2.5.1 Addition Operation

Addition operation can be performed on normalized floating-point numbers if the exponents of these numbers are equal. If the exponents are not equal, then the exponent of the numbers with smaller exponent is made equal to the larger exponent and its mantissa is modified. Shifting the decimal point to the left does this modification by number of places equal to the positive difference between the two exponents. Now the mantissas of these numbers are added.

For example, if 0.7253E2 and 0.5467E5 are to be added, the decimal point of mantissa of 0.7253E2 is shifted by 3 (5–2) positions to the left and the exponent is incremented by 3. The number after this modification becomes 0.0007E5. These two numbers can now be added by adding their respective mantissas.

Note that the mantissa of the number representing sum of these two normalized floating-point numbers may not be normalized. Therefore, to complete the operation, the mantissa must be normalized. Further note that the mantissa of the sum (before normalization) can be maximum of +1.9999, there at most we need to shift the decimal point to left by one position in order to normalize it. As a result of this, the exponent of the sum is incremented by 1. Because of normalization process, the exponent of the sum may become greater than +99 that implies that the number is greater than the largest number that our hypothetical computer can store. This condition is called *overflow*, and the system will signal this error.

Example 2.3: Add 0.5467E5 to 0.7253E2

Solution: The decimal point of the mantissa of 0.7253E2 is shifted three positions to left. It becomes 0.0007, whereas the digits 3, 5, and 2 are chopped off i.e. truncated. The exponent is incremented by 3. Therefore, the number after normalization becomes 0.0007E5. These numbers can be added now as shown below:

Addend	0.0007E5
Augend	0.5467E5
Sum	0.5474E5

Example 2.4: Add 0.5467E2 to 0.7254E2

Solution: Since the exponents are already equal, they can be directly added as:

Addend	0.7254E2
Augend	0.5467E2
Sum	1.2721E2

In this case, the mantissa of the sum is greater than 1.0, so the decimal point is shifted to the left by one position and the exponent is increased by 1. The last digit of the mantissa of sum is chopped off giving the normalized sum as 0.1272E3.

Example 2.5: Add 0.5467E99 to 0.7254E99

Solution: Since the exponents are already equal, they can be directly added as:

```
Addend   0.7254E99
Augend   0.5467E99
         ---------
Sum      1.2721E99
```

In this case, the mantissa of the sum is greater than 1.0, so the decimal point is shifted to the left by one position and the exponent is increased by 1. The last digit of the mantissa of sum is chopped off, giving the normalized sum as 0.1272E100. Since this number is greater than the largest number that our hypothetical computer can handle, it is a case of *overflow* and the system will indicate this condition.

Algorithm 2.1: To add two normalized floating-point numbers

Let us consider that x_1, x_2 be the mantissas and e_1, e_2 be the exponents of the two normalized floating-point numbers. Further consider that x and e represents the mantissa and exponent of the sum. The integer variable k is used as temporary variable.

```
Begin
    read: $x_1$, $e_1$, $x_2$, $e_2$
    set $k = | e_1 - e_2 |$
    if ($e_1 > e_2$) then
        set $x_2 = x_2 / 10^k$
        set $e = e_1$
    else
        set $x_1 = x_1 / 10^k$
        set $e = e_2$
    endif
    set $x = x_1 + x_2$
    if ( $x \geq 1.0$ ) then
        set $x = x / 10$
        set $e = e + 1$
    endif
    if ( $e > 99$ ) then
        write: "Overflow . . . "
    else
        write: x, e, "as mantissa and exponent of the sum"
    endif
End.
```

2.5.2 Subtraction Operation

Like addition operation, the subtraction operation can be performed on normalized floating-point numbers if the exponents of these numbers are equal. If the exponents are not equal, then the exponent of the numbers with smaller exponent is made equal to the larger exponent and its mantissa is modified. Shifting the decimal point to the left does this modification by number of places equal to the positive difference between the two exponents. Now the mantissas of these numbers are subtracted. Here also, the mantissa of the number representing difference of these two normalized floating-point numbers may not be normalized.

Therefore, to complete the operation, the mantissa must be normalized. Further note that the magnitude of mantissa of the difference (before normalization) will be in the range 0.0000 to 0.9999. These extreme values will occur in case the numbers are equal or the mantissa of the number to be subtracted is 0.0000 (zero) respectively. Therefore, the decimal may have to shift to right by more than one position. Correspondingly, the exponent is decremented by one for each shift of the mantissa. During this normalization process, the exponent of the difference may become less than –99 which implies that the number is smaller than the smallest number which our hypothetical computer can store. This condition is called *underflow*, and the system will signal this error.

Example 2.6: Subtract 0.7253E2 from 0.5467E5.

Solution: The number 0.7253E2 is normalized, thus resulting in number 0.0007E5. Now the number 0.0007E5 can be subtracted from 0.5467E5 as:

Minuend	0.5467E5
Subtrahend	0.0007E5
Difference	0.5460E5

In this case, the mantissa of the difference is already in the normalized form.

Example 2.7: Subtract 0.7254E5 from 0.7288E5.

Solution: Since the exponents are already equal, the number 0.7254E5 can be directly subtracted from 0.7288E5 as:

Minuend	0.7288E5
Subtrahend	0.7254E5
Difference	0.0034E5

In this case, the mantissa of the difference is less than 0.1, so the decimal point is shifted two positions to the right and the exponent is decreased by 2. The resultant difference becomes 0.3400E3.

Example 2.8: Subtract 0.7254E–99 from 0.7278E–99.

Solution: Since the exponents are already equal, the number 0.7254E5 can be directly subtracted from 0.7288E5 as:

Minuend	0.7278E–99
Subtrahend	0.7254E–99
Difference	0.0024E–99

In this case, the mantissa of the difference is less than 0.1, so the decimal point is shifted two positions to the right and the exponent is decreased by 2. The resultant difference becomes 0.3400E−101. Since this number is smaller than the smallest number that our hypothetical computer can handle, it is a case of *underflow* and the system will indicate this condition.

Algorithm 2.2: To compute difference of two normalized floating-point numbers

Let us consider that x_1, x_2 be the mantissas and e_1, e_2 be the exponents of the two normalized floating-point numbers. Further consider that x and e represents the mantissa and exponent of the difference. The integer variable k is used as temporary variable.

```
Begin
    read: $x_1$, $e_1$, $x_2$, $e_2$
    set $k$ = | $e_1$ - $e_2$ |
    if ($e_1$ > $e_2$ ) then
        set $x_2$ = $x_2$ / $10^k$
        set e = $e_1$
    else
        set $x_1$ = $x_1$ / $10^k$
        set e = $e_2$
    endif
    set $x$ = $x_1$ - $x_2$
    while ( ( | x | < 0.1 ) and ( | x | > 0.0 ) ) do
        set x = x * 10
        set e = e - 1
    endwhile
    if ( e < -99 ) then
        write: "Underflow. . . . "
    else
        write: x, e, "as mantissa and exponent of the difference"
    endif
End.
```

2.5.3 Multiplication Operation

In multiplication operation, the mantissas are multiplied and the exponents are added. The mantissa of the product will not be in normalized form. Further note than the magnitude of the product will be greater than 1.0 but less than 10.0. Therefore, at most, the decimal point of the mantissa of the product will shift one position to the left and the exponent is incremented by 1. As a result of this increment the exponent can become +99. Now if the mantissa is negative, this results in underflow else overflow.

Example 2.9: Multiply 0.6543E5 by 0.2255E3

Solution: 0.6543E5 × 0.2255E5 = (0.6543 × 0.2255)E(5+3)

= 0.14754465E8

Thus, we obtain 0.1475E8 after truncating the mantissa of the product to four digits. This product is already in the normalized form.

Example 2.10: Multiply 0.1234E5 by 0.1111E13

Solution: 0.1234E5 × 0.1111E13 = (0.1234 × 0.1111)E(5+13)

= 0.01370974E18

Thus, we obtain 0.0137E18 after truncating the mantissa of the product to four digits. And since this product is less than 0.1, the decimal point is shifted one position to the right and the exponent is decreased by 1. The resultant product becomes 0.1370E17.

Example 2.11: Multiply 0.1234E75 by 0.1111E37

Solution: 0.1234E75 × 0.1111E37 = (0.1234 × 0.1111)E(75+37)

= 0.01370974E112

Thus, we obtain 0.0137E112 after truncating the mantissa of the product to four digits. And since this product is less than 0.1, the decimal point is shifted one position to the right and the exponent is decreased by 1. The resultant product becomes 0.1370E111. Since this number is greater than the largest number that our hypothetical computer can handle, it is a case of *overflow* and the system will indicate this condition.

Example 2.12: Multiply 0.1234E–75 by 0.1111E–37

Solution: 0.1234E–75 × 0.1111E–37 = (0.1234 × 0.1111)E[(–75)+(–37)]

= 0.01370974E–112

Thus, we obtain 0.0137E–112 after truncating the mantissa of the product to four digits. And since this product is less than 0.1, the decimal point is shifted one position to the right and the exponent is decreased by 1. The resultant product becomes 0.1370E–113. Since this number is smaller than the smallest number that our hypothetical computer can handle, it is a case of *underflow* and the system will indicate this condition.

Algorithm 2.3: To multiply two normalized floating-point numbers

Let us consider that x_1, x_2 be the mantissas and e_1, e_2 be the exponents of the two normalized floating-point numbers. Further consider that *x* and *e* represents the mantissa and exponent of the product.

```
Begin
    read: $x_1$, $e_1$, $x_2$, $e_2$
    set $x = x_1 \times x_2$
    set $e = e_1 + e_2$
    if ( $|x| \geq 1.0$ ) then
        set $x = x / 10$
        set $e = e + 1$
    endif
    if ( | x | < 0.1 ) then
        set x = x * 10
        set e = e - 1
    endif
    if ( e > 99 ) then
        write: "Overflow . . . "
    else if ( e < -99 ) then
        write: "Underflow . . . "
    else
        write: x, e, "as mantissa and exponent of the product"
    endif
End.
```

2.5.4 Division Operation

In division operation, the mantissa of the one number is divided by the mantissa of the second and the exponent of the second number is subtracted from the first. The mantissa of the quotient will not be in normalized form. Further note than the magnitude of the quotient may become greater than 1.0, but will always be less than 10.0. Therefore, at most, the decimal point of the mantissa of the quotient will shift one position to the left and the exponent is incremented by 1. As a result of this increment the exponent can become +99. Now if the mantissa is negative, this results in underflow else overflow.

Example 2.13: Divide 0.8888E5 by 0.2000E3

Solution: 0.8888E5 ÷ 0.2000E3 = (0.8888 ÷ 0.2000)E(5–3)

= 4.4440E2

The mantissa of the quotient is greater than 1.0, therefore the decimal point is shifted one position to the left and the exponent is increased by 1. The resultant quotient becomes 0.4444E3.

Example 2.14: Divide 0.9998E5 by 0.1000E–99

Solution: 0.9998E5 ÷ 0.1000E–99 = (0.9998 ÷ 0.1000)E[5–(–99)]

= 9.9980E104

The mantissa of the quotient is greater than 1.0, therefore the decimal point is shifted one position to the left and the exponent is increased by 1. The resultant quotient becomes 0.9998E105. This is a case of *overflow*.

Example 2.15: Divide 0.9998E–5 by 0.1000E99

Solution: 0.9998E-5 ÷ 0.1000E99 = (0.9998 ÷ 0.1000)E(–5–99)

= 9.9980E–104

The mantissa of the quotient is greater than 1.0, therefore the decimal point is shifted one position to the left and the exponent is increased by 1. The resultant quotient becomes 0.9998E−103. This is a case of *underflow*.

Algorithm 2.4: To perform division of two normalized floating-point numbers

Let us consider that x_1, x_2 be the mantissas and e_1, e_2 be the exponents of the two normalized floating-point numbers. Further consider that x and e represents the mantissa and exponent of the quotient.

```
Begin
    read: $x_1$, $e_1$, $x_2$, $e_2$
    set $x = x_1 / x_2$
    set $e = e_1 - e_2$
    if ( $|x| \geq 1.0$ ) then
        set $x = x / 10$
        set $e = e + 1$
    endif
    if ( $|x| < 0.1$ ) then
        set $x = x * 10$
        set $e = e - 1$
    endif
    if ( $e > 99$ ) then
        write: "Overflow . . . "
    else if ( $e < -99$ ) then
        write: "Underflow . . . "
    else
        write: x, e, "as mantissa and exponent of the product"
    endif
End.
```

2.6 PITFALLS OF FLOATING-POINT REPRESENTATION

We have seen in the preceding sections that the mantissas have to be truncated to four digits in order to fit into the normalized floating-point format of the hypothetical computer. This truncation leads to a number of surprising results.

2302

1. It is well known fact that

$$4x = x + x + x + x$$

However, when arithmetic is performed using normalized floating-point representation, the above equation may not hold true.

Example 2.16: Consider x = 0.6667E0, then $4x$ written as 0.4000E1 × 0.6667E0 using normalized floating-point representation gives a value of 0.2666E1. However, if we add 0.6667E0 four times as

$$0.6667E0 + 0.6667E0 + 0.6667E0 + 0.6667E0$$

answer will be 0.2665E1. This happens because of truncation in the intermediate and final results in order to fit into the normalized floating-point format.

2. The associative and distributive laws of arithmetic may not hold true. The proof of this fact is left as an exercise for the readers.
3. Another very important point to remember is that while performing arithmetic in normalized floating-point format, the equality of an expression to zero can never be assured.

Example 2.17: The quadratic equation

$$x^2 + 2x - 2 = 0$$

has roots $-1 \pm \sqrt{3}$. Expressing in normalized floating-point, the roots are 0.7320E0 and −0.2732E1.

If we substitute $x = 0.732\text{E}0$, and solve the expression $(x^2 + 2x) - 2$ as

$$\begin{aligned}(0.7320\text{E}0 \times 0.7320\text{E}0 + 0.2000\text{E}1 \times 0.7320\text{E}0) - 0.2000\text{E}1\\ = (0.5358\text{E}0 + 0.2000\text{E}1 \times 0.7320\text{E}0) - 0.2000\text{E}1\\ = (0.5358\text{E}0 + 0.1464\text{E}1) - 0.2000\text{E}1\\ = 0.1999\text{E}1 - 0.2000\text{E}1 = -0.1000\text{E}{-2}\end{aligned}$$

Ideally, the expression should yield the value zero. The point, which we want to emphasize, is that in algorithms, it is not advisable to write a branching or looping instruction that compares the value of an expression with zero.

2.7 ERRORS

An *error* is defined as the difference between the actual value and the approximate value obtained from the experimental observation or from numerical computation. Consider that x represents some quantity and x_a is an approximation to x, then

$$\begin{aligned}\text{error} &= \text{actual value} - \text{approximate value}\\ &= x - x_a\end{aligned}$$

As mentioned earlier, the errors in computed results may be due to errors in input data and/or computational algorithm. It is important to note that it is impossible to remove the errors in the input data, however errors in calculations can be reduced to some extent.

The errors in the computed results can be classified in following categories:

1. Errors in input data
   - due to approximate measurements, known as *data errors.*
   - due to truncation of the digits, known as *truncation errors.*
   - due to rounding-off the digits, known as *round-off errors.*
2. Computational errors
   - due to pitfalls in computational algorithm.
   - due to truncation of digits during the arithmetic operation, because of limitation of storage.

2.7.1 Data Errors

These are errors that occur due to inaccurate measurements or observations that may be due to limitations of the measuring device. For example, vernier calliper, screw gauge, etc. can measure the quantity accurate to certain smallest value. The accuracy of the measurements also depends on the experience of the person.

2.7.2 Truncation Errors

Truncation errors occur when some digits from the number are discarded. There are mainly two situations when

i) During the representation of numbers in normalized floating-point form. Because only few digits in the mantissa can be accommodated, for example, only four digits in our hypothetical computer. For example, the number 0.0356879 takes the form 0.3568E–1 in our hypothetical computer in the normalized floating-point representation. The digits 7 and 9 have been discarded. This truncation introduces errors in the input data.

ii) During the conversion of a number from one system to another. For example, the decimal number 13.1 has equivalent binary representation 1101.0001100110011. . . It has a repeating fraction and therefore the conversion must be terminated after some digits. This introduces the truncation error.

2.7.3 Round-off Errors

Rounding-off the number also causes truncation but with some adjustment to the last digit retained depending on the values of the truncated digits.

The following numbers are rounded-off to two decimal places:

Given decimal number	0,667	4.3743	8.9954
Rounded-off number	0.67	4.37	9.00

Observe that one has been added to the second decimal place in the first and third number because the third decimal place is greater than five.

The procedure for rounding-off a number to given number of decimal places can therefore be stated as follows:

> Suppose a number is required to be rounded-off to n^{th} decimal place, then one is added to the n^{th} decimal digit if the $(n+1)^{th}$ decimal digit is greater than or equal to 5, otherwise the n^{th} decimal digit is kept unaltered.

The error introduced by rounding-off the numbers to given decimal places is known as round-off error.

2.7.4 Computational Errors

As mentioned earlier, the major sources of computational errors are:

- Normalized floating-point representation
- Truncation of infinite series expansion
- Inefficient algorithm

As we have seen in the preceding sections that while performing the basic arithmetic operations using the normalized floating-point representation, some digits in the mantissa have to be truncated in order to fit into the normalized floating-point form. Such truncations introduce varied amount of errors depending on the size of the numbers and the nature of the operations performed.

Similarly, errors are introduced because of finite representation of an inherently infinite series or process, such as $\sin(x)$, $\cos(x)$, $\log(x)$, e^x etc.. For example, the following infinite series

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

is used to compute the value of $\cos(x)$. Since the series is infinite, it is not possible to use all the terms in the computations. In such cases, the process is terminated after a finite number of terms. Therefore, the terms omitted will introduce error in the computed result. Such errors are also called truncation errors, as they are introduced due to truncation of the infinite series. Errors due to truncation of an infinite can be estimated if we know the number of terms included in the computations.

And finally, the algorithms, if not properly selected or formulated, will also introduce errors in the various steps of computations. As an illustration, we consider an example of computing combinations.

Suppose there are n objects and r objects are to selected at a time, then possible number of combinations is given by

$$^{n}C_r = \frac{n!}{r!(n-r)!}$$

Simple and straight forward approach to compute the factorials is that compute $n!$, $r!$ and $(n-r)!$ separately and then substitute these values in the above formula to compute ${}^nc_r$.

The real problem with this approach occurs when the value of n becomes large. For large values of n, the $n!$ will become too large and may result in an overflow error.

However, this problem can be rectified we express the above formula recursively as

$$ {}^nc_r = {}^nc_{r-1}\left[\frac{n-r+1}{r}\right] $$

where ${}^nc_r = 1$ when $r = 1$.

The above formula can also be expressed iteratively as

$$ {}^nc_r = n\prod_{i=2}^{r}\left[\frac{n-i+1}{i}\right] $$

These formulations will compute ${}^nc_r$ without causing an overflow error provided the final answer is not too large.

2.8 MEASURES OF ACCURACY

It must be remembered that the round-off errors are most difficult to estimate and one has to follow some rules to reduce their effect on the final results. However, the truncation errors can be easily estimated and thus can be effectively reduced. In any case, one needs some measures of accuracy of the results. The two commonly used measures are described below.

2.8.1 Absolute Error

Absolute error is the positive difference between the actual value and the approximate value of a variable. If x is the actual value and x_a is the approximate value of a variable, then the absolute error is given by

$$ \text{absolute error} = |x - x_a| $$

2.8.2 Relative Error

Relative error is the ratio of the error to the actual value of a variable. If x is the actual value and x_a is the approximate value of a variable, then the relative error is given by

$$ \text{relative error} = \frac{x - x_a}{x} $$

2.9 SOME IMPORTANT RULES TO REMEMBER

Theoretical proof of these rules is beyond the scope of this book. However, these rules will be proved numerically in the examples to follow.

Consider x be any number expressed as

$$0.d_1d_2d_3\ldots\times10^n$$

where $d_1, d_2, d_3, \ldots$ are decimal digits.

Rule 1: *Absolute error because of truncation*

If x_a is the approximate value of x after truncation to k digits, then

$$|x-x_a|<10^{n-k}$$

Rule 2: *Relative error because of truncation*

If x_a is the approximate value of x after truncation to k digits, then

$$\left|\frac{x-x_a}{x}\right|<10^{-k+1}$$

Rule 3: *Absolute error because of rounding-off*

If x_a is the approximate value of x after rounding-off to k digits, then

$$|x-x_a|<0.5\times10^{n-k}$$

Rule 4: *Relative error because of rounding-off*

If x_a is the approximate value of x after rounding-off to k digits, then

$$\left|\frac{x-x_a}{x}\right|<0.5\times10^{-k+1}$$

This type of analysis is useful if we want to keep track of the errors that occur when the numbers in a calculation are truncated or rounded-off to a certain number of decimal digits.

Example 2.18: The solution of a problem is given as 3.436. It is known that the absolute error in the solution is less than 0.01. Find the interval within which the exact value must lie.

Solution: Given $x_a = 3.436$. If x is the actual value of the solution, then

$$|x-x_a|<0.01 \quad\Rightarrow\quad x-3.436\,|<0.01$$

From the definition of the absolute value

$$-0.01<x-3.436<0.01$$

Therefore

$$3.436-0.01<x<3.436+0.01 \qquad \Rightarrow \qquad 3.436<x<3.446$$

Hence, the exact value of the solution must lie the interval (3.426, 3.446).

Example 2.19: Let $x = 0.00458529$. Find the absolute error if x is truncated to three decimal digits.

Solution: Given $x = 0.00458529$

$= 0.458529\times10^{-2}$ [in the normalized floating-point form]

$x_a = 0.458\times10^{-2}$ [after truncating to three decimal places]

error $= x - x_a$

$= 0.458529\times10^{-2} - 0.458\times10^{-2}$

$= 0.000529\times10^{-2}$

Hence $|x - x_a| = 0.000529\times10^{-2} = 0.529\times10^{-5}$, which is less than 10^{-2-3}, and hence proves rule 1.

Example 2.20: Let $x = 0.005998$. Find the relative error if x is truncated to three decimal digits.

Solution: Given $x = 0.005998$

$= 0.5998\times10^{-2}$ [in the normalized floating-point form]

$x_a = 0.599\times10^{-2}$ [after truncating to three decimal places]

$$\text{Relative error} = \left|\frac{x-x_a}{x}\right| = \left|\frac{0.5998\times10^{-2}-0.599\times10^{-2}}{0.5998\times10^{-2}}\right|$$

$= 0.00133 = 0.133\times10^{-2}$

which is less than 10^{-3+1}, and hence proves rule 2.

Example 2.21: Let $x = 0.00458529$. Find the absolute error if x is rounded-off to three decimal digits.

Solution: Given $x = 0.00458529$

$= 0.458529\times10^{-2}$ [in the normalized floating-point form]

$x_a = 0.459\times10^{-2}$ [after rounding-off to three decimal places]

error $= x - x_a$

$= 0.458529\times10^{-2} - 0.459\times10^{-2}$

$= -0.000471\times10^{-2}$

Hence $|x - x_a| = 0.000471 \times 10^{-2} = 0.471 \times 10^{-5}$, which is less than $0.5 \times 10^{-2-3}$, and hence proves rule 3.

Example 2.22: Let $x = 0.005998$. Find the relative error if x is rounded-off to three decimal digits.

Solution: Given $x = 0.005998$

$= 0.5998 \times 10^{-2}$ [in the normalized floating-point form]

$x_a = 0.600 \times 10^{-2}$ [after rounding-off to three decimal places]

$$\text{Relative error} = \left|\frac{x - x_a}{x}\right| = \left|\frac{0.5998 \times 10^{-2} - 0.600 \times 10^{-2}}{0.5998 \times 10^{-2}}\right|$$

$$= 0.000333 = 0.333 \times 10^{-3}$$

which is less than $0.5 \times 10^{-3+1}$, and hence proves rule 4.

Example 2.23: Given the solution of a problem as $x_a = 35.25$ with relative error in the solution at most 2%. Find, to four decimal digits, the range of values within which the exact value of the solution must lie.

Solution: Given maximum relative error $= 0.02$

Therefore $$-0.02 < \frac{x - x_a}{x} < 0.02$$

If $$\frac{x - x_a}{x} < 0.02$$

then $x - x_a < 0.02x$ [since $x > 0$]

$\Rightarrow \quad x(1 - 0.02) < x_a$

$$\Rightarrow \quad x < \frac{x_a}{1 - 0.02} = \frac{35.25}{0.98} = 35.9693877551$$

However, if $-0.02 < \dfrac{x - x_a}{x}$ then $x - x_a > -0.02x$ [since $x > 0$]

$\Rightarrow \quad x(1 + 0.02) > x_a$

$$\Rightarrow \quad x > \frac{x_a}{1 + 0.02} = \frac{35.25}{1.02} = 34.3588235294$$

Hence $34.5588 < x < 35.9694$