# MATH 7203
# Numerical Analysis 1
# Northeastern University
# Spring 2023

# Numerical Analysis 1

- Lecturer:
  - Dr. Stuart Brorson s.brorson@northeastern.edu
  - Office hours: Friday 3:30-5:30pm (before class)
- Grader:
  - Hiu Ying Man man.h@northeastern.edu
- Format: Two 1 ½ hour lecture sessions on Zoom.
  - 10 minute break between sessions.
- Communicating with class: e-mail and Canvas.
- You need a computer + Matlab (or other numerically-oriented math package).

# Major themes

- This class emphasizes numerical computation as done in real-world engineering and science.
- Continuous quantities.
- Evaluation of functions. Root finding.
- Matrix computations. Numerical linear algebra.
- Filtering, interpolating, processing and analyzing real data.
- Integration and solving ODEs.
- Assumes you have some programming experience (MATLAB).

# Major themes cont'd

- We will mention many different software libraries and environments for numerics.
- Use the existing numerical libraries – don't write what's already written!
- How computer architecture impacts your computation. You can easily get bitten by computer math!
- Numerical stability and accuracy.
- Emphasis on real-world software practice.
  - Test, test, test!

# Class format

- Class is in Forsyth 129
- Session 1 – 90 min (5:50 – 7:25)
  - Expand on material from previous lecture and/or do a problem
  - New material
- Break – 10 min.
- Session 2 – 90 min (7:35 – 9:10)
  - New material

# About the homework...

- Problems are a mixture of derivations and writing computer programs.
- Solutions due 1 1/2 weeks after lecture (at 11:59pm on Sunday).
  - But don't start late!
- Write a program in Matlab
  - Other languages by pre-arrangement with instructors.
- Please create a test program to exercise your function implementation.
- Stumped? Google is your friend.
  - But don't just copy somebody else's code.

# Copying and plagiarism

- It's OK to meet with each other and discuss the homework.  In fact, we encourage it.
- It's OK to implement the same algorithm based upon discussion with your classmates.
- It's ***not*** OK to simply copy somebody's code.
  - Copying code and changing variable names is ***not*** OK.
  - We will give 0 credit to ***both*** the copyer and the copyee if we suspect you of copying.
- The goal is for you to learn by doing.  This implies ***you*** need to do the problems.

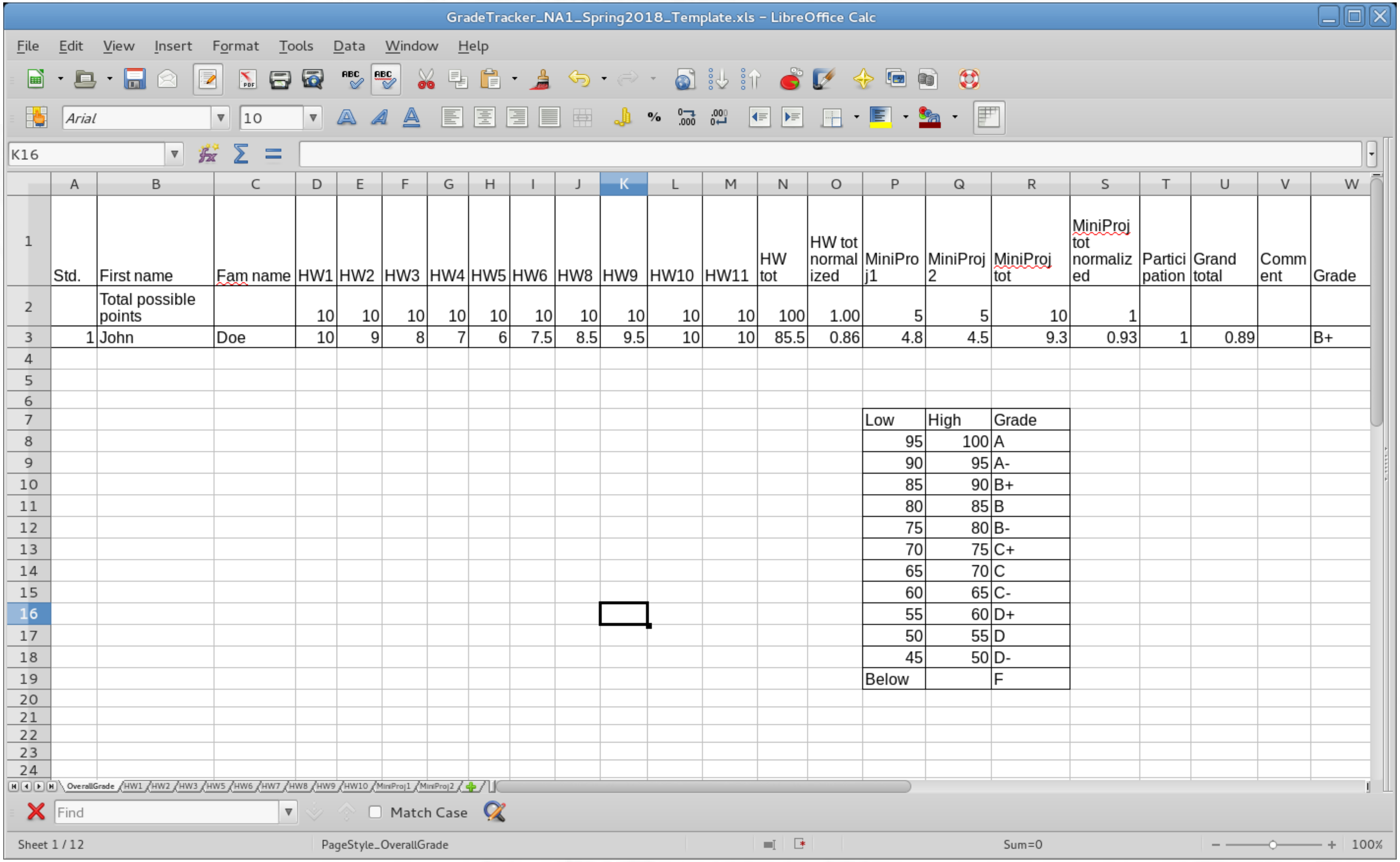
# Mini-projects

- Write a non-trivial program to implement one or more algorithms pertinent to a topic which interests you.
  - We will circulate a list of suggestions long before projects are due.
  - Be on the lookout for things which interest you.
- Fill out proposal form & get approval prior to starting project.
- Write code and test solving your problem.
- Give a 10 minute PPT presentation about your work to class.
  - Overview of problem domain – educational for everybody.
  - Specifics of your work.
- Turn in your code and your project slides for grade.
- One person – one project.

# Grading

- Problem sets: 60% (problems are individually ranked for importance).
- Mini-projects: 35%.
- Class participation: 5%.
- Grades computed using spreadsheet.

# Grading spreadsheet




| Std. | First name | Fam name | HW1 | HW2 | HW3 | HW4 | HW5 | HW6 | HW8 | HW9 | HW10 | HW11 | HW tot | HW tot normalized | MiniProj1 | MiniProj 2 | MiniProj tot | MiniProj tot normalized | Participation | Grand total | Comment | Grade |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total possible points | | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 100 | 1.00 | 5 | 5 | 10 | 1 | | | | |
| 1 | John | Doe | 10 | 9 | 8 | 7 | 6 | 7.5 | 8.5 | 9.5 | 10 | 10 | 85.5 | 0.86 | 4.8 | 4.5 | 9.3 | 0.93 | 1 | 0.89 | | B+ |

| Low | High | Grade |
|---|---|---|
| 95 | 100 | A |
| 90 | 95 | A- |
| 85 | 90 | B+ |
| 80 | 85 | B |
| 75 | 80 | B- |
| 70 | 75 | C+ |
| 65 | 70 | C |
| 60 | 65 | C- |
| 55 | 60 | D+ |
| 50 | 55 | D |
| 45 | 50 | D- |
| Below | | F |

Copy of template is available on Canvas

# Textbook and Readings

- Our “official” textbook is “Data-Driven Modeling and Scientific Computation” by J. Nathan Kutz.
- There are many excellent numerical analysis books available online for free, including
  - “Numerical Recipes in C”, by W. H. Press et al.
  - “Numerical Computing with MATLAB”, by C. Moler.
- Good “dead tree” books:
  - “Numerical Methods using MATLAB”, by Lindfield and Penny.
  - “Computational Science and Engineering”, by G. Strang.
- We will use many sources, including online sources, papers, etc.

# Class Resources on the Web

- This class is on NEU's Canvas system.
  - Lecture and recitation slides
  - Homework
  - Links to online materials
  - Discussion group
- Videos about Numerics (not mine):
  - Nathan Kutz: http://faculty.washington.edu/kutz/KutzBook/KutzBook.html
  - Gil Strang: https://www.youtube.com/playlist?list=PL49CF3715CB9EF31D

# Office hours and support

- Dr. Brorson:  Friday 3:30 – 5:30pm in Nightingale 543B or via Zoom.
- You can always E-mail the instructors with questions or concerns.
- Hiu Ying Man (Mandy) is Grader/TA.
- We have a discussion forum on Canvas.

# Common math programs/languages

- **MATLAB** – Strongly matrix and numerics oriented.
- **Octave** – Open-source clone of MATLAB.
- **Scilab** – French university clone of MATLAB.
- **Python/Numpy** – Strongly matrix and numerics oriented, based on Python.
- **Julia** – New, fast math language from MIT.
- **Mathematica & Maple** – Aimed at symbolic computing, also do numerics.
- **R** – Statistics and data analysis.
- **LabView** – Graphical programming language, originally used for laboratory automation.
- **Fortran** – Original language for number crunching, still used for that purpose.
- **C/C++** -- Common, general purpose languages.
- **Java** – Another general purpose language with good math library.
- Others: **SAS, SPSS, SAGE,** etc.

# MATLAB

- Our default programming environment
- Who doesn't have it?
- Any problems running it?
- Moller's intro to Matlab is linked on Canvas:

  http://www.mathworks.com/moler/chapters.html

- Matlab demo:

```
X = randn(5)
Xi = inv(X)
Y = X*Xi
I = eye(5)
Z = Y - I
norm(Z)
```

# NumPy

- Analogous to Matlab, but built upon Python
- Python is general purpose language
- Linked on Canvas: “In Introduction to Numpy and Scipy”:

  http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf

# C/C++

- Compiled languages
- Produce fast running code
- Can be hard to use.
- Anybody used them before?

# Numerical evaluation of scalar functions

Kinds of functions we are talking about:

- Polynomials
- Rational functions
- Elementary functions – sin, cos, exp, log, etc.
- Special functions – gamma, Bessel, Mathieu, etc.
  - Abramowitz and Stegun – Used to be the “bible”.
  - Gradstein & Ryzhik
  - Jahnke and Emde
  - Bateman manuscript project
  - DLMF: http://dlmf.nist.gov/

# Computation of polynomials

What if you want to compute $3x^3 + 2x^2 + 5x + 1$ ???

**Bad:**

```
3*x*x*x + 2*x*x + 5*x + 1
```

**Very bad:**

```
3*power(x, 3) + 2*power(x, 2) + 5*x + 1
```

**Good:**

```
1 + x*(5 + x*(2 + x*(3)))
```

# Horner's rule

$$1 - 4x + 6x^2 - 3x^3$$
$$= 1 + x*(-4 + x*(6 + x*(-3)))$$

- Eliminates unnecessary duplication of multiplications (performance hit).
- Potentially reduces errors incurred when subtracting one large quantity from another. (Consider x = -2.0001 in above eq.)
- Amenable to use in for loop.
- Be careful of signs!

# Matlab examples

- Simple polynomial evaluation:
  - horners_rule.m
- Evaluate polynomial in loop:
  - eval_poly_bad.m
  - eval_poly.m
- Usage (evaluates $2 + 3x + 4x^2 +5x^3 +6x^4$):

```
a = [2 3 4 5 6]
eval_poly_bad(a, 2.3)
eval_poly_good(a, 2.3)
```

# Series expansions

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + \ldots$$

- Long ago, this is how many special functions were computed (or at least properties proven).
- In general, ***not*** a good way to compute functions over whole domain. Use a library function instead.
- Taylor series expansions start to fail near function singularities.
- If you have to write your own series expansion, be very mindful of numerical stability.

# Series expansion code – what does it look like?

- Series is written as a for loop over terms.
    1. Initialize computation (set local variables like running sum).
    2. Compute current term.
    3. Add term to running sum.
    4. Check for convergence by checking to see if the term falls below some tolerance.
    5. If not converged yet, update term.
- Simple example: arctan_series(x).

# Series expansion for arctan(x)

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \cdots$$

- Want to use a for loop to express this.
- Quantities to compute this:
  - Running sum $y$
  - Individual term $t = x, x^3, x^5, x^7, \ldots$
  - denom $n = 1, 3, 5, \ldots$
  - Sign (+/-) $s = +1, -1, +1, -1, \ldots$
  - Stopping tolerance, usually small

# Series summation algorithm

- Initialize computation

```
y = 0
s = 1
t = x
```

- Loop:
  - Add term to running sum `y = y + s*t/n;`
  - Update term `t = t*x^2`
  - Update sign `s = -s`
  - Check for convergence

Initialize computation

```
y = 0
s = 1
t = x
```

```
For n = 1:2:infinity
    y = y + s*t/n;
    Check for convergence
    t = t*x^2
    s = -s
```

| | $n$ | $y$ | $t$ | $s$ |
|---|---|---|---|---|
| Initial values (before loop) | | $0$ | $x$ | $1$ |
| After loop 1 | $1$ | $x$ | $x^3$ | $-1$ |
| After loop 2 | $3$ | $x-\frac{x^3}{3}$ | $x^5$ | $1$ |
| After loop 3 | $5$ | $x-\frac{x^3}{3}+\frac{x^5}{5}$ | $x^7$ | $-1$ |

$$\arctan(x)=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}\cdots$$

# Horner's rule example

*Do example on blackboard*

```
function y = arctan_series(x)
  % arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + ...

  % Initialize computation
  y = 0;
  s = 1;
  t = x;
  yold = y;
  tol = 1e-5;

  % Now compute terms inside loop.
  for n = 1:2:100
    y = y + s*t/n;

    % See if we are ready to stop summing
    if (abs(yold - y) < tol)
      % fprintf('----  Converged!  -----\n')
      return
    else
      yold = y;
      s = -s;
      t = t*x*x;
    end
  end
  error('----  Failed to converge  -----\n')
end
```

# Bad arctan(x)

```
function y = arctan_series_bad(x)
  % arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + ...

  % For comparison against our computed value...
  act = atan(x);

  % Initialize computation
  y = 0;  % Running sum
  s = 1;  % Sign of this term
  t = x;  % This term
  tol = 1e-10;

  % Now compute terms inside loop.
  for n = 3:2:100
    y = y + t;
    err = act - y;
    % printf('n = %f, s = %f, t = %f, y = %f, act = %f, err = %f\n', n, s, t, y, act, err);

    % See if we are ready to stop summing
    if (abs(t) < tol)
      % printf('----  Converged!  -----\n')
      return
    else
      % Update t
      s = power(-1, (n-1)/2);
      t = s*power(x, n)/n;
    end
  end
  % printf('----  Failed to converge  -----\n')
end
```

Bad practice! Power() is a time-consuming function call. Also, you're wasting work – computing powers of x over and over.

# arctan(x)

```
function y = arctan_series(x)
  % arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + ...

  % For comparison against our computed value...
  act = atan(x);

  % Initialize computation
  y = 0;    % Running sum
  s = 1;    % Sign of current term
  t = x;    % This is current term
  yold = y;
  tol = 1e-5;

  % Now compute terms inside loop.
  for n = 1:2:100
    y = y + s*t/n;
    err = act - y;
    printf('n = %f, s = %f, t = %f, y = %f, act = %f, err = %f\n', n, s, t, y, act, err);

    % See if convergence has been reached
    if (abs(yold - y) < tol)
      printf('----  Converged!  -----\n')
      return
    else
      % Not converged – update term
      yold = y;
      s = -s;
      t = t*x*x;
    end
  end
  printf('----  Failed to converge  -----\n')
end
```

This loop runs until the difference between yold and y is smaller than tolerance.

This is tricky – update current term by multiplying in x^2.

# Aside: Measuring performance using tic/toc

- Used for timing your code.
- tic -> start stopwatch, toc -> stop and report time
- In general, it's best to time your function in a loop to average out fluctuations.

```
octave:132> tic; for idx=1:10000; arctan_series(.3); end; toc
Elapsed time is 0.11665487289429 seconds.

octave:133> tic; for idx=1:10000; arctan_series_bad(.3); end; toc
Elapsed time is 0.25414800643921 seconds.
```

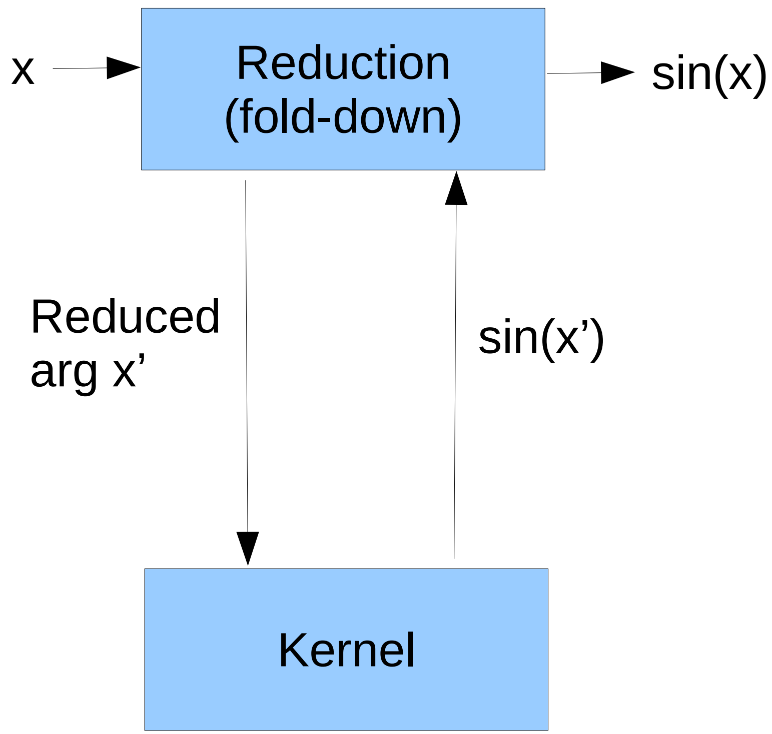
- Main point:  Pay attention to timing when your code is intended to scale

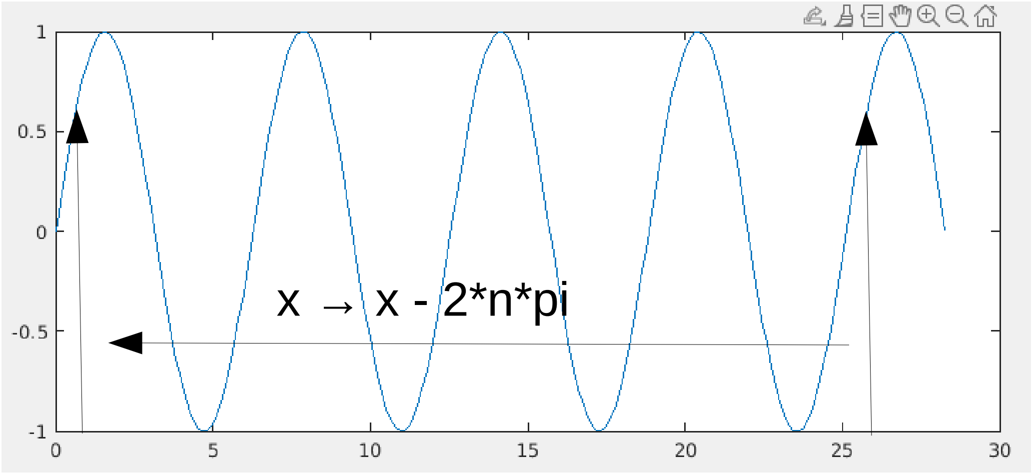
# Real-world example: computing trig functions

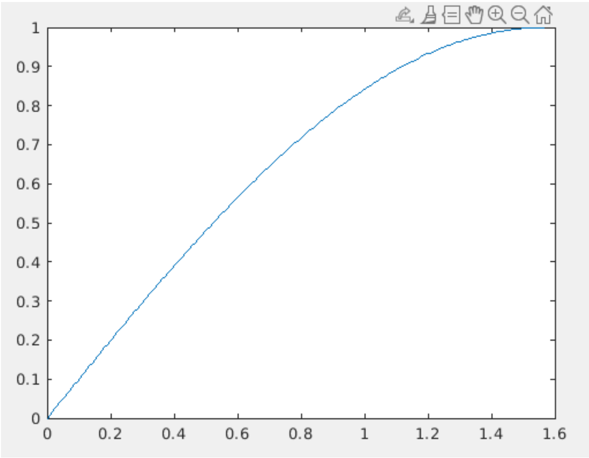
- Trig functions are periodic.
- Sin, cos are “entire”. Tan, cot are meromorphic.
- They all have series representations. However, don't sum the series for large inputs!
- Instead:
  - Fold input down to fundamental domain
  - Then use polynomial approximation (can be series, or other poly approximation).

# Computing sin(x)

- Top: reduction (fold down)
- Bottom level: kernel (series evaluation)




Evaluate series approximation for sin(x) in first quadrant.

# mysin(x) – top level

```
function y = mysin(x, tol)
  % Computes sin by folding input into domain 0 <= x <= pi/2
  % Then computes value using polynomial approximation

  piover2 = pi/2;
  pitimes2 = 2*pi;
  s = mod(x, pitimes2);  % Fold down to first full cycle

  if (s < piover2)
    y = P(s, tol);  % 0 <= x < pi/2
    return
  elseif (s < pi)   % pi/2 <= x < pi
    y = P(pi-s, tol);
    return
  elseif (s < 3*piover2)   % pi <= x < 3pi/2
    y = -P(s-pi, tol);
    return
  elseif (s <  pitimes2)   % 3pi/2 <= x < 2pi
    y = -P(pitimes2-s, tol);
    return
  else
    error("We failed!  x = %15.12e, s = %15.12e\n", x, s)
    y = nan;
    return
  end

end
```

# mysin(x) – kernel (series sum)

$$s\left(1-\frac{1}{3\cdot 2}s^2+\frac{1}{5\cdot 4\cdot 3\cdot 2}s^4-\frac{1}{7!}s^6\cdots\right)$$

```
function z = P(s, tol)
  sum = 1;
  sign = 1;
  term = 1;
  for p = 2:2:100
    denom = (p+1)*p;
    term = s*s*term/denom;
    sign = -sign;
    sum = sum + sign*term;
    if (term < tol)
      break
    end
  end
  z = s*sum;
  return
end
```

Note this is Horner's rule applied in a loop.

# sin(x) in fdlibm

- http://www.netlib.org/fdlibm/index.html
- Look for s_sin.c (arg reduction) and k_sin.c (computational kernel – polynomial evaluation)
- These use few terms – no loop needed.
- “Freely distributable lib m” lives on Netlib.
- If you have ever linked to -lm (lib m) in a C program, you have likely used this library.
- Netlib: A collection of commonly used math libraries for everybody to use.

# Some points

- You generally only need a few terms of a series.
  - Only use series expansions which converge quickly.
- Adding dozens or hundreds of terms to a series places you in danger of round-off error.
- There is a balance between truncation error and round-off error.
- You must also be aware of the domain of convergence for your power series.

# Continued Fraction Expansions

$$f(z) := b_0(z) + \cfrac{a_1(z)}{b_1(z) + \cfrac{a_2(z)}{b_2(z) + \cfrac{a_3(z)}{b_3(z) + \ddots}}}$$

- Generally converge faster than series summation.
- Some functions may be computed this way easily.
  - sqrt(x) – Derived at blackboard
  - tanh(x) is another example -- homework.

# sqrt(x)

$$\sqrt{x} = 1 + \cfrac{(x-1)}{2 + \cfrac{(x-1)}{2 + \cfrac{(x-1)}{2 + \cdots}}}$$

Alternate way to write the continued fraction

$$= 1 + \frac{(x-1)}{2+} \; \frac{(x-1)}{2+} \; \frac{(x-1)}{2+} \; \frac{(x-1)}{2+} \; \cdots$$

- Derived on blackboard

# Derivation

- Start with true statement $(\sqrt{x}+1)(\sqrt{x}-1)=x-1$

$$\sqrt{x}-1=\frac{x-1}{\sqrt{x}+1}$$ Divide out

$$\sqrt{x}=1+\frac{x-1}{\sqrt{x}+1}$$ Move 1 to RHS

- Now observe that I have $\sqrt{x}$ on both RHS and LHS. Substitute on RHS to get

$$\sqrt{x}=1+\frac{x-1}{1+\frac{x-1}{\sqrt{x}+1}+1}=1+\frac{x-1}{2+\frac{x-1}{\sqrt{x}+1}}$$

# Derivation

- From previous page: $$\sqrt{x}=1+\frac{x-1}{2+\frac{x-1}{\sqrt{x}+1}}$$
- Now we have $\sqrt{x}$ on both LHS and RHS again. Substitute again to get

$$\sqrt{x}=1+\frac{x-1}{2+\frac{x-1}{1+\frac{x-1}{\sqrt{x}+1}+1}}=1+\frac{x-1}{2+\frac{x-1}{2+\frac{x-1}{\sqrt{x}+1}}}$$

- Obviously I can repeat this procedure forever, thereby obtaining the desired continued fraction.

# Continued Fraction Computation Method by J. Wallis (1655)

$$f(z) := b_0(z) + \cfrac{a_1(z)}{b_1(z) + \cfrac{a_2(z)}{b_2(z) + \cfrac{a_3(z)}{b_3(z) + \ddots}}}$$

Convergents → $f_0$ $f_1$ $f_2$

- Problem: Where to start computation?
- Consider the "convergents" $f_n = \dfrac{A_n}{B_n}$

  where each convergent is the fraction truncated after n iterations.

- Convergents form a convergent sequence:

$$f_0, f_1, f_2, f_3, \cdots$$

# Wallis' Algorithm (1655)

Compute the sequence of convergents

1. Initialize variables: $\begin{pmatrix} A_{-1} \\ B_{-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} A_0 \\ B_0 \end{pmatrix} = \begin{pmatrix} b_0 \\ 1 \end{pmatrix}$
2. Loop: Compute next set of convergents:

$$\begin{pmatrix} A_j \\ B_j \end{pmatrix} = b_j \begin{pmatrix} A_{j-1} \\ B_{j-1} \end{pmatrix} + a_j \begin{pmatrix} A_{j-2} \\ B_{j-2} \end{pmatrix}$$

3. Check $|f_n - f_{n-1}| < \epsilon$ . If tolerance check passes, return.
4. Otherwise, loop again.

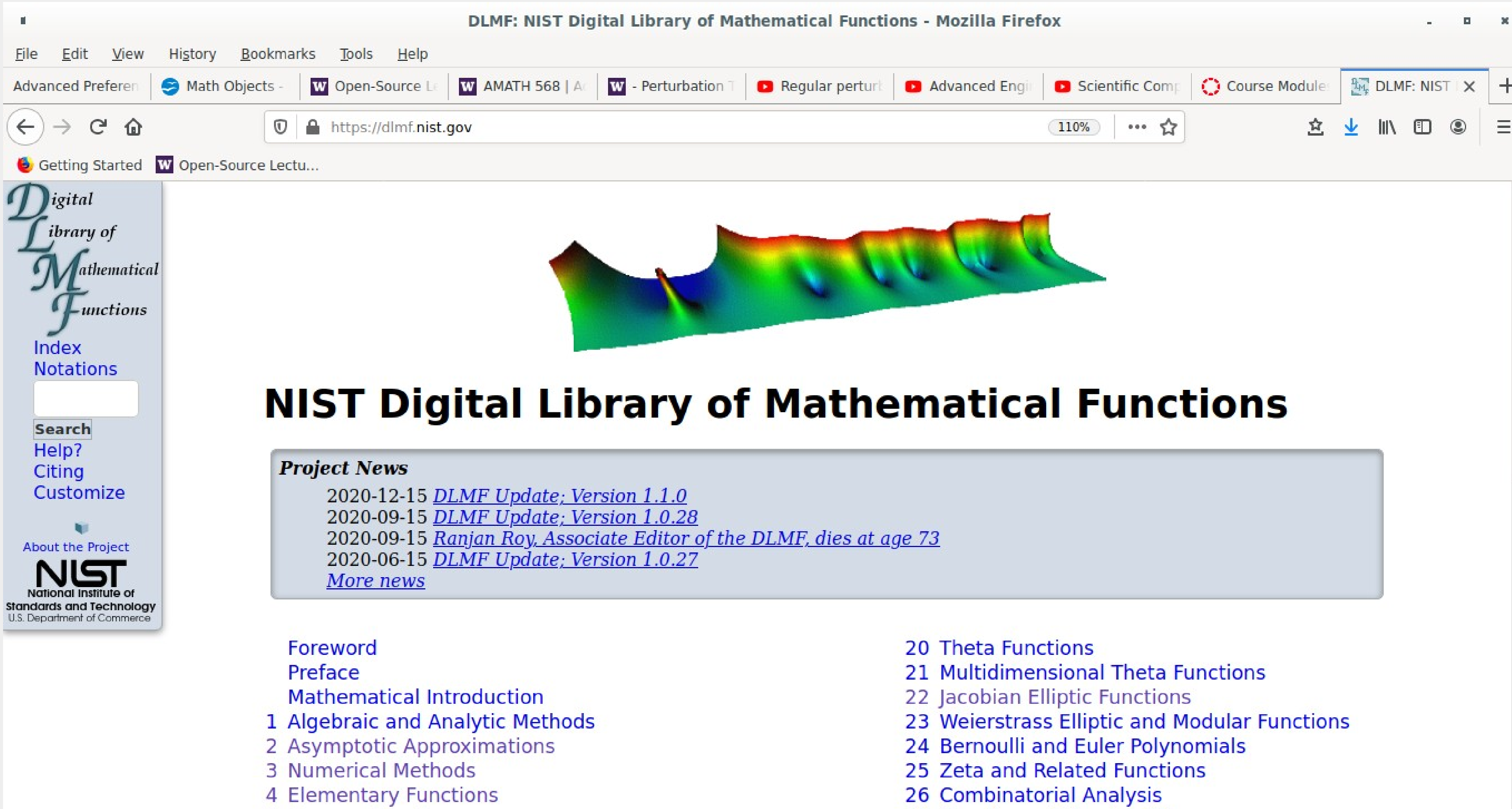
`Look at sqrt_contdfrac(x)`

```
function yn = sqrt_contdfrac(n)

 % Initialize computation
 a = n-1;  b0 = 1;  b = 2;
 Ajm2 = 1;  Bjm2 = 0;  Ajm1 = b0;  Bjm1 = 1;

 % Previous value and stopping tolerance used for stopping check.
 ynm1 = 0;
 tol = 1e-8;  % Stopping tolerance.

 % Note I use a for loop here, not a while loop.  While loops
 % can get stuck in infinite loops.
 for j = 1:500
   Aj = b*Ajm1 + a*Ajm2;
   Bj = b*Bjm1 + a*Bjm2;
   yn = Aj/Bj;
   diff = yn - ynm1;     % Compute difference to know if I am converging

   if (abs(diff) < tol)
     fprintf('sqrt_contdfrac converged in %d iterations, result = %f\n', j, yn)
     return
   end

   % Move values back in preparation for next loop iteration.
   ynm1 = yn;
   Ajm2 = Ajm1;  Bjm2 = Bjm1;  Ajm1 = Aj;  Bjm1 = Bj;

end
```

# Final remarks on continued fractions and computing functions

- Continued fraction expansions are “better” than Taylor's series, if you can find one.
  - Typically converges faster than a Taylor expansion for the same function.
  - Domain of convergence is larger.
- Better algorithm for numerics: Lentz algorithm.
- Large body of interesting theory about their properties.
- Not that common in the real world.

# The DLMF




https://dlmf.nist.gov/

# Some programs have their own built-in libraries

- Excel
- Matlab
- Python, Java, C/C++, C#, etc.

## Some are better than others

  - Completeness
  - Function coverage (stats, math, matrices, etc.)
  - Accuracy
  - Standards-compliance

# Math function libraries you should know about

- On the web
  - Netlib – fdlibm  (math.h)
  - Gnu Scientific Library (GSL)
- Commercial companies
  - NAG
  - IMSL
- Chip vendors offer their own libraries
  - MKL -- Intel
  - ACML -- AMD

# Next topic: Floating point numbers

- Attempt to model continuous real numbers (real number line).
  - In contrast to integers.
- Defined by IEEE-754 spec.
- 32 and 64 bit versions are implemented in hardware (***fast***).
- Float (32 bits) and double (64 bits)
  - single and double.
- Paper: “What every computer scientist should know...”

# What is sin(n*pi)?

```
>> n = 5; sin(n*pi)

ans =

     6.123233995736766e-16

>> n = 50; sin(n*pi)

ans =

     9.821933618642360e-16

>> n = 500; sin(n*pi)

ans =

    -1.607083229637817e-13
```

# What is $A*A^{-1} - I$?

```
>> A = randn(4)

A =

    0.8422   -0.1542   -0.0891    1.5512
   -2.0379   -0.8499    2.7480    1.6785
    0.7932    1.0704   -0.4785   -1.2255
   -0.6963   -1.4631   -0.4026    0.0918

>> B = inv(A)

B =

    1.6866    0.9562    3.5863    1.8906
   -1.1250   -0.8049   -2.6807   -2.0587
    1.0982    1.1497    3.0747    1.4659
   -0.3198   -0.5331   -2.0370   -1.1469

>> E = eye(4)

E =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> A*B-E
```

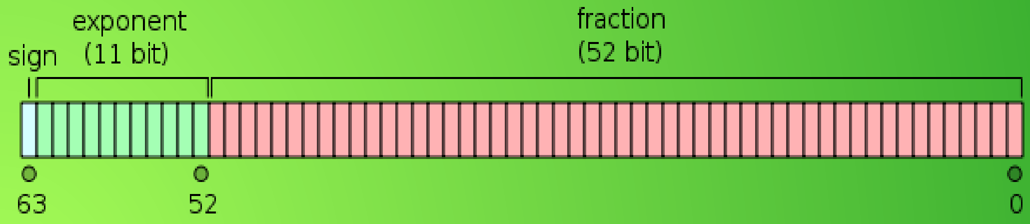
????

# Why?

- Numbers computers use are an *approximation* to “mathematically true” numbers.
- Matlab does most computations using “floating point” numbers.  Implemented in hardware.

# Representing real numbers in the computer

Floating point doubles – what you use in Matlab or R.




By Codekaizen (Own work) GFDL

- Doubles are 64 bits in exponential format.  Sign, mantissa, and exponent are encoded in the bit field.
- Standardized in IEEE 754. Leading light of the standard was Berkeley mathematician William Kahan.
- Implemented in hardware on modern microprocessors.  
==> Fast computations!

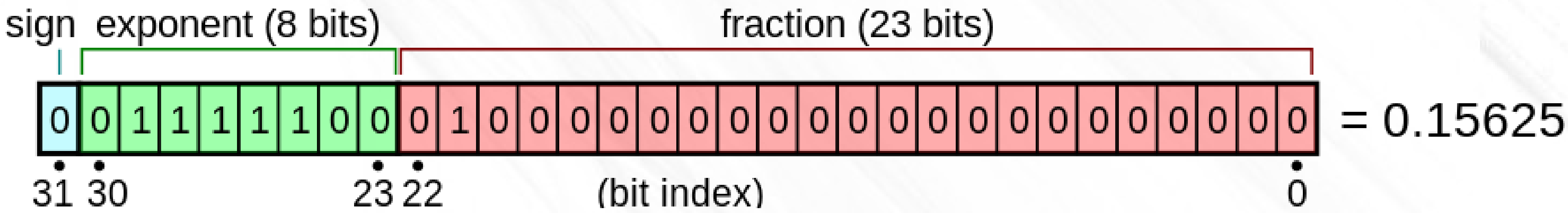
William Kahan (Wikipedia)

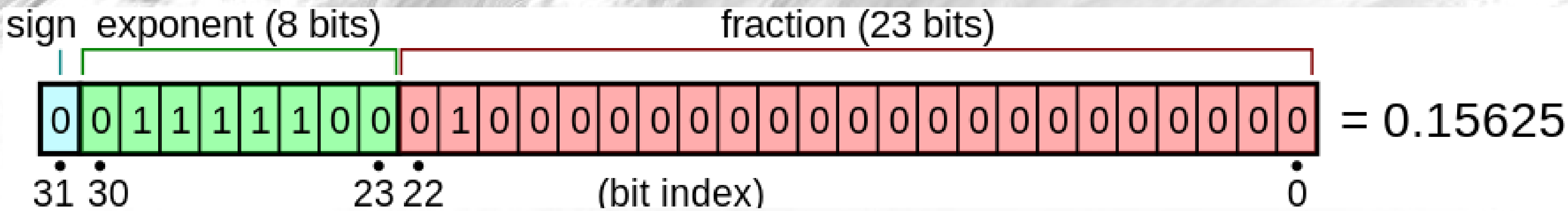
# Structure of a float (32 bit)

Floats are of form $s*2^{(e-127)}*mantissa$

s = sign bit
e = exponent
Mantissa (significand) = 1.xxx

Each group is encoded into some field in the 32 bit word as binary.






- Sign bit:
  - 0 = pos
  - 1 = neg
- Exponent: To accommodate positive and negative exponent values, exponent is “biassed” (offset).
  - Add 127 to get exponent (single)
  - Add 1023 to get exponent (double)
- Mantissa is normalized to form 1.xxxxx. In memory, only xxxx is stored, the 1 is implied.

Value = $-1^{\mathtt{sign}} \times 2^{\mathtt{exponent}-127} \times 1.\mathtt{mantissa}$

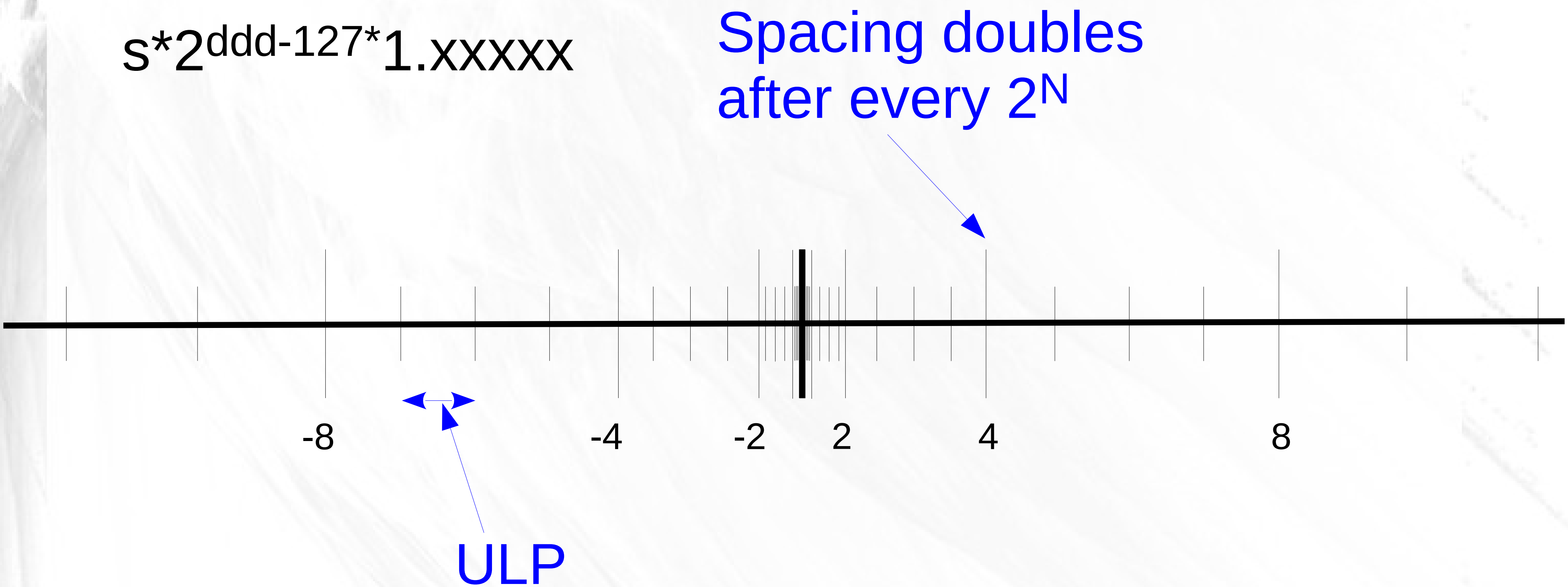
# Some examples

| Binary Value | Biased Exponent | Sign, Exponent, Mantissa |
|---|---|---|
| -1.11 | 127 | 1 01111111 11000000000000000000000 |
| +1101.101 | 130 | 0 10000010 10110100000000000000000 |
| -.00101 | 124 | 1 01111100 01000000000000000000000 |
| +100111.0 | 132 | 0 10000100 00111000000000000000000 |
| +.0000001101011 | 120 | 0 01111000 10101100000000000000000 |

$$-1^{\mathrm{sign}} \times 2^{\mathrm{exponent}-127} \times 1.\mathrm{mantissa}$$

# Floating point number line

$s*2^{ddd-127}*1.xxxxx$




- Non-periodic spacing between valid numbers.
- Smallest spacing: ULP = Unit of Least Precision, Unit in Last Place.

# IEEE-754 numerical values

- Zero – positive and negative
- Postitive & negative numbers
  - Note that all numerical values are actually rational numbers.
- There is a minimum floating point value:
  - realmin('single') = 1.17549435082229e-38 = $2^{-126}$
  - realmin('double') = 2.22507385850720e-308 = $2^{-1022}$
- Also, there is a maximum floating point value:
  - realmax('single') =  3.40282346638529e+38
  - realmax('double') =  1.79769313486232e+308

# IEEE-754 meta-numerical values

- +0 0 00000000 00000000000000000000000
- -0 1 00000000 00000000000000000000000
- +inf 0 11111111 00000000000000000000000
- -inf 1 11111111 00000000000000000000000
- Nan s 11111111 axx xxxx xxxx xxxx xxxx xxxx
- The point is to create a finite field so that any math operation stays within the defined values – the field is “closed”.

# Arithmetic with meta-numeric values

- 5/0 = inf
- -5/0 = -inf
- 0/0 = nan
- Inf + inf = inf
- inf – inf = nan
- Nan + 1 = nan
- Nan + inf = nan

# Some other examples using Octave

```
octave:1> csc(+0)
warning: division by zero
ans = Inf
octave:2> csc(-0)
warning: division by zero
ans = -Inf
octave:3> sin(inf)
ans = NaN
```

# Denorms

- Smallest normal floats:

$s*2^{ddd-127}*1.xxxxx$

1.0 x $2^{-38}$

0 00000001 00000000000000000000000

- Denorms:

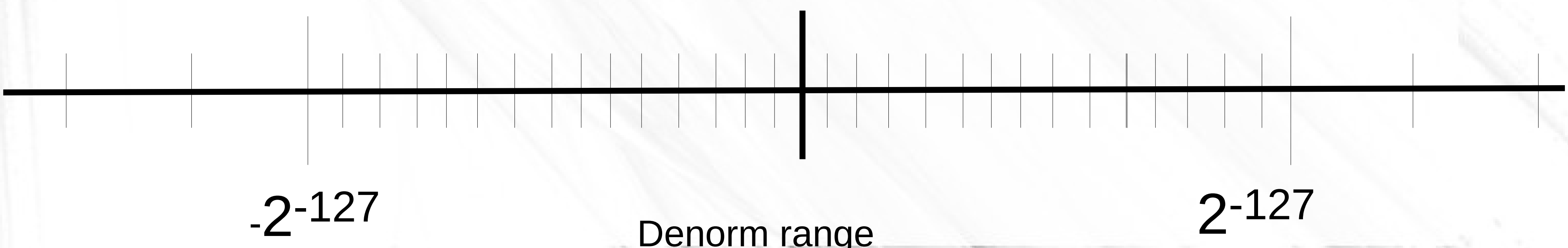
$s*2^{-127}*0.xxxxx$

0 00000000 11111111111111111111111

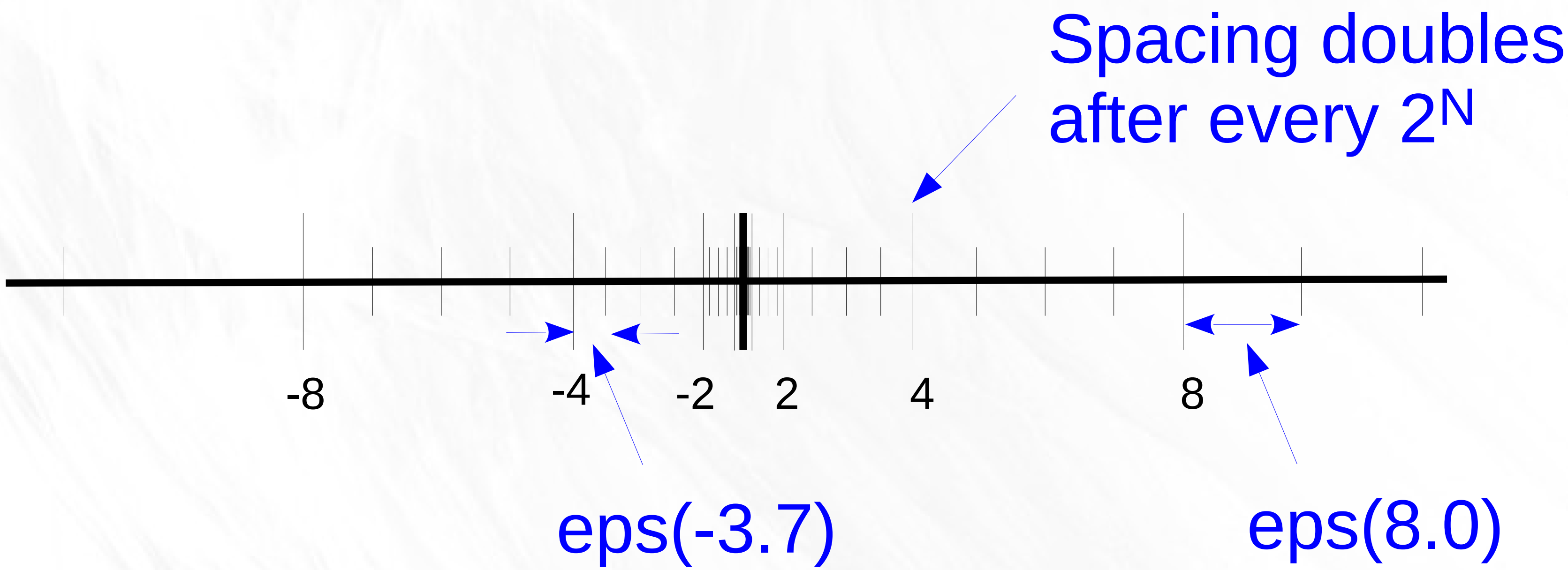
0.1111... x $2^{-38}$

0 00000000 11111111111111111111101

0 00000000 11111111111111111111011



# More about ULP




- Size of ULP depends upon position on number line.
- Depends upon single vs. double.
- Matlab function eps(x).

# Rounding modes

- IEEE spec defines 4 rounding modes for floating point computations:
  - Round to nearest (default)
  - Round towards zero
  - Round towards -inf
  - Round towards +inf
- You will probably never see these unless you write C code and set compiler flags or #pragma directives

# Next topic: Complex numbers

- Not defined by any spec, implementation left to hardware/software vendor.
- Not implemented in hardware.
- Just about everybody uses z = real + i*imag.
  - This is a pair of floating point values.
  - (As opposed to r*exp(i*theta) representation.)

# Some Octave examples

```
octave:9> exp(i*pi)
ans = -1.0000e+00 + 1.2246e-16i
```

```
octave:14> A = complex(1, 1)
A =  1 + 1i
octave:15> abs(A)
ans =  1.4142
octave:16> arg(A)
ans =  0.78540
octave:17>
octave:17>
octave:17> pi/4
ans =  0.78540
```

# How to compute absolute value?

- Don't: sqrt(r*r + i*i)
  - Result will overflow/underflow if either r or i are too large/small
  - Example:

```
octave:5> A = complex(single(1e23), single(1e31))
A = 1.0000e+23 + 1.0000e+31i
octave:6> sqrt(real(A)*real(A) + imag(A)*imag(A))
ans = Inf
octave:7> abs(A)
ans =  1.0000e+31
```

- Do: `abs(a)*sqrt(1+(b/a)^2)` for abs(a) > abs(b)

  `abs(b)*sqrt(1+(a/b)^2)` for abs(b) > abs(a)

- Better: Use built-in abs() function.

# Crazy complex behavior for meta-numeric inputs

```
octave:39> sin(inf + i*2)
ans = NaN + NaNi
octave:40> sin(inf + i*inf)
ans = NaN + Infi
```

```
octave:47> atan(inf)
ans =  1.57079632679490
octave:48> atan(complex(inf,0))
ans = NaN + NaNi
```

# Main ideas of lecture

- Computing polynomials
  - Horner's rule
  - Summing series in loop
- Computing scalar functions
  - Series summation
  - Continued fraction expansions.
- Floating point numbers
  - Sign, mantissa and exponent => rational approximation to continuous real numbers.
  - Floating point numbers lie on non-uniform grid.