PY502, Computational Physics
Instructor: Prof. Anders Sandvik
Office: SCI 450A, phone: 353-3843, e-mail: sandvik@bu.edu

Lectures: Tuesday/Thursday 11-12:15 in SCI B58
• Tutorials/discussions, Fridays 2:30 – 3:20 PM

Homework: ≈7 assignments
- Kai-Hsin Wu (khwu@bu.edu) is the grader (office SCI 351)

Grade: 100% homeworks
Course web site

http://physics.bu.edu/py502

- Lecture presentations and notes
- Example programs
- Homework assignments and solutions
- Messages (“Course News”)

Submitting homework

Use e-mail to submit programs: py502@buphy.bu.edu
- Include the program(s) as attachment(s)!
- Name your programs: e.g., hwx_lastname.f90

Turn in write-up (hardcopy) report to instructor or grader

Some discussion/collaboration on homework problems is allowed, but each student has to turn in her/his independently written programs and reports.
Computers and programming language

- Bring your own laptop to class if possible
  - operating system: Linux/Unix (similar under OS X)
  - install emulator software if you use Windows
  - computer help from Guoan Hu (office PRB 453)
- The Julia language will be used in lectures
- You can possibly use other languages for homework
  - but I strongly discourage this!
- Introduction to Julia will be given (~4-5 lectures)
- Extensive background in programming not needed
- Some Unix/Linux knowledge assumed (e.g., text editing)
- Come to office hours if you need help!

- Access to SCC computing cluster will be arranged
Course material

- Materials will be posted on the web site ahead of the lectures
- Online resources pointed out
- No additional required text

Recommended reading

- Computational Physics, by J. M. Thijssen
- Computational Physics, by N. J. Giordano and H. Nakanishi
  (free on-line with codes in many languages: http://www.nr.com/)
What is computational physics?

• “Scientific computing” in physics

• Studies of models of physical systems using computers
  - Numerical solutions of equations that cannot be done analytically
  - Direct studies of models to “simulate” a system

• Most subfields of physics use some computations, e.g.,
  - Dynamics of solar systems, galaxies, etc
  - Studies of mechanical models of earthquakes
  - Fluid dynamics; turbulence
  - Molecular dynamics of gases, fluids
  - Electrostatics and dynamics (Maxwell’s equations)
  - Electronic structure of materials
  - Statistical mechanics of polymers, magnetic systems, etc.
  - Lattice gauge theory (numerical QCD)

• Sometimes considered third “branch” of physics
  - Experimental, theoretical, computational

• Most physicists need to do some computational work
Topics covered in PY502

- The Julia programming language
- Numerical integration (principles and Julia practice)
- Numerical solution of differential equations
  - classical and quantum mechanics problems
- Monte Carlo simulations (statistical mechanics)
- Basic methods for quantum many-body (lattice) systems

Goals

- Learn the basics of the above techniques
- Gain proficiency in scientific computing in general
Teaser: The last topic of the course will combine several of the previous methods we have learned to study: Quantum Annealing (a paradigm for quantum computing). You will learn what is going on (supposedly…) in machines made by D-wave, Google,....
Why Julia?

There are traditionally two categories of computer languages:

Compiled - script file translated to machine code and linked to libraries once
- the executable program file is static, data types static
- examples: C/C++, Fortran
- fast, suitable for demanding high-performance computing
- not user-friendly handling of external packages, e.g., graphics

Interpreted - the script file is translated line-by-line at run time
- there is no static executable, allows more flexible functionality
- examples: Python, Perl, R
- slow; most time is spent translating the script over and over again
- more flexible handling of data (dynamic, automatic data typing)
- friendly integration of packages, graphics, notebooks,…
- not user-friendly for improving efficiency (e.g., precompiled parts)
Julia: first successful “best of both worlds” language
- v0 launched in 2012, v1.0 in 2018, now v1.6.2

Key: Just-in-time (just-ahead-of-time) compilation
- goes through the script line-by-line, but saves
  compiled machine code for efficiency-critical parts
  (loops, entire functions)

Almost as fast as C/C++ and Fortran (within ~10%)
- designed for high-performance scientific computing

As dynamic as Python
- data types can change dynamically, but can also be declared

Good mechanism for incorporating external packages/libraries
- C/C++ and Fortran codes can also be incorporated easily

Library module “Base” is automatically included, extensive functionality

Other modules can easily be imported and used
- growing user community, many packages available in different fields
Introduction to Julia
The language has many features; here we just cover the basics
- PY502 is not a software engineering course
- We will not cover advanced programming
- We will (later) pay attention to code performance (execution speed)

Teaching method: brief general principles + code examples
- commented codes available on the course web site
  http://physics.bu.edu/py502/lect1/examples/

<table>
<thead>
<tr>
<th>Variable types and elements to get started</th>
</tr>
</thead>
<tbody>
<tr>
<td>[int1.jl] Integer declaration and wrap-around (mod) behavior</td>
</tr>
<tr>
<td>[int2.jl] Integer declarations; modified version of int1, run-time error due to type mismatch</td>
</tr>
<tr>
<td>[randomarray.jl] Function with two methods; generates array of Float32 or Float64 random numbers</td>
</tr>
<tr>
<td>[matrix.jl] Matrices and matrix operations</td>
</tr>
</tbody>
</table>

There are not yet any good Julia books (?)
Documentation on the Julia site is quite good  https://julialang.org
- please read and practice elements we do not cover here!
Three ways to run Julia

1) Code written in file, run from terminal command line
   $ julia yourcode.jl    (list of arguments may follow)
   This is the way for serious work

2) Using interactive REPL (read-execute-print-loop) session
   $ iulia    (opens interactive session)

   julia>    Documentation: https://docs.julialang.org
   Type "?" for help, "]?" for Pkg help.
   Version 1.6.1 (2021-04-23)
   Official https://julialang.org/ release

   julia>
   - Useful for learning and testing (small code pieces)
   - Package manager (import modules with specific functionality)

3) Run in Jupyter notebook
   - Install the Julia kernel first    Examples with animations:
     http://docs.juliaplots.org
**Bit representation of integers**

A “word” representing a number in a computer consists of B bits
- normally B=32 or 64, also in some cases 16 or 128
- a group of 8 bits is called a “byte” (normally a word is 4 or 8 bytes)

<table>
<thead>
<tr>
<th>B-1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>···</td>
<td>0</td>
</tr>
</tbody>
</table>

bit index i = 0,1,…,B-1

bit values b(i) = 0/1

For signed integers, the last bit (B-1) is called the “sign bit”
- \( b_{B-1} = 0 \) for positive (or zero) values, \( b_{B-1} = 1 \) for negative values

For positive (or 0) integer I, the value corresponding to the bits is

\[
I = \sum_{i=0}^{B-1} b(i)2^i
\]

00 .... 0000 = 0
00 .... 0001 = 1
00 .... 0010 = 2,....

For \( I < 0 \), “two’s complement” representation:

\[
I = \sum_{i=0}^{B-2} b(i)2^i - b(B - 1)2^{B-1}
\]

11 .... 1111 = -1
11 .... 1110 = -2
11 .... 1101 = -3,....

- most practical way for computer algebra
- integer operations have “wrap around” behavior (mod \( 2^B \) for unsigned)
Example: integer declarations and operations

```julia
function integertest()
    a::UInt32=typemax(UInt32)
    b::UInt32=1
    c=a+b
    return a,c
end

x,y=integertest()
println(x)
println(y)
```

Base function `typemax` gives largest value
- `typemin` gives smallest

Try also with “Int32” instead of “UInt32”!
Example with an error [int2.jl]

Changing the function to (keep the rest of the previous example)

```julia
function integertest()
    a::UInt32=typemax(UInt32)
    b::UInt32=1
    b=a+1
    return a,b
end
```

Running gives this error message (+ more):

```
ERROR: LoadError: InexactError: trunc(UInt32, 4294967296)
```

Reason: My computer (and likely yours) is based on 64-bit architecture
- the constant “1” is then of type Integer64
- a+1 also becomes of type Integer64 (because too big for UInt32)
- b is declared as UInt32 and cannot represent the value pf a+1

**Integer types in Julia**

Int8, Int16, Int32, Int64, Int128
(UInt8, UInt16, UInt32, UInt64, UInt128)

Int is the default integer type
- normally same as Int64
**Bit representation of floating-point numbers**

Arbitrary real-valued numbers cannot be represented by bits
- approximated by certain rational numbers; “floating-point numbers”
- \( p \) bits for “significand” (fraction, mantissa)
- \( m \) bits for exponent
- 1 sign bit

\[
R = \text{sign} \times 2^e \sum_{i=0}^{p-1} b(i)2^{-i} \rightarrow \text{sign} \times 2^e \left( 1 + \sum_{i=0}^{p-1} b(i)2^{-(i+1)} \right)
\]

1 \( \leq \) significand \( < 2 \)

The exponent can be positive or negative
- exactly how the exponent is stored is a bit subtle (we don’t need the details)

On most computers:
- single-precision (4 bytes); \( p=23, m=8 \) (precision about 7 decimals)
- double-precision (8 bytes); \( p=53, m=10 \) (precision about 16 decimals)
- some timed 16-byte quadruple precision is available

Special values represented
+0, -0, +infinity, -infinity, “not a number” NaN
Example: floats, random numbers, arrays, multiple dispatch  [randomarray.jl]

```julia
function makerandom(n::Int)
    r=Array{Float64}(undef,n)
    for i=1:n
        r[i]=rand()
    end
    return r
end

function makerandom(m::Float64)
    n=round(Int,m)
    r=Array{Float32}(undef,n)
    for i=1:n
        r[i]=rand()
    end
    return r
end
```

- First method, Int argument
  - array with n elements (undefined contents)
  - one way to loop over values i
  - i:th element assigned a random value in [0,1)

- Second method, Float64 argument
  - round to closest integer and convert to Int
  - In general, any number of methods can be used, as long as they can be uniquely identified by their arguments
    (more on functions later)

Two function declarations, same name, different argument types
- it’s really one function with two methods
- the method that matches calling arguments is dispatched
Code calling this function:

```julia
n=5
m=convert(Float64,n)       # converts integer n to 64-bit float
a=makerandom(n)
for i=1:n
    println(i,"  ",a[i])
end
```

```julia
a=makerandom(m)
for i=1:n
    println(i,"  ",a[i])
end
```

**Floating-point types in Julia**

*Float16, Float32, Float64*

---

**Output**

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.768629462884634</td>
</tr>
<tr>
<td>2</td>
<td>0.2031804749902122</td>
</tr>
<tr>
<td>3</td>
<td>0.1664474670812679</td>
</tr>
<tr>
<td>4</td>
<td>0.5501970241421752</td>
</tr>
<tr>
<td>5</td>
<td>0.4978716671303165</td>
</tr>
</tbody>
</table>

**Note, in second method:**

*Float64 value is assigned to a Float32 variable; OK but of course some precision is lost*