


```

        od;
# now, expmax - expmin = 1 and powermin <= x < powermax,
# powermin = 2^expmin and powermax = 2^expmax, so expmin is the exponent of x
exponent := expmin;
fi;
infmantissa := x*2^(52-exponent);
if frac(infmantissa) <> 0.5 then mantissa := round(infmantissa)
else
    mantissa := floor(infmantissa);
    if type(mantissa,odd) then mantissa := mantissa+1 fi
fi;
mantissa := mantissa*2^(-52);
fi;
fi;
sgn,exponent,mantissa;
end:

```

Dinechin's mantissa is between 1 and 2 (there is a typo in his original), but I want it between $\frac{1}{2}$ and 1, in the range $\text{RealRange}\left(\frac{1}{2}, \text{Open}(1)\right)$, with some 'readable' notations and for complex floats as well. His original solution has slightly to be modified:

```

> my_nearest := proc(x)
local sign, exponent, mantissa;
sign,exponent,mantissa := IEEEdouble(x);
if mantissa=2 then # happens for 1 -pow(2.0,-55)
    sign*mantissa/4 * 'pow'(2,exponent+2);
else
    sign*mantissa/2 * 'pow'(2,exponent+1);
end if;
end:

nearest:=proc(z) # complex version
local x,y;
Digits:=54;
x,y:= Re(evalf(z)), Im(evalf(z));
my_nearest(x) + my_nearest(y)*I;
end proc;

nearest := proc(z) local x, y; Digits := 54; x, y := Re(evalf(z)), Im(evalf(z)); my_nearest(x) + my_nearest(y)*I end proc

```

As already said: my interest stems from complex numerics. Then there seems to be no agreed convention (ok, maybe C99), especially for functions.

Anyway: for coding I mostly use Microsoft - but now using C++ and the stdlib to handle complex numerics - MS does not assert something towards IEEE, what I would be aware of. And then one is faced with the problem to judge results, for which Maple is fine (say: external callings using DLLs).

Using high precision is ok. But looking at errors one has to accept, that we have representation problems and only within that one can expect something. Relative errors may be denoted in `DBL_EPSILON`, at 'natural constant' in that setting.

```

> DBL_EPSILON:=1.0/2^52;
DBL_EPSILON:=nearest(DBL_EPSILON);

DBL_EPSILON := 0.222044604925031308 10-15
DBL_EPSILON :=  $\frac{1}{2}$  pow(2, -51)

```

Let us look at a 'trivial' task: computing powers (no it is not trivial at all and some libraries do not provide that per se).

First take that function symbolically in Maple and use its (Watson) compiler to have a compiled version (or use `evalhf` by activating parts of the code below):

```

> p:=proc(z::complex(numeric), a::complex(numeric)) return(z^a); end proc;
cp:=Compiler:-Compile(p);

p := proc(z::complex(numeric), a::complex(numeric)) return z^a end proc

cp := proc()
option call_external, define_external(_m4df963f914692ff91006b33d738b4fb0, MAPLE,
LIB = "C:\DOKUME~1\AXELVO~1\LOKALE~1\Temp\AxeL_Vogt-3304\_m4df963f914692ff91006b33d738b4fb0U1yYq2xV.dll");
call_external(0, 67571824, true, false, args)
end proc

```

Then choosing some test values look at results

```

> zTst, aTst:=56+I, 156+I;
zTst, aTst:= evalf(zTst), evalf(aTst);
#zTst, aTst:= nearest(zTst), nearest(aTst);
` `;

```

```

'cp(zTst,aTst)': '%=' %; ``=nearest(rhs(%)); c:=rhs(%):
#'evalhf(zTst^aTst)': '%=' %; ``=nearest(rhs(%)); c:=rhs(%):
'p(zTst,aTst)': '%=' %; ``=nearest(rhs(%)); m:=rhs(%):
#`error absolute` = nearest(m - c);
`error relative` = nearest(abs(1 - c/m));
k*DBL_EPSILON = rhs(%):
``=evalf[3](solve(%, k)) * 'DBL_EPSILON';

zTst, aTst := 56. + 1. I, 156. + 1. I

cp(zTst, aTst) = 0.453799788618191220 10273 + 0.264515621002634284 10273 I
=  $\frac{7555757099289153}{9007199254740992} \text{pow}(2, 906) + \frac{8808359242957195}{9007199254740992} I \text{pow}(2, 905)$ 

p(zTst, aTst) = 0.453799788618194973 10273 + 0.264515621002636308 10273 I
=  $\frac{7555757099289215}{9007199254740992} \text{pow}(2, 906) + \frac{4404179621478631}{4503599627370496} I \text{pow}(2, 905)$ 

error relative =  $\frac{2553654517784193}{4503599627370496} \text{pow}(2, -46)$ 
= 36.3 DBL_EPSILON

```

This says: up to a relative error of 36 DBL_EPSILON the result is ok

Now do the same with VC2005 (the 'free' version), for which I get

```

> (4.5379978861818113e+272, 2.6451562100262837e+272):
[%]: VC=op(1,%) + op(2,%)*I;
``=nearest(rhs(%)); vc:=rhs(%):
#`error absolute` = nearest(m - vc);
`error relative` = nearest(abs(1 - vc/m));
k*DBL_EPSILON = rhs(%):
``=evalf[3](solve(%, k)) * 'DBL_EPSILON';

VC = 0.45379978861818113 10273 + 0.26451562100262837 10273 I
=  $\frac{7555757099288985}{9007199254740992} \text{pow}(2, 906) + \frac{4404179621478499}{4503599627370496} I \text{pow}(2, 905)$ 

error relative =  $\frac{2402368605253773}{4503599627370496} \text{pow}(2, -44)$ 
= 137. DBL_EPSILON

```

One more example:

```

> zTst, aTst:=56.0+2.0*'DBL_EPSILON*I', 156.0+2.0*'DBL_EPSILON*I';
#zTst, aTst:= evalf(zTst), evalf(aTst);
#zTst, aTst:= nearest(zTst), nearest(aTst);
``;
'cp(zTst,aTst)': '%=' %; ``=nearest(rhs(%)); c:=rhs(%):
#'evalhf(zTst^aTst)': '%=' %; ``=nearest(rhs(%)); c:=rhs(%):
'p(zTst,aTst)': '%=' %; ``=nearest(rhs(%)); m:=rhs(%):
#`error absolute` = nearest(m - c);
`error relative` = nearest(abs(1 - c/m));
k*DBL_EPSILON = rhs(%):
``=evalf[3](solve(%, k)) * 'DBL_EPSILON';

zTst, aTst := 56.0 + 2.0 I DBL_EPSILON, 156.0 + 2.0 I DBL_EPSILON

cp(zTst, aTst) = 0.521593550809776937 10273 + 0.157767491846524222 10259 I
=  $\frac{8684521838308383}{9007199254740992} \text{pow}(2, 906) + \frac{7393856401829291}{9007199254740992} I \text{pow}(2, 858)$ 

p(zTst, aTst) = 0.521593550809776928 10273 + 0.157767491846524222 10259 I
=  $\frac{8684521838308383}{9007199254740992} \text{pow}(2, 906) + \frac{7393856401829291}{9007199254740992} I \text{pow}(2, 858)$ 

error relative = 0
= 0.

```

Here we have even an exact result. And using MS?

```

> (5.2159355080984006e+272, 1.5776749184654332e+258):
[%]: VC=op(1,%) + op(2,%)*I;
``=nearest(rhs(%)); vc:=rhs(%):
#`error absolute` = nearest(m - vc);
`error relative` = nearest(abs(1 - vc/m));
k*DBL_EPSILON = rhs(%):

```

```
``=evalf[3](solve(% , k)) * 'DBL_EPSILON';
```

$$\begin{aligned} VC &= 0.52159355080984006 \cdot 10^{273} + 0.15776749184654332 \cdot 10^{259} \cdot I \\ &= \frac{4342260919154717}{4503599627370496} \text{pow}(2, 906) + \frac{3696928200915093}{4503599627370496} \cdot I \text{pow}(2, 858) \\ \text{error relative} &= \frac{4794092314627155}{9007199254740992} \text{pow}(2, -42) \\ &= 545 \cdot \text{DBL_EPSILON} \end{aligned}$$

That means: one would lose the last 2 - 3 possible decimals using VC here.

One way to define power functions is to use $e^{\ln(z) a} = z^a$. Which is quite critical if reading stuff about it, the example shows large relative errors (it is adopted from the Real case, we are extremely close, and I just took an educational example for that).

And testing on more values it seems, that MS does just that in the complex case: treating powers via the (mathematical correct) formula, which numerical is not stable.

While Maple's solution is quite correct, the implementation both in evalhf and using the Watcom compiler seems to be much more careful (ok, this is not a systematic test ... and one should be aware that evalhf does not cover all needs, like 2^{1024} or $\text{abs}(\text{large complex})$...).