r2
    d = sqrt(b**2-4.0*a*c)
    r1 = (-b+d)/(2.0*a)
    r2 = (-b-d)/(2.0*a)
    print*, r1, r2
end program test

The exact results are -1 and -c, and we expect the numerical results to be close approximations. Run the code and explain the results. What kind of errors are corrupting the result? Which is the critical stage of the calculation?

Re-run the code with all variables declared double precision. Explain the results.

To overcome the cancellation implement the mathematically equivalent formulas:

\[ e_{1,2} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}. \]
Problem. Consider the Fortran code

```fortran
program test
    real :: x=100000.0, y=100000.1, z
    z = y-x
    print*, 'z=',z
end program test
```

We would expect the output

\[ Z = 0.100000 \]

What do we get? Explain.

Problem. Run the code and explain the results:

```fortran
program test
    real :: x=12345.6, y=45678.9, z=98765432.1
    real :: w1, w2
    w1 = x*y/z
    w2 = y*(x*(1.0/z))
    print*, w1-w2
end program test
```

Problem. The following Fortran90 program performs some messy calculations, like division by 0, etc. Run this program and comment on its behavior.

```fortran
program test_arit
    real :: a, b, c, d
    c = 0.0
    d = -0.0
    print*, 'c=',c,' d=',d
    a = 1.0/c
    print*, a
    b = 1.0/d
    print*, 'a=',a,' b=',b
    print*, 'a+b=',a+b
    print*, 'a-b=',a-b
    print*, 'a/b=',a/b
end program test_arit
```
Problem. Write a program that computes the smallest positive floating point number \((2^{-p})\) in single and in double precision. What is \(p\) in each case?

Problem. The following F90 code fragment is an attempt to find the \(\epsilon\)-machine. By changing the definition of the parameterized types it can run for single or double precision.

```
MODULE EPS_MACHINE_1
    ! Single precision
    INTEGER, PARAMETER :: myreal = SELECTED_REAL_KIND(6,30)
    ! Double precision
    ! INTEGER, PARAMETER :: myreal = SELECTED_REAL_KIND(14,300)
    CONTAINS

    REAL(kind=myreal) FUNCTION WLAMCH( )

        INTEGER     :: i
        REAL(kind=myreal) :: Sum, Eps
        REAL(kind=myreal), PARAMETER :: ONE=1.0_myreal

        Eps = ONE
        DO i = 1, 80
            Eps = Eps*0.5_myreal
            Sum = ONE + Eps
            IF (Sum.LE.ONE) THEN
                WLAMCH = Eps*2
                RETURN
            END IF
        END DO
        PRINT*, 'ERROR IN WLAMCH. Eps < ', Eps

    END FUNCTION WLAMCH

END MODULE EPS_MACHINE_1
```

Run the program. The returned accuracy is smaller than expected (in fact, it corresponds to extended precision).
The functions that find the machine accuracy in LAPACK (SLAMCH, DLAMCH) are implemented along the following lines:

MODULE EPS_MACHINE

   ! Single precision
   INTEGER, PARAMETER :: myreal = SELECTED_REAL_KIND(6,30)
   ! Double precision
   ! INTEGER, PARAMETER :: myreal = SELECTED_REAL_KIND(14,300)

CONTAINS

REAL(kind=myreal) FUNCTION WLAMCH( )

   INTEGER :: i
   REAL(kind=myreal) :: Sum, Eps
   REAL(kind=myreal), PARAMETER :: ONE=1.0_myreal

   Eps = ONE
   DO i = 1, 80
      Eps = Eps*0.5_myreal
      CALL WLAMCH_ADD(ONE,Eps,Sum)
      IF (Sum.LE.ONE) THEN
         WLAMCH = Eps*2
         RETURN
      END IF
   END DO
   PRINT*,’Error in WLAMCH. Eps < ’,Eps

END FUNCTION WLAMCH

SUBROUTINE WLAMCH_ADD( A, B, Sum )
   REAL(kind=myreal) A, B, Sum
   Sum = A + B
END SUBROUTINE WLAMCH_ADD

END MODULE EPS_MACHINE
Now the reported accuracy is in line with our expectations.

Explain in detail what causes the difference in results between the 2 implementations.