FIXED POINT, FLOATING POINT AND POSITS

GÉRARD MEURANT*

1. Introduction. We would like to experiment with Krylov methods for solving linear systems with low accuracy arithmetic and also with variable precision. Another point to be studied is the influence of the rounding mode.

To be able to do this within Matlab we developed classes to do fixed point arithmetic, floating point arithmetic with variable precision and posits. Posits were introduced some time ago by J. Gustafsson [2, 3] as a possible replacement for IEEE floating point arithmetic [5, 6].

2. The `fixp` class. This class implements fixed point arithmetic. The numbers are defined as

$$x = s\,(\,I.F\,),$$

where s is the sign, I and F are binary numbers. The length of the fractional part (or significand) F is *nbits*. The length of I can grow as needed.

A number is coded as a structure with fields,

$$'sign', sign, 'I', I, 'F', F, 'float', x, 'nbits', nbits,$$

where *sig* is 0 (resp. 1) for positive (resp. negative) numbers, I and F are binary numbers that are arrays with entries 0 or 1 and x is the double precision floating point value of the number.

A fixed point number (or matrix) is created by `f = fixp(x,nbits)`, where `x` is a scalar or matrix. The rounding mode is initialized by the function `init_round(rounding)` where `rounding` is an integer between 1 and 6, representing respectively rounding to nearest with ties to even (default), to $+\infty$, to $-\infty$, to zero, stochastic rounding with a probability proportional to the distance to the two closest integers and stochastic rounding to $+\infty$ or $-\infty$ with equal probability.

We implemented the four basic arithmetic operations $+, -, *, /$, so we can write expressions like `x = a * b + c` where `a`, `b`, `c` and `x` are fixed point numbers. We also implemented the \ operator.

Some elementary functions are available, `sqrt, log, log10, exp, sin, cos, tan, cot, asin, acos, atan, acot`. Some of these functions use algorithms used in the C math library *fdlibm* developed at Sun Microsystems as well as algorithms from the book by Cody and Waite [1].

One can construct matrices of fixed point numbers and use the functions `diag, tril, triu, trace, lu, inv`. The `lu` function is a straightforward implementation of LU factorization with partial pivoting for dense matrices. It is used by the \ operator and `inv`.

Of course, the problem of fixed point arithmetic is the limited range of the numbers that can be represented. It was used in the early days of digital computers but the algorithm designers had to be very careful with the scaling of their problems. For instance, if we take `nbits`=16, the smallest number is only $2^{-16} = 1.5259\ 10^{-5}$.

*(`gerard.meurant@gmail.com`) started in Paris, April 2020, version June 9, 2020

If we take 200 random numbers, convert them to fixed point with `nbits`=16 and compute relative difference between the given numbers and the double precision value of the corresponding fixed point numbers we obtain Figure 2.1. The random numbers were between 3 and -3.

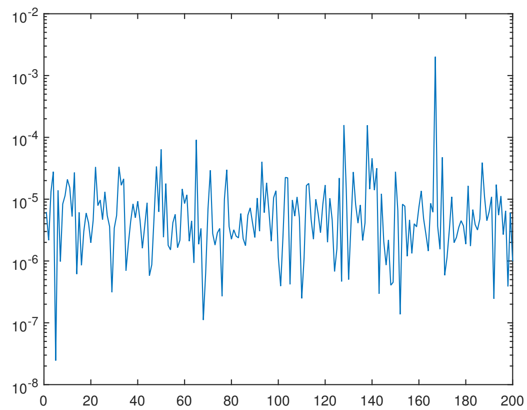

FIG. 2.1. **fixp**, *relative difference, nbits = 16, random numbers in [-3, 3]*

The result is as good as we can hope. But, if we take random numbers multiplied by 10^{-4}, w obtain Figure 2.2 with large relative differences.

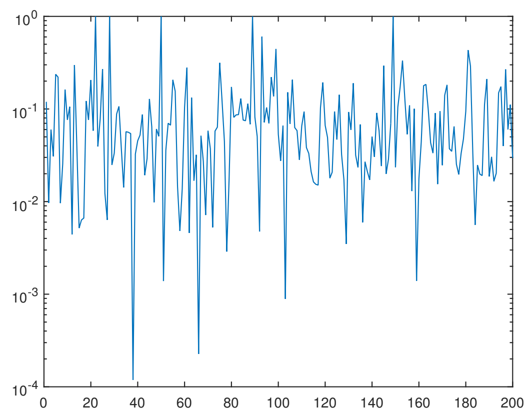

FIG. 2.2. **fixp**, *relative difference, nbits = 16, random numbers in* $10^{-4} \times [-3, 3]$

Let us now multiply two sets of random numbers converted to fixed point and compare to the result of the double precision multiplication. With numbers in the range $[-3, 3]$ we obtain Figure 2.3. But, if the numbers of one of the sets are in $10^{-4} \times [-3, 3]$ we obtain large relative differences as it can be seen in Figure 2.4.

Hence, as it is known, in problems with data of different magnitudes, it is difficult to use fixed point arithmetic.

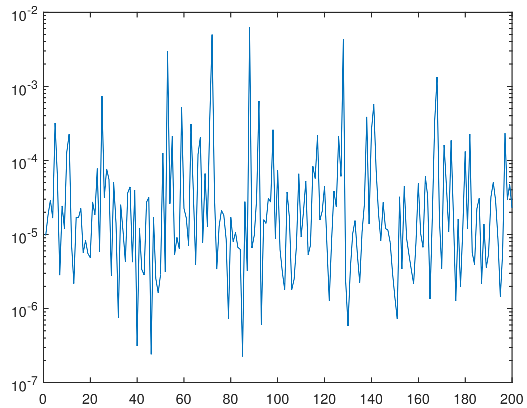

Fig. 2.3. **fixp**, *multiplication relative difference, nbits = 16, random numbers in* $[-3, 3]$

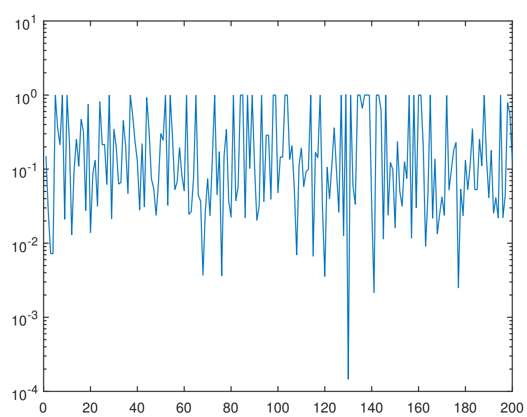

Fig. 2.4. **fixp**, *multiplication relative difference, nbits = 16, x random numbers in* $[-3, 3]$*, y random numbers in* $10^{-4} \times [-3, 3]$

The functions available in class `fixp` are described in Tables 2.1 and 2.2.

TABLE 2.1
Functions available in the class `fixp`

name	
abs	absolute value of a binary fixed point number
acos	componentwise inverse cosine
acos_binf	inverse cosine function
acot	componentwise inverse tangent
acot_binf	inverse cotangent function
add_binf	addition of two fixed point binary numbers
add_binfm	addition of two matrices of binary fixed point numbers
asin	componentwise inverse sine
asin_binf	inverse sine function
atan	componentwise inverse tangent
atan_binf	inverse tangent function
bin2frac	converts the input array to a fractional part
binary	print the fields of a fixed point number
binf2dec	converts a fixed point binary number to a float
binf2decm	binary fixed point to double matrix
binf_inv_Newton	computation of binary fixed point 1/d by Newton iteration
ceil_binf	ceil for a binary fixed point number
cos	componentwise cosine
cos_binf	cos function
cot	componentwise cotangent
cot_binf	cotangent function
ctranspose	transpose of a (real) binary fixed point matrix
dec2binf	converts a double float to binary fixed point
dec2binfm	double to binary fixed point matrix
diag	diagonal function for a binary fixed point matrix or vector
disp	displays the binary fixed point as a double
display	for fixp
div_binf	division of binary fixed point numbers diva / divb
div_binfm	componentwise division of two matrices
div_binfms	division of a matrix by a scalar
dot_binf	dot product of two binary fixed point vectors
double	double precision value of binary fixed point bin
exp	componentwise exponential
exp_binf	exponential
find_min_max	find the first and last significand bits in bin
fix_binf	fix for binary fixed point numbers
fixp	constructor for the class fixp, binary fixed point arithmetic
float2binfb	conversion of a float (double) to fixed point binary
floor_binf	floor for a binary fixed point number
inv	inverse of a binary fixed point matrix
iszero_binf	returns true (1) if the fixed point binary number is zero
ldivide	binb . bina
log	componentwise natural logarithm
log10	componentwise base 10 logarithm
log_binf	natural logarithm
lu	triangular factorization, fixed point numbers
lu_solver_binf	linear solve for binary fixed point
mat_prod_binf	matrix-matrix product
minus	subtraction of two binary fixed point numbers or matrices
minus_binf	subtraction of two fixed point binary numbers, bina - binb
minus_binfm	subtraction of two matrices of binary fixed point numbers
mldivide	division of two binary fixed point numbers or matrices
mpower	bina to the power p for fixed point numbers
mrdivide	division of two binary fixed point numbers or matrices
mtimes	product of two binary fixed point numbers or matrices

Table 2.2
Functions available in the class `fixp` *(continued)*

name	
mul_binf	product of two fixed point numbers
mul_binfm	componentwise multiplication of two matrices
mul_binfo	outer product of two vectors
mul_binfsm	componentwise multiplication of a scalar and a matrix
norm	Frobenius norm of a binary fixed point matrix
plus	addition of two binary fixed point numbers or matrices
pow2	power of 2 of a number
power	bina to the power p for fixed point numbers
printfix	print the fields of binary fixed point
prod	product of vector or matrix binary fixed point numbers
rdivide	componentwise division of two binary fixed point numbers or matrices
round2int	round the binary fixed point number
sin	componentwise sine
sin_binf	sine function
sqrt	componentwise square root
sqrt_binf	square root of a binary fixed point number
subsasgn	for binary fixed point
subsref	for binary fixed point
sum	sum of vector or matrix binary fixed point numbers
tan	componentwise tangent
tan_binf	tangent function
times	componentwise product of two binary fixed point numbers or matrices
trace	trace of a binary fixed point matrix
tril	lower triangular part of a binary fixed point matrix
triu	upper triangular part of a binary fixed point matrix
uminus	change signs of bina
uplus	do not change signs of bina

3. The `floatp` class. This class implements floating point arithmetic with variable precision. The numbers are defined as

$$x = s\,(\,I.F\,)2^E,$$

where s is the sign, I and F are binary numbers and E is the exponent. The length of the fractional part (or significand) F is *nbits*. We normalized the numbers, so I is always equal to 1 except if $x = 0$.

A number is coded as a structure with fields,

$$'sign', sign, 'I', I, 'F', F, 'E', E, 'float', x, 'nbits', nbits,$$

where *sig* is 0 (resp. 1) for positive (resp. negative) numbers, I and F are binary numbers that are arrays with entries 0 or 1 with I equal to 1 or 0 because we normalize the numbers. Generally, I is known as the *hidden bit* and, in IEEE arithmetic, it is not stored. But, here we explicitly store it because this makes the coding easier. The exponent E is the double precision value of a signed integer and x is the double precision floating point value of the number. Representing the exponent in this way simplifies the coding. The values of the exponent can also be limited to simulate a given number of bits. Since the inputs of our conversion functions are double precision IEEE numbers, we cannot represent numbers larger than 10^{308}, but we can vary the number of bits in the fractional part of our floating point numbers. The smallest positive representable number is $2^{-1074} = 4.9407\ 10^{-324}$ which is the smallest IEEE double precision subnormal number.

A floating point number (or matrix) is created by `f = floatp(x,nbits)`, where `x` is a scalar or matrix.

We implemented the same functions as before that is, `sqrt, log, log10, exp, sin, cos, tan, cot, asin, acos, atan, acot, diag, tril, triu, trace, lu, inv`.

We also offer the possibility to change the rounding mode. The rounding mode is initialized by the function `f_d_init_round(rounding)` as for the class `fixp`. Even though the exponent is stored as a double, the function `f_d_init_bits_expo(n)` simulates the limitation of the exponent to n bits. When the computed exponent becomes larger than $e = 2^{n-1} - 1$ the number becomes infinite and if the exponent is smaller than $-(e - 1)$ the number is flushed to zero. If $n = 0$, which is the default, the exponent is not limited. Using this function, we can, for instance, simulate half precision arithmetic `fp16`. We have to use `f_d_init_bits_expo(5)` and *nbits*=10. With `f_d_init_bits_expo(8)` and *nbits*=7, we can simulate `bfloat16` arithmetic.

The functions available in class `floatp` are described in Tables 3.1 and 3.2.

Table 3.1
Functions available in the class `floatp`

name	
abs	absolute value of a binary floating point number
acos	componentwise inverse cosine
acos_binfl	inverse cosine function
acot	componentwise inverse tangent
acot_binfl	inverse cotangent function
add_binf	addition of two fixed point binary numbers
add_binfl	addition of two binary floating point numbers
add_binflm	addition of two matrices of binary floating point numbers
asin	componentwise inverse sine
asin_binfl	inverse sine function
atan	componentwise inverse tangent
atan_binfl	inverse tangent function
bin2frac	converts the input array to a fractional part
binary	outputs the fields of a floating point number
binfl2dec	binary floating point to double
binfl2decm	binary floating point to double matrix
binfl_inv_Newton	computation of binary fixed point 1/d by Newton iteration
ceil	ceil for a binary floating point number
conv_binfl	conversion to a floating point number with a different value of nbits
cos	componentwise cosine
cos_binfl	cos function
cot	componentwise cotangent
cot_binfl	cotangent function
ctranspose	transpose of a (real) binary floating point matrix
diag	diagonal function for a binary floating point matrix or vector
disp	displays the binary floating point as a double
display	display for a binary floating point number
div_binfl	division of binary floating point numbers diva / divb
div_binflm	componentwise division of two matrices of binary floating point numbers
div_binflms	division of a matrix by a scalar
dot_binfl	dot product of two binary floating point vectors
double	double precision value of a binary floating point
exp	componentwise exponential
exp_binfl	exponential of a binary floating point number
find_min_max	finds the first and last significand bits
fix	fix for binary floating point numbers
floatp	constructor for the class floatp, binary floating point arithmetic
floatp2quire	converts a floating point number to a quire structure
floor	floor for a binary floating point number
inv	inverse of a binary floating point matrix
iszero_binf	returns true (1) if the floating point binary number is zero
ldivide	binb . bina
log	componentwise natural logarithm
log10	componentwise base 10 logarithm
log_binfl	natural logarithm of a floating point number
lu	triangular factorization
lu_solver_binfl	linear solve for binary floating point
mat_prod_binfl	floating point matrix-matrix product
minus	subtraction of two binary floating point numbers or matrices
minus_binf	subtraction of two fixed point binary numbers
minus_binfl	subtraction of two binary floating point numbers
minus_binflm	subtraction of two matrices of binary floating point numbers
mldivide	division of two binary floating point numbers or matrices
mpower	bina to the power p for floating point numbers
mrdivide	division of two binary floating point numbers or matrices
mtimes	product of two binary floating point numbers or matrices

Table 3.2
Functions available in the class `floatp` *(continued)*

name	
mul_binf	product of two fixed point numbers
mul_binfl	multiplication of two binary floating point numbers
mul_binflm	componentwise multiplication of two matrices
mul_binflo	outer product of two vectors
mul_binflsm	scalar-matrix product
norm	Frobenius norm of a binary floating point matrix
plus	addition of two binary floating point numbers or matrices
pow2	power of 2 in a number
power	bina to the power p for floating point numbers
printfloatp	prints the fields of binary floating point
prod	product of vector or matrix binary floating point numbers
rdivide	componentwise division of two binary floating point numbers or matrices
right_shift_binfl	shift to the right by k places
round2int	rounds the binary floating point number
sin	componentwise sine
sin_binfl	sine function
sqrt	componentwise square root
sqrt_binfl	square root of a binary floating point number
subsasgn	for binary floating point
subsref	for binary floating point
sum	sum of vector or matrix binary floating point numbers
tan	componentwise tangent
tan_binfl	tangent function
times	componentwise product of two binary floating point numbers or matrices
trace	trace of a binary floating point matrix
tril	lower triangular part of a binary floating point matrix
triu	upper triangular part of a binary floating point matrix
uminus	change signs
uplus	does not change signs

Since there is a large overhead due to the class operator overloading, we also give access in the directory `f_d_floatp` to the following functions that operate only on structures and not on objects of the class `floatp`. Moreover, some of these functions are used by the class.

Table 3.3
Functions available in `f_d_floatp`

name	
d_float_dec2floatp	converts a double x to binary floating point format
f_d_abs	absolute value of a binary floating point number
f_d_add	addition of two binary floating point numbers
f_d_add_bin_carry	addition of two unsigned binary strings with a carry in
f_d_add_bin_one_carry	add 1 to a binary number
f_d_add_binfp	addition of two fixed point binary numbers
f_d_add_floatp2quire	addition of a floatp and a quire towards a quire
f_d_add_quire	addition of two quires
f_d_addbin	addition of two binary strings
f_d_addm	addition of two matrices of binary floating point numbers
f_d_bin2dec	converts the binary input array to a decimal (double) number
f_d_bin2frac	converts the input array to a double fractional part
f_d_bin2str	binary to string
f_d_binary	prints the fields of a floating point structure
f_d_dec2bin	converts a decimal to binary
f_d_dec2floatp	double to binary floating point matrix
f_d_dec2quire	conversion of a float (double) to a quire
f_d_diag	diagonal function for a binary floating point matrix or vector
f_d_div	division of binary floating point numbers
f_d_divm	componentwise division of two matrices
f_d_divms	division of a matrix by a scalar
f_d_dot	dot product of two binary floating point vectors
f_d_dot_prod	dot product using a quire
f_d_double	decimal value of a floating point number
f_d_eye	identity matrix of binary floating point numbers
f_d_find_min_max	find the first and last significand bits
f_d_floatp2dec	binary floating point to double matrix
f_d_floatp2quire	converts a floatp structure to a quire
f_d_frac2bin	converts a fractional part to binary
f_d_init_bits_expo	initializes the number of bits of the exponents
f_d_init_floatp	construction a floatp structure from its elements
f_d_init_round	initializes the rounding mode
f_d_inv	inverse of a binary floating point matrix
f_d_inv_Newton	computation of binary floating point 1/d by Newton iteration
f_d_isge_bin	compares two binary numbers
f_d_iszero	returns true (1) if the floating point binary number is zero
f_d_lu	triangular factorization
f_d_lu_solver	linear solver for binary floating point
f_d_mat_prod	floating point matrix-matrix product
f_d_mat_prod_b	floating point matrix-matrix product
f_d_minus	subtraction of two binary floating point numbers
f_d_minus_bin	subtraction of two binary strings
f_d_minus_binf	subtraction of two fixed point binary numbers
f_d_minus_binfp	subtraction of two fixed point binary numbers
f_d_minus_quire	subtraction of two quires
f_d_minusm	subtraction of two matrices of binary floating point numbers

Using these functions makes the programming less straightforward but more efficient in terms of computing time. For instance, to code `a*b+c*d` we have to write `f_d_add_binfl( f_d_mul_binfl(a,b), f_d_mul_binfl(c,d) )`.

TABLE 3.4
Functions available in `f_d_floatp` *(continued)*

name	
f_d_mul	multiplication of two binary floating point numbers
f_d_mul_binf	product of two fixed point numbers
f_d_mulm	componentwise multiplication of two matrices
f_d_mulo	outer product of two vectors
f_d_mulsm	scalar-matrix product
f_d_printfloatp	prints the fields of a binary floating point
f_d_prod	product of vector or matrix binary floating point numbers
f_d_quire2dec	converts a quire to decimal
f_d_quire2floatp	converts a quire structure to a floatp structure
f_d_right_shift	shift to the right by k places
f_d_round_bin	round the binary number
f_d_sqrt	square root of a binary floating point number
f_d_sum	sum of vector or matrix binary floating point numbers
f_d_tril	lower triangular part of a binary floating point matrix
f_d_triu	upper triangular part of a binary floating point matrix
fix_binf2dec	converts a fixed point binary number (structure) to a float (double)
fix_dec2binf	converts a double float to binary fixed point
fix_dec2binfm	double to binary fixed point matrix
fix_float2binfb	conversion of a float (double) to fixed point binary
floatp_eye	identity matrix of binary floating point numbers
print_round_mode	prints the rounding mode

If we convert random numbers of order 1 to `floatp`, we obtain Figure 3.1 which is not much different from Figure 2.1. But, since we now have an exponent for our numbers, the result for random numbers in $10^{-4} \times [-3, 3]$ (see Figure 3.2) is much better than with `fixp`. The multiplication of two sets of random numbers gives what can be expected; see Figure 3.3.

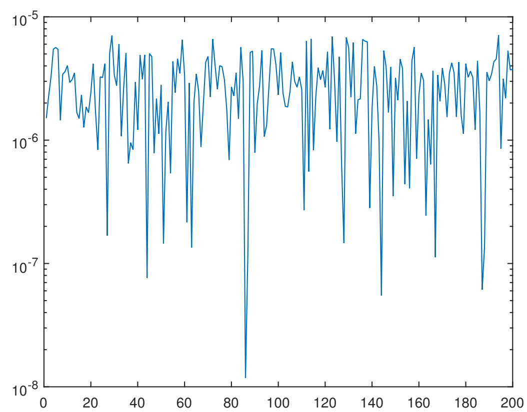

FIG. 3.1. `floatp`, *relative difference, nbits = 16, random numbers in [-3, 3]*

As an exemple of the use of the functions in `f_d_floatp`, the following code implements the Conjugate Gradient (CG) method for solving a linear system $Ax = b$ for a symmetric positive definite matrix A.

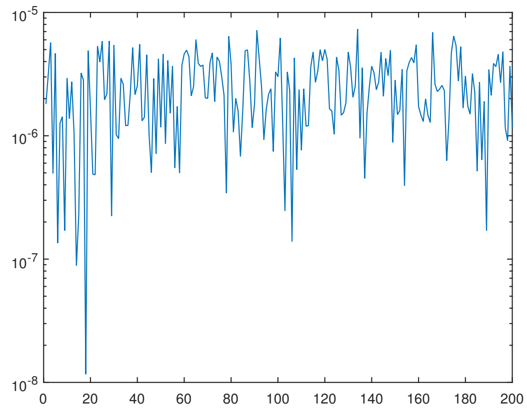

Fig. 3.2. **floatp**, *relative difference, nbits = 16, random numbers in* $10^{-4} \times [-3, 3]$

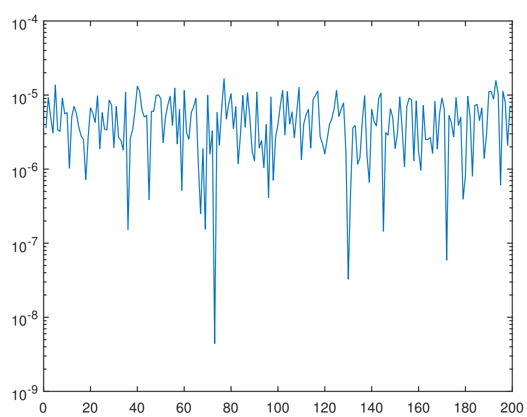

Fig. 3.3. **floatp**, *multiplication relative difference, nbits = 16, x random numbers in* $[-3, 3]$, *y random numbers in* $10^{-4} \times [-3, 3]$

```
function [x,resn,resnt,errn,errnl2,nit]=cg_f_d_floatp_A(A,b,x0,xex,...
   epsi,nitmax,nbits,rounding,expo);
%CG_F_D_FLOATP_A CG for a matrix A of floating point binary numbers
‘% uses functions of f_d_floatp
% A = matrix
% b = rhs, x0 initial vector
% xex = "exact" solution
% epsi = stopping criterion threshold
% nitmax = number of iterations
% nbits = size of the significand (fractional part)
% rounding = rounding mode (1,...,6)
% expo = number of bits of the exponent, if = 0 no limitation
%
% the inputs are double precision numbers
%
rng('default') % for stochastic rounding
if nargin < 9
 expo = 0;
end % if
f_d_init_bits_expo(expo);
if nargin < 8
 rounding = 1;
end % if
f_d_init_round(rounding); % initialize the rounding mode
% convert inputs to binary floating point
bA = f_d_dec2floatp(A,nbits);
bb = f_d_dec2floatp(b,nbits);
bx = f_d_dec2floatp(x0,nbits);
xec = f_d_dec2floatp(xex,nbits); % "exact solution"
bAx = f_d_mat_prod(A,bA,bx);
r = f_d_minusm(bb,bAx); % initial residual vector
errn = zeros(1,nitmax+1); % double precision values
errnl2 = zeros(1,nitmax+1);
resn = zeros(1,nitmax+1);
resnt = zeros(1,nitmax+1);
err = f_d_minusm(bx,xec); % initial error
bAe = f_d_mat_prod(A,bA,err);
er = f_d_dot(err,bAe);
errn(1) = sqrt(f_d_floatp2dec(er));
er = f_d_dot(err,err);
errnl2(1) = sqrt(f_d_floatp2dec(er));
res = f_d_dot(r,r);
resn(1) = sqrt(f_d_floatp2dec(res));
resnt(1) = resn(1);
nr = resn(1);
p = r;
rtr = res;
dbb = f_d_dot(bb,bb);
nb = sqrt(f_d_floatp2dec(dbb));
```

```
nit = 0;
%
while (nit < nitmax) && (nr > epsi * nb)
 nit = nit + 1;
 Ap = f_d_mat_prod(A,bA,p); % Ap = A * p
 pAp = f_d_dot(p,Ap); % pAp = p' * Ap
 alp = f_d_div(rtr,pAp); % alp = rtr / pAp
 bx = f_d_addm(bx,f_d_mulsm(alp,p)); % x = x + alp * p
 r = f_d_minusm(r,f_d_mulsm(alp,Ap)); % r = r - alp * Ap
 rk = f_d_dot(r,r); % rk = r' * r
%
 err = f_d_minusm(bx,xec); % error
 bAe = f_d_mat_prod(A,bA,err);
 er = f_d_dot(err,bAe);
 errn(nit+1) = sqrt(f_d_floatp2dec(er)); % norm of the error
 er = f_d_dot(err,err);
 errnl2(nit+1) = sqrt(f_d_floatp2dec(er));
%
 res = f_d_dot(r,r);
 resn(nit+1) = sqrt(f_d_floatp2dec(res)); % norm of the computed residual
 bAx = f_d_mat_prod(A,bA,bx);
 rt = f_d_minusm(bb,bAx);
 res = f_d_dot(rt,rt);
 resnt(nit+1) = sqrt(f_d_floatp2dec(res)); % norm of the true residual
%
 bet = f_d_div(rk,rtr); % bet = rk / rtr
 rtr = rk;
 p = f_d_addm(r,f_d_mulsm(bet,p)); % p = r + bet * p
end % while
x = f_d_floatp2dec(bx);
```

4. The `posit` class. As we said above posits were proposed as an alternative to the IEEE floating point arithmetic standard. A posit number depends on two given numbers *nbits*, the length of the binary number and *es*, the number of bits for the exponent. It is defined as

$$x = s\,(\,I.F\,)u^k\,2^m,$$

u is equal to $2^{2^{es}}$, the binary number k is known as the regime and the binary number m of length es is the exponent. The lengthes of F and k are not determined a priori. The binary number k encodes a signed integer in the following way. A positive integer p is coded as $1\cdots10$ with $p+1$ leading ones, a zero integer is 10 and a negative integer $-p$ is $0\cdots01$ with p leading zeros. The total exponent of 2 is $k\,2^{es}+m$. The length of the regime depends on the value of x and, therefore, the length of F is what is left that is, *nbits* minus a number which is the length of the regime plus $es+1$ (1 for the sign) since the hidden bit I is not stored.

In our class a posit number is coded as a structure with fields,

$$'sign', sign, 'regime', reg,\ 'exponent', expo,\ 'mantissa', mantiss, 'nbits', nbits,\\ 'es', es,\ 'float', x.$$

The basic operations $+,-,*$ are done by computing the total exponent and using what we have already done for `fixp` and `floatp`, even though the fractional part does not have always the same length. The division is done with a multiplication with the inverse of the divisor obtained by Newton iteration.

A posit number (or matrix) is created by `p = posit(x,nbits)`, where `x` is a scalar or matrix. We use the posit standard corresponding to *nbits*. Note that *nbits* does not have the same signification as for `fixp` or `floatp`. Here, it is the total number of bits of the posit number whence before it was the number of bits in the mantissa. The rounding mode is initialized by the function `p_init_round(rounding)`.

The functions available in class `posit` are described in Tables 4.1 and 4.2.

Table 4.1
Functions available in the class `posit`

name	
abs	absolute value of a posit
acos	componentwise inverse cosine
acos_posit	inverse cosine function for a posit number
acot	componentwise inverse tangent
acot_posit	inverse cotangent function for a posit number
add_binfp	addition of two fixed point binary numbers
add_posit	addition of two posit numbers
add_posit_quire	addition of a posit and a quire towards a quire
add_positm	addition of two posit matrices
asin	componentwise inverse sine
asin_posit	inverse sine function for a posit number
atan	componentwise inverse tangent
atan_posit	inverse tangent function for a posit number
axpyqq	axpy palp * pa + pb with quires
binary	prints the fields of a posit as binary digits
ceil	ceil for a posit number
cos	componentwise cosine
cos_posit	cos function for a posit number
cot	componentwise cotangent
cot_posit	cotangent function for a posit number
ctranspose	transpose of a (real) posit matrix
diag	diagonal function for a posit matrix or vector
disp	displays a posit as a double
display	displays the double value of a posit
div_posit	division of posits
div_positm	componentwise division
div_positms	division of a matrix by a scalar
dot_posit	dot product of two posit vectors
dot_prod_posit	dot product of two posit vectors using a quire
dot_prod_positq	dot product of two posit vectors using a quire
double	double precision value of a posit
exp	componentwise exponential
exp_posit	exponential of a posit number
fix	fix for posit numbers
floor	floor for a posit number
inv	inverse of a posit matrix
iszero_binfp	returns true (1) if the floating point binary number is zero
iszero_posit	returns true (1) if the posit number is zero
ldivide	binb . bina
log	componentwise natural logarithm
log10	componentwise base 10 logarithm
log_posit	natural logarithm of a posit number
lu	triangular factorization
lu_solver_posit	linear solver for posit linear systems
mat_prod_posit	matrix-matrix product for posits
minus	subtraction of two posit numbers or matrices
minus_binfp	subtraction of two fixed point binary numbers
minus_posit	subtraction of two posits
minus_positm	subtraction of two posit matrices
mldivide	division of two posit numbers or matrices
mpower	bina to the power p for posit numbers
mrdivide	division of two posit numbers or matrices
mtimes	product of two posit numbers or matrices

TABLE 4.2
Functions available in the class `posit` *(continued)*

name	
mul_binfp	product of two mantissas of posits
mul_posit	multiplication of two posit numbers
mul_positm	componentwise multiplication of two posit matrices
mul_posito	outer product of two vectors of posit numbers
norm	Frobenius norm of a binary floating point matrix
plus	addition of two posit numbers or matrices
posit	constructor for the class posit, posit arithmetic
posit2bin	converts a posit to a binary string
posit2dec	converts posit to decimal (double floating point)
posit2decm	converts a posit matrix to decimal (double floating point)
posit2floatp	converts a posit to a floatp structure
posit2quire	converts a posit to a quire structure
posit2struct	converts a posit to a structure
posit_inv_Newton	computation of posit 1/d by Newton iteration
power	bina to the power p for posit numbers
printposit	prints the fields of a posit
prod	product of vector or matrix of posit numbers
rdivide	componentwise division of two posit numbers or matrices
right_shift_binfp	shift to the right by k places
round2int	round the posit
sin	componentwise sine
sin_posit	sine function for a posit number
sqrt	square root of a posit number or matrix
sqrt_posit	square root of a posit number
subsasgn	for posits
subsref	for posits
sum	sum of vector or matrix posit numbers
sum_abs_posit	sum of the absolute values of the components of a vector using a quire
sum_posit	sum of the components of a posit vector using a quire
tan	componentwise tangent
tan_posit	tangent function for a posit number
times	componentwise product of two posit numbers or matrices
trace	trace of a posit matrix
tril	lower triangular part of a posit matrix
triu	upper triangular part of a posit matrix
uminus	change signs
uplus	do not change signs

As before there is a large overhead due to the class operator overloading and we give access to the following functions in the directory `p_posit` that operate only on structures. Moreover, some of these functions are used by the class.

TABLE 4.3
Functions available in `p_posit`

name	
p_abs	absolute value of a posit
p_add_bin_carry	addition of two unsigned binary strings with a carry in
p_add_bin_one_carry	add 1 to a binary number
p_add_binfp	addition of two fixed point binary numbers
p_add_posit	addition of two posit structures
p_add_positm	addition of two posit matrices
p_addbin	addition of two binary strings
p_addbinone	add 1 to a binary number
p_bin2dec	converts the input array of 0's and 1's to a decimal number
p_bin2frac	converts the input array to a double fractional part
p_bin2str	binary to string
p_binary	prints the fields of a posit structure
p_binshift	shift a bit string by n positions, left (right) if positive (negative)
p_dec2bin	converts a decimal integer to binary
p_dec2posit	converts a double (matrix) x to a posit structure
p_diag	diagonal function for a posit matrix or vector structure
p_div_posit	division of posits
p_div_positm	componentwise division
p_div_positms	division of a matrix by a scalar
p_dot_posit	dot product of two posit vectors
p_dot_prod_posit	dot product of two posit vector structures using a quire
p_dot_prod_positq	dot product of two posit vectors using a quire
p_double	decimal value of a posit
p_eye	identity matrix of posit numbers
p_find_regime_expo	finds the powers of 2 for a posit
p_frac2bin	converts a fractional part to binary
p_init_round	initializes the rounding mode
p_inv	inverse of a binary floating point matrix
p_isdiv	true if x is divisible by p
p_isge_bin	comparison of binary numbers
p_iszero_posit	returns true (1) if the posit number is zero
p_lu	triangular factorization
p_lu_solver	linear solver for posits
p_mat_prod_posit	posit matrix-matrix product
p_mat_prod_posit_b	posit matrix-matrix product
p_minus_bin	subtraction of two binary strings
p_minus_binfp	subtraction of two fixed point binary numbers
p_minus_posit	subtraction of two posits
p_minus_positm	subtraction of two posit matrices

"Standard" values for *nbits* and *es* are $(8,0)$, $(16,1)$, $(32,2)$, $(64,3)$ and $(128,7)$. We will speak of `posit8`, `posit16`, and so on. The only exceptional value for a posit is the binary number $10\cdots0$ which we denote as *Inf*.

Posits were designed to have a good accuracy for numbers around 1. Let us first illustrate that with `posit8`. Figure 4.1 shows the double precision IEEE numbers in blue and the corresponding posit numbers in red. The double precision numbers x are logarithmically spaced between 10^{-2} and 10^{2}. The x-axis shows the log_{10} of x, so 0 represent $x = 1$. We see that the numbers around 1 are well represented but when we move away from 1 it becomes worse and worse.

TABLE 4.4
Functions available in `p_posit` *(continued)*

name	
p_mul_binfp	product of two mantissas of posits
p_mul_posit	multiplication of two posit numbers
p_mul_positm	componentwise multiplication of two posit matrices
p_mul_posito	outer product of two vectors of posit numbers
p_mul_positsm	scalar-matrix product for posits
p_posit2bin	converts a posit to a binary string
p_posit2dec	converts a posit scalar or matrix to decimal
p_posit2decm	converts a posit matrix to decimal
p_posit2floatp	converts a posit to a floatp structure
p_posit2quire	converts a posit to a quire structure
p_posit_inv_Newton	computation of posit 1/d by Newton iteration
p_print_round_mode	prints the current rounding mode
p_printposit	prints the fields of a posit structure
p_regrunlength	returns the regime bits run length
p_right_shift_binfp	shift to the right by k places
p_round_bin	round the binary number bin
p_set_posit_env	standard posits
p_setenvposit	sets the posit and quire parameters
p_struct2posit	converts the floating point structure to a posit structure
p_tril	lower triangular part of a binary floating point matrix
p_triu	upper triangular part of a binary floating point matrix
posit_eye	identity matrix of posit numbers
q_add_posit_quire	addition of a posit and a quire towards a quire
q_add_quire	addition of two quires
q_dec2quire	conversion of a float (double) to a quire
q_minus_quire	subtraction of two quires
q_mul_quire	product of two quire structures
q_posit2quire	converts a posit to a quire structure
q_quire2dec	converts a quire to double
q_quire2posit	converts a quire structure to a posit structure
q_set_quire2zero	returns a zero quire

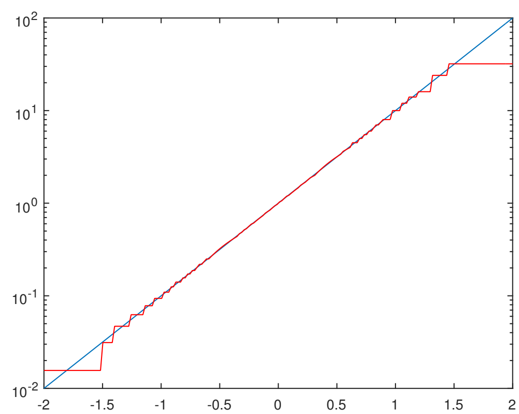

FIG. 4.1. `posit8`, *comparison of* x *and posit*$(x, 8)$

Figure 4.2 shows what happens for `posit16`. The relative difference between x and $posit(x, 16)$ is shown in Figure 4.3.

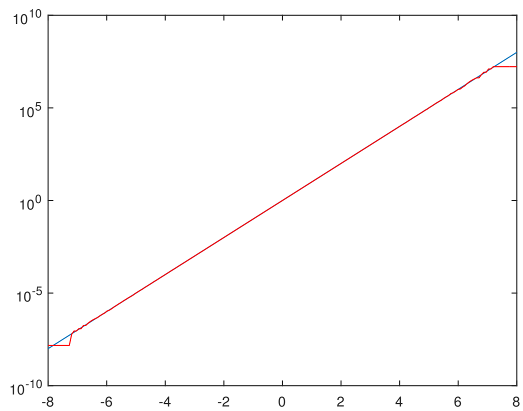

Fig. 4.2. `posit16`*, comparison of* x *and* $posit(x, 16)$

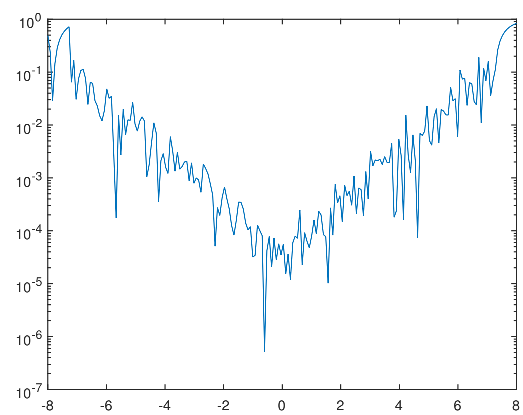

Fig. 4.3. `posit16`*, relative difference between* x *and* $posit(x, 16)$

Figure 4.4 shows the relative differences between the double precision x and its representations with `posit16`, `fp16`, the IEEE half precision format [6] defined in 2008 and `bfloat16`, the half precision format proposed by Google and Intel. We use rounding to nearest. We observe that around $x = 1$ `posit16` gives a better representation than `fp16`. Out of $[10^{-2}, 10^{2}]$ `fp16` yields a better accuracy than `posit16`. For numbers larger than 10^5 `fp16` returns `Inf` because there are not enough bits for the exponent. `bfloat16` gives the worst result in $[10^{-4}, 10^{4}]$ because there is only 7 bits for the mantissa, instead of 10 for `fp16`. But, since 8 bits are available for the exponent, it yields better results outside of $[10^{-4}, 10^{4}]$.

Figure 4.5 displays the relative difference for 200 random numbers in $[-3, 3]$ with `posit16`. If the random numbers are multiplied by 10^4 the relative differences are increasing; see Figure 4.6.

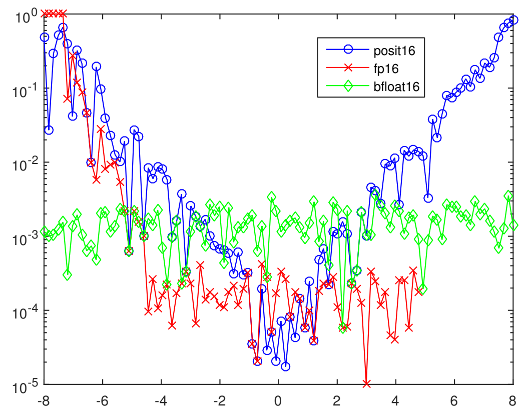

FIG. 4.4. *Relative difference between x and $posit(x, 16)$, $fp16(x)$ and $bfloat16(x)$*

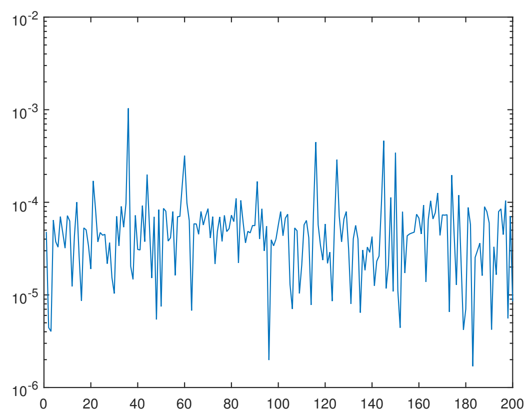

FIG. 4.5. `posit16`*, relative difference for random numbers in* $[-3, 3]$

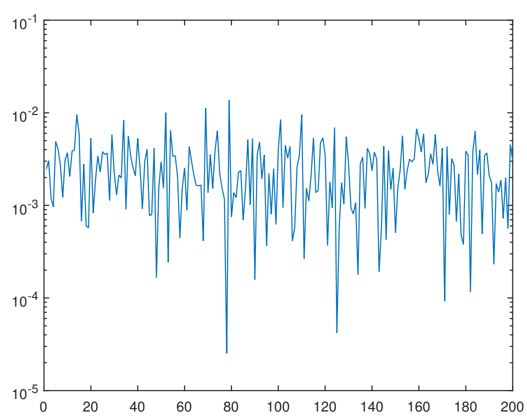

FIG. 4.6. `posit16`*, relative difference for random numbers in* $10^4 \times [-3, 3]$

If we compute with numbers around 1, posits may provide a better accuracy than the IEEE standard. For instance, let us consider $x = 1.1$. In the IEEE half precision format `fp16` we have 10 bits for the mantissa since 5 bits are used for the exponent and one bit for the sign. With `posit16`, we obtain

$$sign = [0],\ regime = [10],\ exponent = [0],\ mantissa = [000110011010].$$

We see that 12 bits are used for the mantissa that is, two more bits than with `fp16`. The relative difference with the exact value is $8.88\ 10^{-5}$. However, if $x = 1.1\ 10^4$, the posit is

$$sign = [0],\ regime = [11111110],\ exponent = [1],\ mantissa = [010110].$$

The regime is using 8 bits and there are only 6 bits left for the mantissa. The decoded regime gives us 6 and u is equal to 4 since es=1. Hence, the multiplying factor is $4^6 \times 2 = 8192$. The mantissa with the hidden bit gives 1.343750. Multiplying the two values we obtain 11008 and a relative difference of $7.28\ 10^{-4}$, ten times larger than for $x = 1.1$.

If we would have taken $x = 1.1\ 10^5$, only 4 bits would have been available for the mantissa since 10 bits are used for the regime. So, we can expect a good representation of real numbers with posits only when the number of bits used for the regime is small. For very large or very small positive numbers almost all the bits are used for the regime and there is no bits left for the mantissa which means that posits can then only represent powers of 2.

Figure 4.7 shows the relative differences between the double precision x and its representations with `posit32` and `fp32`, the IEEE single precision format. With 32 bits, posits are worse roughly out of $[10^{-8}, 10^{8}]$.

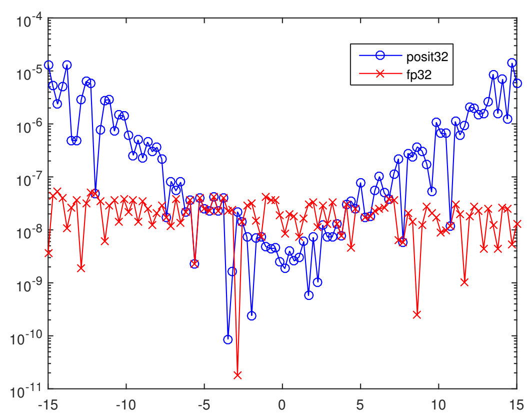

FIG. 4.7. *Relative difference between x and $posit(x, 32)$, $fp32(x)$*

Figure 4.8 shows the relative difference of the result of the multiplication of two sets of 200 random numbers converted to `posit16` with the double precision result. The random numbers were in a small interval around zero. If one of the sets is multiplied by 10^5, the result is much worse as we can see in Figure 4.9.

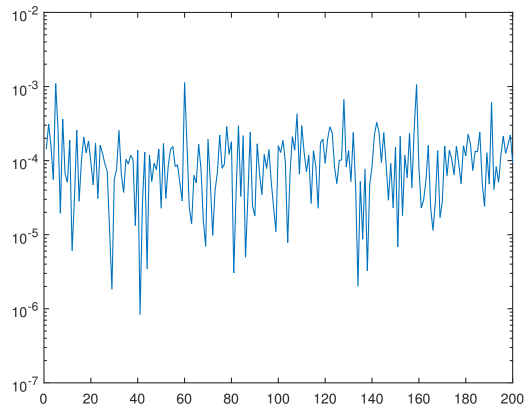

FIG. 4.8. `posit16`, *multiplication relative difference, x and y random numbers in* $[-3, 3]$

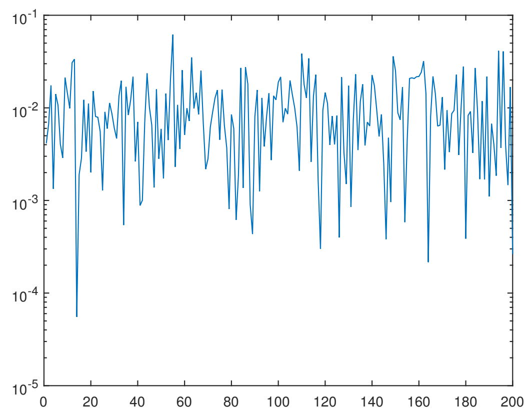

FIG. 4.9. `posit16`, *multiplication relative difference, x random numbers in* $[-3, 3]$*, y random numbers in* $10^5 \times [-3, 3]$

The proposal for posits also includes a *quire*, a long fixed point register to accumulate the results of sums (or differences) without rounding. This idea was proposed earlier by U. Kulisch [9, 8]. If p is a posit and q is the quire we must implement the following operations: $p \to q$, $q \to p$, $p \pm q \to q$ and $(pa * pb) \pm q \to q$. This allows to do a series of sums or differences with only one rounding at the end when the content of the quire is converted to a posit. Let $nq = nbits^2/4 - nbits/2$. The quire has four different binary zones, the sign bit s, C, I and F,

$$q = s\ [C,\ I].\ F.$$

I and F have length nq and C, designed to absorb the overflows of I has length $nc = nbits - 1$. F stores the mantissa. We implemented a `quire` class. Some operations, like the sum of a posit to a quire, are also implemented in the `posit` class using the conversion of the posit to a temporary scratch quire before doing the addition to the quire. This allows to implement the function `dot_prod_posit(pa,pb)` which does the dot product of two posit vectors with only one rounding at the end. Note that the length of the quire is large, 128 bits for $nbits = 16$, 512 bits for $nbits = 32$ and 2048 bits for $nbits = 64$.

Figure 4.10 shows the relative difference with the double precision dot product of 100 random vectors of length 50. Let `px` and `py` be the two posit vectors. The red curve is obtained by computing `px' * py` only with posits and the blue curve corresponds to `dot_prod_posit(px,py)` using the quire which, in most cases, yields a more accurate result. Note that we may loose some accuracy when converting back to posits at the end of the sum.

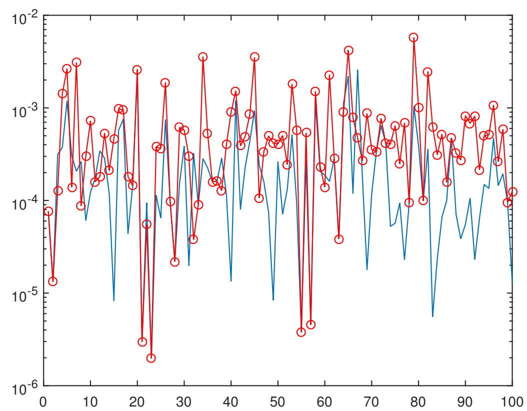

FIG. 4.10. `posit16`, *relative difference for dot products*

Table 4.5 lists the functions in the class `quire`.

TABLE 4.5
Functions available in the class `quire`

name	
add_quire	addition of two quires
disp	displays a quire as a double
display	displays the quire as a double
double	double precision value of a quire
minus	subtraction of two quires
minus_quire	subtraction of two quires
mul_quire	product of two quires
plus	addition of two quires
quire	constructor for the class quire, posit arithmetic
quire2dec	converts a quire to decimal
quire2posit	converts a quire to a posit
set_quire2zero	returns a zero quire
uminus	change signs
uplus	do not change signs

5. Other possibilities. Another possibility to compute with high precision arithmetic is to use the `vpa` function of the Matlab Symbolic Math Toolbox but not everybody has access to this toolbox. As a second argument, one can give `digits`, the number of significant decimal digits wanted. But, this does not allow to simulate low precision arithmetic since, for instance, with `digits`=4, you can have numbers like 0.00003333, the last four "3" being the four significant digits.

For 8 bits or 16 bits floating point arithmetic one can use the classes `fp8` and `fp16` developed by Cleve Moler [10]. Another possibility is to use the `chop` function by N.J. Higham and S. Pranesh [4]. It allows to simulate `fp16` and `bfloat16` as well as choosing the rounding mode. We encapsulated this function in a class named `chop` but, as noted in [4], it is much slower than directly using the function.

Figure 5.1 shows the relative difference of the result of the multiplication of two sets of 200 random numbers converted to `fp16` with the double precision result.

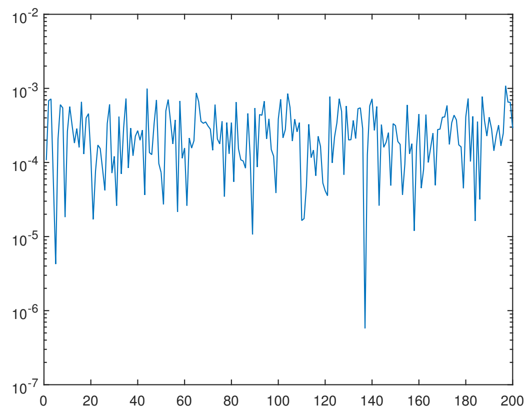

FIG. 5.1. `chop` *for half-precision* `fp16`*, multiplication relative difference, x and y random numbers in* $[-3, 3]$

REFERENCES

[1] W.J. Cody and W. Waite, *Software manual for the elementary functions*, Prentice-Hall, (1980).
[2] J. Gustafson, *Posit Arithmetic*, (2017), *https://posithub.org/docs/Posits4.pdf*.
[3] J. Gustafson and I. Yonemoto, *Beating floating point at its own game: Posit arithmetic*, Supercomput. Front. Innov., v 4, n 2 (2017), pp. 71–86.
[4] N.J. Higham and S. Pranesh, *Simulating low-precision floating point arithmetic*, SIAM J. Sci. Comput., v 41 n 5 (2019), pp. C585-C602.
[5] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*, IEEE Computer Society, New York, (1985).
[6] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*, IEEE Computer Society, New York, (2008).
[7] Intel Corporation, *BFLOAT16—Hardware numerics definition*, white paper, document number 338302-001US, (2018).
[8] R. Kirchner and U.W. Kulisch, *Arithmetic for vector processors*, in Reliability in Computing, R.E. Moore Ed., Elsevier, (1988), pp. 3–41.
[9] U.W. Kulisch and W.L. Miranker, *The arithmetic of the digital computer: a new approach*, SIAM Review, v 28 n 1 (1986), pp. 1–40.
[10] C.B. Moler, *Half Precision16-Bit Floating Point Arithmetic* posted May 8, 2017, http://blogs.mathworks.com/cleve/2017/05/08/half-precision-16-bit-floating-point-arithmetic.