General Interest Seminar

Computing in Arbitrary Precision

Ge Baolai, Western University
SHARCNET | Compute Ontario | Compute Canada

Floating point arithmetic...

Floating point numbers

Example. C code

```
#include <stdio.h>

int main(void)
{
    float a = 0.5, b = 0.1, c = 0.25;

    printf("a=%40.40f\n", a);
    printf("b=%40.40f\n", b);
    printf("c=%40.40f\n", c);

    return 0;
}
```

```
a=0.5000000000000000000000000000000000000000
b=0.1000000014901161193847656250000000000000
c=0.2500000000000000000000000000000000000000
```

Example. Fortran code

```
program rrep
   implicit none
   real :: a = 0.5, b = 0.1, c = 0.25

   print '("a="f45.40)', a
   print '("b="f45.40)', b
   print '("c="f45.40)', c
end program rrep
```

```
a=    0.5000000000000000000000000000000000000000
b=    0.1000000014901161193847656250000000000000
c=    0.2500000000000000000000000000000000000000
```

???

Floating point numbers

Example. Consider a simplified case,

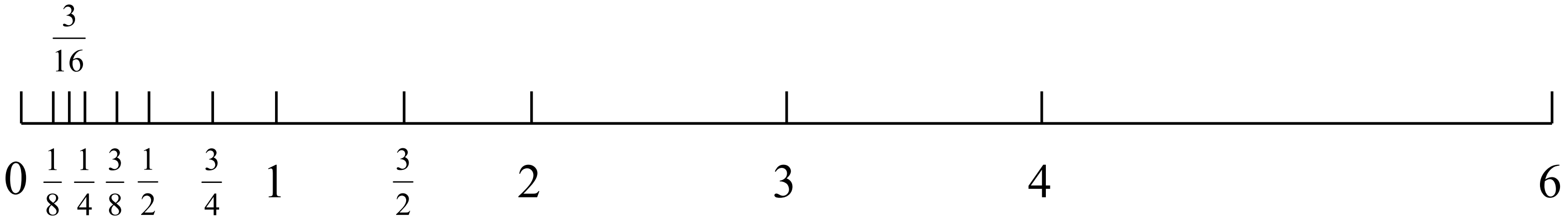
$$\pm d_1 d_2 \times 2^E \quad E \in \{-2, -1, 0, 1, 2, 3\}$$

The following are all the positive numbers:

$$0.10 \times 2^E = \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4\},$$

$$0.11 \times 2^E = \{\frac{3}{16}, \frac{3}{8}, \frac{3}{4}, \frac{3}{2}, 3, 6\}.$$

There are **gaps** that are uneven in size



Floating point numbers

- This indicates that before we do anything, we already have lost the accuracy due to the limit in floating point representation.
- We can imagine, during the computation, further inaccuracy will be introduced due to round off error.
- So people use double precision, quad precision or higher precision in order to obtain the accuracy desired.

Floating-point Representation

When normalized, a number is represented in the form

$$\pm d_1 d_2 \cdots \times b^E$$

On computers, numbers are stored in binary format (*b* = 2).

Let's convert the numbers by hand. First

$$0.5 = 1.0 \times 2^{-1}$$

Exact!

But 0.1 is converted to an infinite sequence

$$0.1 = 0.00011001100110011 0\ldots$$

When converted back to decimal representation, we see, when set float b = 0.1

b = 0.100000001490116120000000000000000000000

Floating point numbers

Example. Something to be aware of, e.g. adding 0.001 thousand times

Code:

```
float a = 0.001, s = 0.0;
for (i=0; i<1000; i++)
    s += a;
printf("Sum of 0.001 1000 time = %10.8f\n",
     s);
```

Result: Sum of 0.001 1000 times = 0.99999070

Facts:

- Summation is not associative

$$(a + b) + c \neq a + (b + c)$$

- In double precision

$$1+0.0000000000000001 = 1$$

See Kahan's ***Summation Formula*** (Theorem) for more accurate algorithm.

Floating point numbers

Example. Common “mistake”

```
if (a == b) { /* This may never happen */
    do something
}
```

Error – Floating point number comparison with equal sign is “risky”, due to round off

- Either the condition is never true or
- Results are system dependent

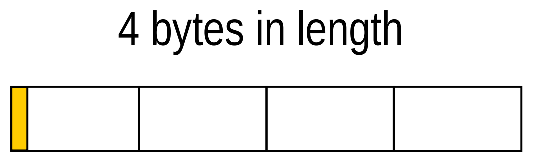
Correction

```
if (fabs(a – b) < tol) { /* Instead, loose the condition $|a - b| < \epsilon$ */
    do something
}
```

See ***“What every computer scientist should know about floating-point arithmetic”*** for more detailed discussions.

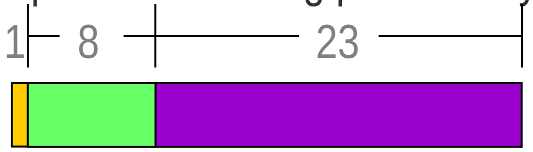
IEEE Floating Point Arithmetic – Storage

Integer – 4 bytes = 32 bits



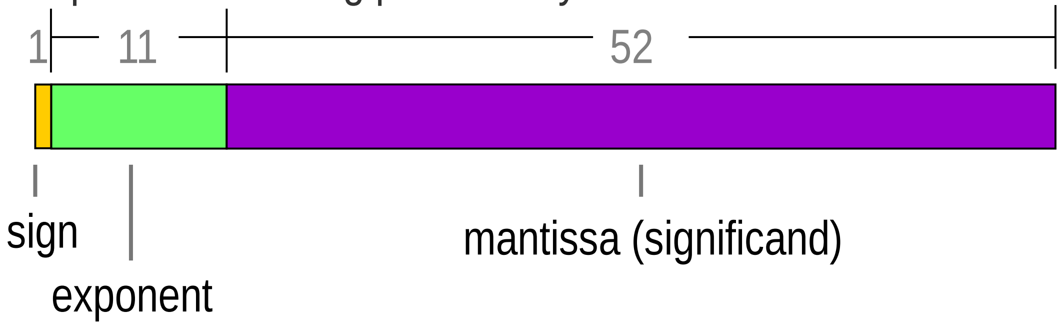
What is the largest signed integer?

Single precision floating point – 4 bytes = 32 bits



About 7 significant digits

Double precision floating point – 8 bytes = 64 bits



About 15 significant digits

Floating point numbers

Example. Adding two numbers. Some one has the code with following

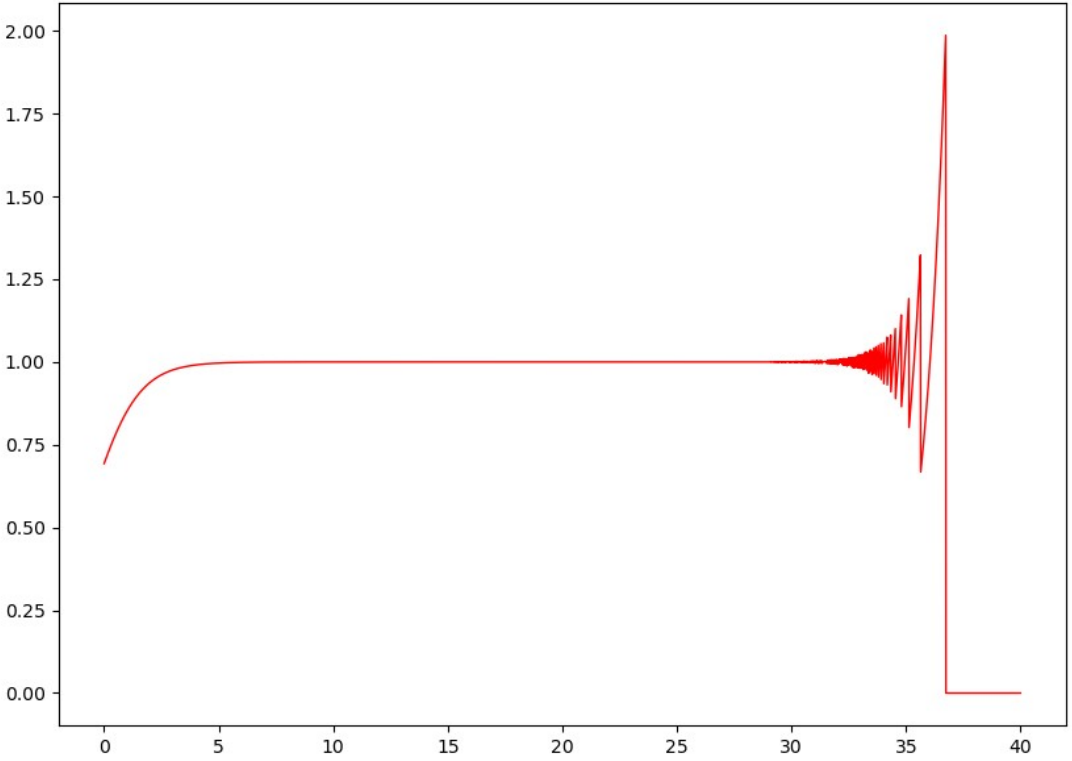
```
float a= -1e-10; // a = 0.0000000001
… …
If (a < 0) { // This is fine
   a += 10.0;
   do something using a; // Problematic: a is 10.
}
```

Problem: a is 10 rather than 9.9999999999, the value wanted, which causes floating point exception in the user’s code.

Fix: Using double precision fixes the problem.

Floating point numbers

Example. Broken curve $e^x \ln(1 + e^{-x})$



C. Essex, M. Davison, C. Schulzky, “Numerical monsters”, *ACM SIGSAM Bulletin*, vol. 34, Iss. 4, Dec., 2000, pp 16-32.

Some facts, needing higher precision:

- Base conversion. Work at IBM in 1999 showed that in some cases to guarantee correct number-base conversion, intermediate results
  - 32-bit format needs 126 bits (38 decimal digits),
  - 64-bit format needs 752 bits (227 decimal digits),
  - 128-bit format needs 11,503 bits (3463 decimal digits).
- Correct rounding.
- Vector inner product (or summation) accurate to the next-to-last digit:
- Science and engineering applications where higher precision than 17 digits after decimal point (double) are desired.
- Mathematical reasoning/proof (aka “experimental math”) expects extremely large number of digits (e.g. 500 digits in computing the determinants of some matrices).

Applications

- Supernova simulations (32 or 64 digits) (1999).
- Climate modeling (32 digits).
- Planetary orbit calculations (32 digits).
- Coulomb n-body atomic system simulations (32-120 digits) – ***A. Frolov*** and ***D. H. Bailey*** (2003).
- Schrodinger solutions for lithium and helium atoms (32 digits).
- Electromagnetic scattering theory (32-100 digits).
- Studies of the fine structure constant of physics (32 digits).
- Scattering amplitudes of quarks, gluons and bosons (32 digits).
- Theory of nonlinear oscillators (64 digits).
- Study of Riemann Hypothesis (500 digits) – ***John Nuttall*** (2011)

MP: Languages and packages

MP: Languages and packages

Example. Calculate 12345678910*10987654321

```
#include <stdio.h>

void main(void) {
    long int n1, n2, n3;

    printf("Enter n1: ");
    scanf("%ld", &n1);
    printf("Enter n2: ");
    scanf("%ld", &n2);
    n3 = n1 * n2;
    printf("n1 * n2 = %ld\n", n3);
}
```

What answer do you get on your computer?

This is a WRONG code

Example. Calculate 12345678910*10987654321

```
program xlimul
   implicit none
   integer(kind=selected_int_kind(36)) :: n1, n2, n3 ! 10^36
   !integer(kind=16) :: n1, n2, n3        ! 128-bit integer

   print *, 'Enter n1, n2:'
   read *, n1
   read *, n2
   n3 = n1 * n2
   print *, 'n3 =', n1, 'x', n2, '=', n3
end program xlimul
```

```
12345678910 x 10987654321 = 135650052221140070110
```

Note: **gfortran** 4.4 and newer supports large integers.

MP: Languages and packages

Python

```
Python 3.9.1 (default, Dec  8 2020, 07:51:42)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 135650052221140070110*135650052221140070110
18400936667598028068313222048255715412100
>>>
```

Julia

```
julia> n1 = big"135650052221140070110"
135650052221140070110

julia> n2 = big"135650052221140070110"
135650052221140070110

julia> n1*n2
18400936667598028068313222048255715412100
```

MP: Languages and packages

Example: Calculating large integers using **GMP**

```
#include <stdio.h>
#include <gmp.h>

void main(void)
{
    mpz_t n1, n2, n3;

    printf("Enter n1: ");
    gmp_scanf("%Zd", n1);
    printf("Enter n2: ");
    gmp_scanf("%Zd", n2);

    mpz_mul(n3, n1, n2);
    gmp_printf("n1 * n2 = %Zd\n", n3);
}
```

Example: Calculating large integers using **ARPREC**

```
#include <iostream>
#include "arprec/mp_real.h"

void main(void)
{
    mp::mp_init(100);
    mp_real n1, n2, n3;

    std::cout << "Enter n1: ";          // Must append a comma ‘,’
    std::cin >> n1;
    std::cout << "Enter n2: ";          // Must append a comma ‘,’
    std::cin >> n2;

    n3 = n1 * n2;                       // More elegant

    std::cout << "n1 * n2 = " << n3 << std::endl;
}
```

Function names an operators overloaded

MP: Languages and packages

Languages that support arbitrary precision arithmetic:

- **BC** – A Unix command line, multiprecision calculator.
- **Perl** – Some people are using perl for HPC.
- **PHP** – Used on web servers.
- **Ruby** – Mostly used on web servers.
- **Haskell**.
- **Python** – **mpmath**.
- **Fortran** – The Fortran standard has **selected_int_kind()** and **selected_real_kind()** that specifies the range and precision that variable of that kind can take.
- ***Julia*** – A new, emerging language for high performance and productivity.

MP: Languages and packages

A number of open source arbitrary precision **packages** available:

- **ARPREC**. Uses 64-bit FP arrays to represent numbers. Includes many algebraic and transcendental functions. Has C++ and Fortran 90 interfaces, supporting ***real***, ***integer*** and ***complex*** data types.
- **MPFUN 2020**. Fortran subroutines.
- **QD**. C++ and Fortran 90 interfaces.
- **GMP**. Uses platform dependent word size for exponent. C interface only.
- **MPFR**. A C library for multiple-precision floating-point computations with exact rounding, based on the GMP multiple-precision library. http://www.mpfr.org.
- **MPMATH** – A Python package.
- **MPACK** – Multiprecision BLAS and LAPACK C library based on GMP, etc. by NAKATA Maho (中田真秀) etc.

MP: Languages and packages

Commercial packages

- **Maple**.
- **Mathematica**.
- **Matlab** multiprecision toolboxes:
  - Advanpix
  - Multiple Precision Toolbox by Ben Barrowes (Free, at Matlab file exchange). Uses GMP.
  - Z. L. Krougly and D. J. Jeffrey, **Implementation and application of extended precision in Matlab**, *MMACTEE'09: Proceedings of the 11th WSEAS international conference on Mathematical methods and computational techniques in electrical engineering*, 2009, pp 103-108. (Uses ARPREC)

Multiprecision BLAS, LAPACK and special functions are really wanted.

MP: Languages and packages

Choosing A Package

- Does it have the interface to the language you use?
- Does it support operator, function overloading?
- Does it have transcendental function implementation?
- Easy to port to existing code, software?
- Performance?

So **GMP/MPFR** or **ARPREC**?

- Are you using C/C++ or Fortran?
- What are you doing?
- How good are you at the language?
- How much time do you have?

MP: Languages and packages

Notes on the key packages of potential significance:

- **ARPREC** support integer, real and complex numbers, for both C++ and Fortran, intrinsic functions, operators are overloaded, convenient for scientists to use. Source: https://www.davidhbailey.com/dhbsoftware/

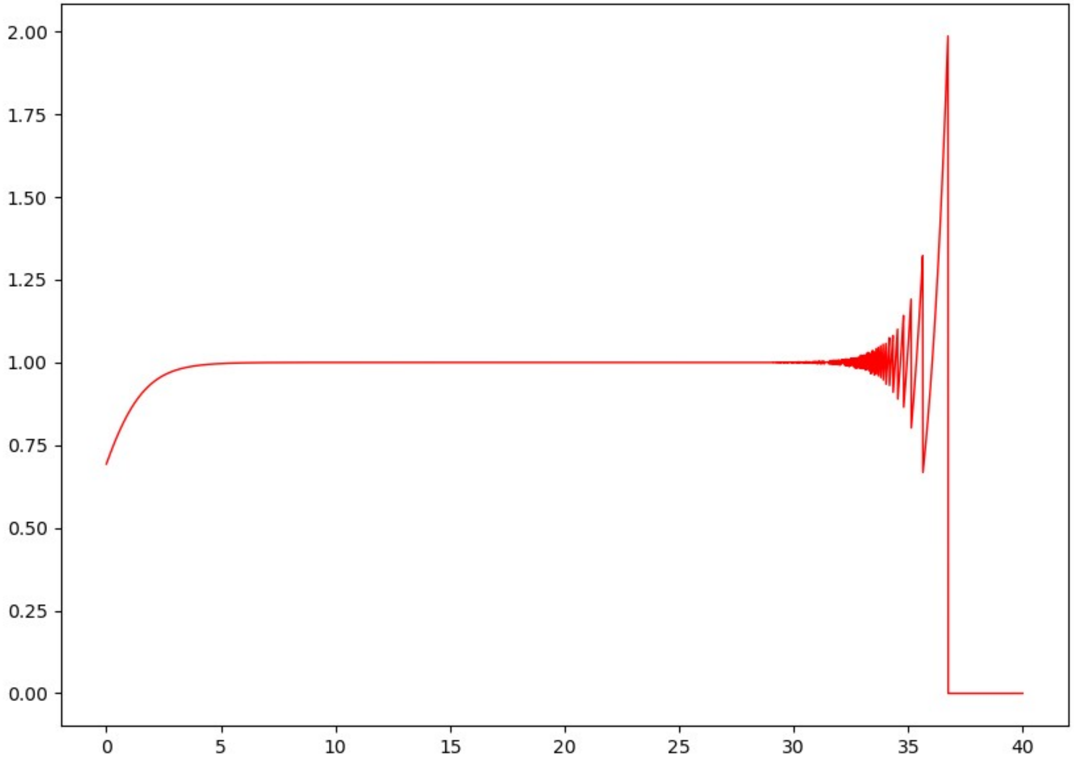
  Note: There is a bug in the latest release 2.2.18, a simple fix was suggested by Paul Preney (University of Windsor, SHARCNET).

- **MPFUN 2020** same as **ARPREC**, supports integer, real and complex numbers, for Fortran only, thread-safe.
- **MPACK** contains several hundreds of multi-precision BLAS and LAPACK routines, but has been inactive since 2012. Source: https://github.com/nakatamaho/mplapack
- **MPMATH**, a Python package, supports a wide range of mathematical functions and linear algebra options.

Examples

Floating point numbers

Example. Broken curve $e^x \ln(1 + e^{-x})$

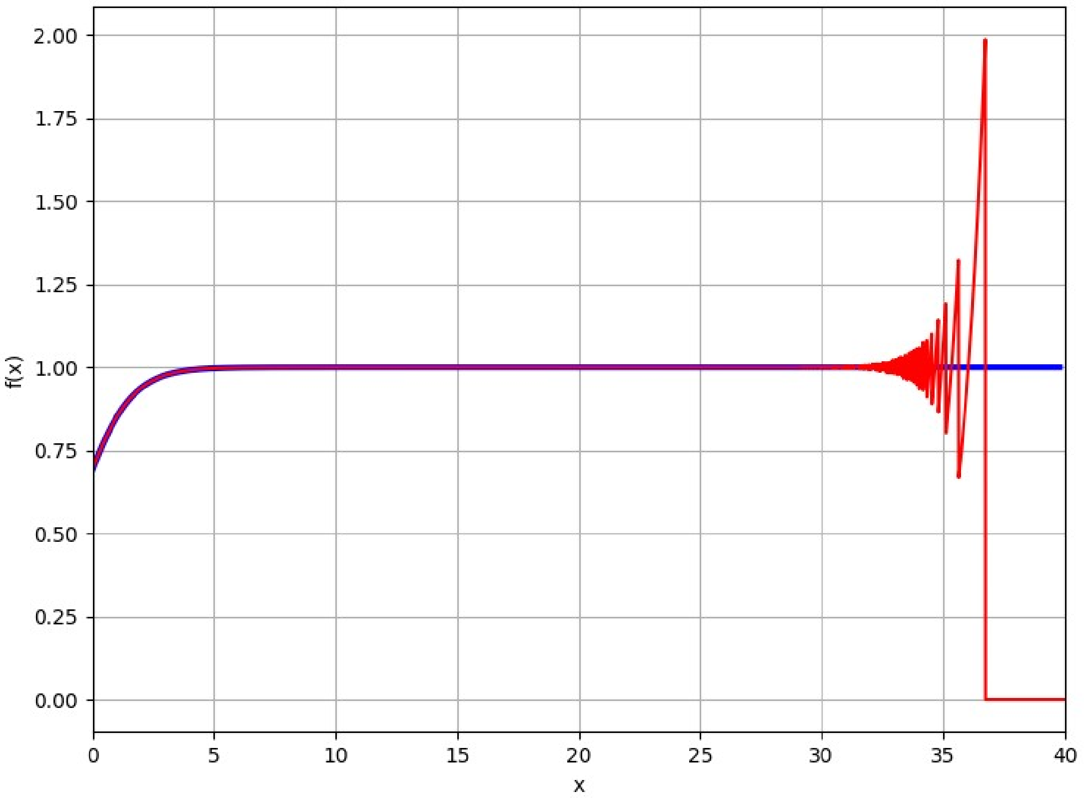


C. Essex, M. Davison, C. Schulzky, “Numerical monsters”, *ACM SIGSAM Bulletin*, vol. 34, Iss. 4, Dec., 2000, pp 16-32.

Examples

Example: Broken curve $e^x \ln(1+e^{-x})$

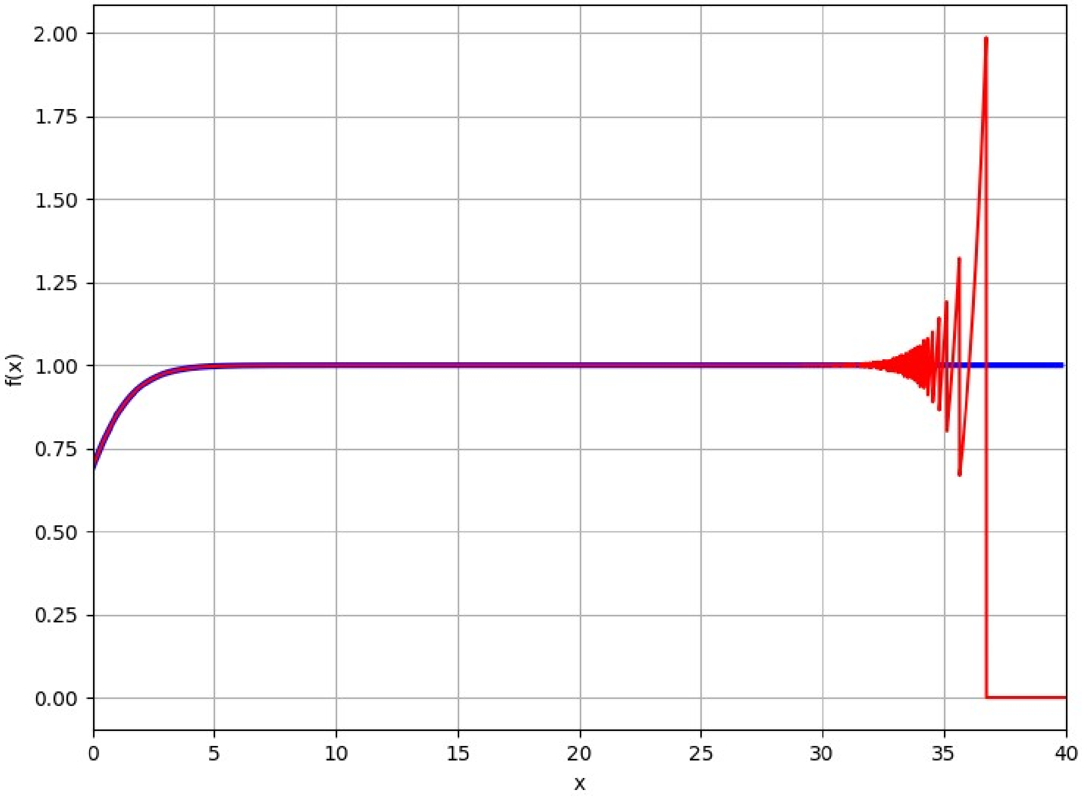
```
import matplotlib.pyplot as plt
import mpmath as mp
mp.dps = 50

# Plot the fixed curve – in blue
from mpmath import *
plot(lambda x: exp(x)*log(1+exp(-x)),[0,40])
```



Examples

Example: Broken curve $e^x \ln(1 + e^{-x})$, fixed using Python package mpmath. 50 digits are used.

Question: Where does it break again?



Examples

Example: Compute the eigenvalues $Ax = \lambda x$. The Wilkinson's eigenvalue test matrices W: An b-by-n tridiagonal matrix, with diagonal elements

$$d_i = |n \div 2 - i + 1|\,, \text{ when } i \text{ is odd; } d_i = |n \div 2 - i + 0.5| \text{ otherwise, } i = 1, \ldots, n$$

and 1's on its off-diagonals. The Wilkinson matrix of order 7 is as follows

$$W = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \end{bmatrix}.$$

The test matrix has a property that the eigenvalues seem to come in pairs of identical values, but they are really not. To verify this, we need higher precision.

Examples

Example: Eigenvalues of Wilkinson test matrix of order 21, this is frequently used case

```
-1.1254415221199840502
0.25380581709667954238
0.94753436752929509623
1.7893213526950828562
2.1302092193625097316
2.9610588841857246045
3.0430992925788821391
3.9960482013836222267
4.0043540234408530054
4.9997824777429036303
5.0002444250019131289
6.0002175222570954816
6.0002340315841653506
7.0039522095286681491
7.0039517986163675189
8.0389411228290263978
8.0389411158142749514
9.2106786473613322386
9.2106786473049151454
10.746194182903389347
10.746194182903316516
```

Examples

Example: Python for eigenvalues of Wilkinson 50

```
import numpy as np
import mpmath as mp

mp.dps = 40
n = 50
m = n % 2

# Create the diagonal of Wilkinson test matrix
from mpmath import *

d = zeros(n,1)
if (m == 1):
    for i in range(n):
        d[i] = abs(n//2 - i)
else:
    for i in range(n):
        d[i] = abs(n//2 - i - 0.5)
t = ones(n-1,1)

# Create Wilkinson test matrix W of order n
W = np.diag(t,-1) + np.diag(d,0) + np.diag(t,1)
mW = matrix(W)

E = eig(mW,right=False)
for x in E:
    nprint(x,n=40)
```

```
-0.9641321726904788469203807653558725272822
0.2126628425131107295702128494459102204897
1.078164406734610586751527260933515250175
1.654730092254134485579766753257886141779
2.400771751580549653471719453673745125832
2.613723558709657970251705419366973354982
3.486274955522127377316131346219042782907
3.517639311060085206913357746819011821998
4.498968620932769352208565164016883576925
4.501192469653747967060030800142379067747
5.499953844648219428838229822039180544886
5.500050238625574493614629202431453164637
6.499998623277988915591995913147693974563
6.500001455954136799897192333983951377828
7.49999997047881137165744688031253514501
7.50000003073005952645705174835347834576
8.499999999521540927978771807129468065256
8.500000000493206277760349451609484527164
9.500000000006222657447847663151767192198
9.499999999993923902705172850882166649745
10.50000000000006335023029689638865159737
10.49999999999993785719572185114285224896
11.50000000000000531458797213955220917777
11.49999999999999947691728926053358946624
12.50000000000000000402493853651720306438
12.49999999999999999659868424472185766044
13.50000000000000000442144353253788810578
13.50000000000000004416996709974985823935
14.50000000000000057424636181085647939243
14.500000000000000574246384732497574527509
15.500000000000061978957209295208731548252
15.500000000000061978957209397995975329569
16.5000000000544881079904291432979229451
16.500000000005448810799042914734465413927
17.50000000380812688353694056838881030954
17.50000000380812688353694056840291039124
18.50000020507043780031864988451635333252
18.50000020507043780031864988451630954353
19.50000815867294501046405034523704928011
19.50000815867294501046405034523704915813
20.50022568018517034412527273640041808201
20.50022568018517034412527273640041808232
21.50395200266536132836611456811806949642
21.50395200266536132836611456811806949642
25.24619418290335757058688396767205574031
22.5389411193064408897674059026371188482
22.5389411193064408897674059026371188482
23.71067864733304648832769963393362026697
23.71067864733304648832769963393362026697
25.24619418290335757058688396767205574031
```

Python
+
mpmath

Examples

Fortran code using MPFUN

```
program steig
  use mpmodule
  implicit none
  type ( mp_real ), allocatable :: d(:), t(:), work(:), z(:,:)
  integer :: info, integer :: i, n, m, num_digits

  print *, 'Enter matrix size:'
  read *, n
  print *, 'Enter accuracy (number of digits):'
  read *, num_digits
  call mpinit( num_digits )
  call mpsetoutputprec( num_digits )

  allocate(d(n), t(n), z(n,n), work(2*n-2))

  create Wilkinson eigenvalue test matrix

  call imtql1(n, d, t, info)
  print *, 'Eigenvalues (Using EISPACK):'
  do i = 1, n
    call mpwrite( 6,c(i) )
  end do
end program steig
```

EISPACK subroutine

```
! This QL algorithm was written in the 1969s…
! We don’t want to rewrite it!
      subroutine imtql1(n,d,e,ierr)
      use mpmodule
      integer i,j,l,m,n,ii,mml,ierr
      type ( mp_real ) d(n),e(n)
      type ( mp_real ) b,c,f,g,p,r,s,tst1,tst2
c
c this subroutine is a translation of the algol procedure
c imtql1, num. math. 12, 377-383(1968) by martin and
c wilkinson,
c … …

      … … Rest of the code … …
      … …
      return
      end
```

With little effort – three lines in this case – I was able to compute eigenvalues to 60 digits or more!

Examples

Example: Eigenvalues of Wilkinson test matrix of order 50, computed using double and 40 digits

```
-0.96413217269049067        10 ^   -1 x -9.641321726904788469203807653558725272820102494785,
 0.21266284251310127        10 ^   -1 x  2.126628425131107295702128494459102204898454357475,
 1.0781644067346108         10 ^    0 x  1.078164406734610586751527260933515250175346242597,
 1.6547300922541341         10 ^    0 x  1.654730092254134485579766753257886141779285333348,
 2.4007717515805513         10 ^    0 x  2.400771751580549653471719453673745125831512658936,
 2.6137235587096592         10 ^    0 x  2.613723558709657970251705419366973354981509584181,
 3.4862749555221253         10 ^    0 x  3.486274955522127377316131346219042782906283078552,
 3.5176393110600825         10 ^    0 x  3.517639311060085206913357746819011821997391706608,
 4.4989686209327697         10 ^    0 x  4.498968620932769352208565164016883576924323795219,
 4.5011924696537475         10 ^    0 x  4.501192469653747967060030800142379067747029642296,
 5.4999538446482141         10 ^    0 x  5.499953844648219428838229822039180544884672380507,
 5.5000502386255690         10 ^    0 x  5.500050238625574493614629202431453164635212134466,
 6.4999986232779925         10 ^    0 x  6.499998623277988915591995913147693974561916204077,
 6.5000014559541404         10 ^    0 x  6.500001455954136799897192333983951377827078801077,
 7.4999999704788101         10 ^    0 x  7.499999970478811371657446880312535145009630930559,
 7.5000000307300576         10 ^    0 x  7.500000030730059526457051748353478345759432546457,
 8.4999999995215454         10 ^    0 x  8.499999999521540927978771807129468065254241367295,
 8.5000000004932090         10 ^    0 x  8.500000000493206277760349451609484527161829931395,
 9.4999999999939302         10 ^    0 x  9.499999999993923902705172850882166649743950242226,
 9.5000000000062279         10 ^    0 x  9.500000000006222657447847663151767192196337560297,
 10.499999999999938         10 ^    1 x  1.049999999999993785719572185114285224895492394635,
 10.500000000000062         10 ^    1 x  1.050000000000006335023029689638865159736459003164,
 11.500000000000004         10 ^    1 x  1.149999999999999947691728926053358946623871069579,
 11.500000000000012         10 ^    1 x  1.150000000000000053145879721395522091776566792268,
 12.499999999999995         10 ^    1 x  1.249999999999999999659868424472185766044029876359,
 12.500000000000009         10 ^    1 x  1.250000000000000000402493853651720306437850545540,
 13.499999999999998         10 ^    1 x  1.350000000000000000441699670997498582393493325464 2,
 13.500000000000004         10 ^    1 x  1.350000000000000000442144353253788810577961989052 7,
 14.499999999999998         10 ^    1 x  1.450000000000000057424636181085647939243013823437 2,
 14.500000000000007         10 ^    1 x  1.450000000000000057424638473249757452750886745169 7,
 15.500000000000616         10 ^    1 x  1.550000000000061978957209295208731548251605838777,
 15.500000000000622         10 ^    1 x  1.550000000000061978957209397905975329569117374561,
 16.5000000000054[illegible]  10 ^    1 x  1.65[illegible],
 16.500000000054488         10 ^    1 x  1.650000000000544881079904291473446541392706013251 2,
 17.5[illegible]            10 ^    1 x  1.75[illegible],
 17.5[illegible]            10 ^    1 x  1.75[illegible],
 18.500000205070439         10 ^    1 x  1.850000020507043780031864988451630954352408854039,
 18.500000205070439         10 ^    1 x  1.850000020507043780031864988451635333251853861565,
 19.500008158672937         10 ^    1 x  1.950000815867294501046405034523704915813165864541,
 19.500008158672948         10 ^    1 x  1.950000815867294501046405034523704928010408453453,
 20.500225680185167         10 ^    1 x  2.050022568018517034412527273640041808200744223750,
 20.500225680185167         10 ^    1 x  2.050022568018517034412527273640041808231370908819,
 21.503952002665365         10 ^    1 x  2.150395200266536132836611456811806949641606804821,
 21.503952002665386         10 ^    1 x  2.150395200266536132836611456811806949641675645570,
 22.538941119306436         10 ^    1 x  2.253894111930644088976740590263711884819586263287,
 22.538941119306440         10 ^    1 x  2.253894111930644088976740590263711884819586491979,
 23.710678647333044         10 ^    1 x  2.371067864733304648832769963393362026697201398322,
 23.710678647333058         10 ^    1 x  2.371067864733304648832769963393362026697201896063,
 25.246194182903331         10 ^    1 x  2.524619418290335757058688396767205574031125611895,
 25.246194182903359         10 ^    1 x  2.524619418290335757058688396767205574031125782293,
```

20.500225680185167   **10 ^**   **1 x 2.050022568018517034412527273640041808200744223750,**

20.500225680185167   **10 ^**   **1 x 2.050022568018517034412527273640041808231370908819,**

Fortran

+

mpfun/arprec

+

An EISPACK routine

Example

Example: Use of MPACK

```
#include <mblas_gmp.h>
#include <mlapack_gmp.h>

int main()
{
    mpackint n = 3;
    mpackint lwork, info;

    int default_prec = 256;
    mpf_set_default_prec(default_prec);

    mpf_class *A = new mpf_class[n * n];
    mpf_class *w = new mpf_class[n];

    // Create matrix [[1 2 3], [ 2 5 4], [3 4 6]]
    A[0 + 0 * n] = 1;   A[0 + 1 * n] = 2;   A[0 + 2 * n] = 3;
    A[1 + 0 * n] = 2;   A[1 + 1 * n] = 5;   A[1 + 2 * n] = 4;
    A[2 + 0 * n] = 3;   A[2 + 1 * n] = 4;   A[2 + 2 * n] = 6;

    //work space query
    lwork = -1;
    mpf_class *work = new mpf_class[1];

    // Compute the eigenvalues and eigenvectors
    Rsyev("V", "U", n, A, n, w, work, lwork, &info);

    //print out some results
    printf("#eigenvalues \n");
    printf("w =");
    printmat(n, 1, w, 1);
    printf("\n");
    printf("#eigenvecs \n");
    printf("U =");
    printmat(n, n, A, n);
    printf("\n");

    delete[]work;
    delete[]w;
    delete[]A;
}
```

MPACK keeps the same interface as BLAS and LAPACK routines, with *Rname* for real and *Cname* For complex routines

Examples

Note: linear algebra operation, there seems to be only two viable options:

- One may use **Python** package **MPMATH** for multiprecision linear algebra operations, but it’s Python.
- For the group using compiled languages Fortran and C/C++, the support for multiprecision linear algebra operations, as provided in BLAS and LAPACK, is still limited. **MPACK** is for C++, can’t be used directly for Fortran
- Because the underling key routines are “hard coded” in specific precision, there is (so far) no automatic mechanism to translate the targeted functions automatically to arbitrary precision.

References

Further reading

- D. H. Bailey, **MPFUN2020: A new thread-safe arbitrary precision package**, December 12, 2020, https://www.davidhbailey.com/dhbsoftware/
- D. H. Bailey, J. M. Borwein and , R. Barrio, "**High Precision Computation: Mathematical Physics and Dynamics**", 2009.
- Fredrik Johansson et al, **MPMATH** 1.1.0 documentation, https://mpmath.org/doc/current/index.html
- 中田真秀 (Nakata Maho), **The MPACK: Multiple precision arithmetic BLAS and LAPACK**, https://github.com/nakatamaho/mplapack/