Numerical Linear Algebra for Computer Science and Information Engineering

Lecture 02
Essentials of the Julia Language

Nicolas Venkovic
nicolas.venkovic@tum.de

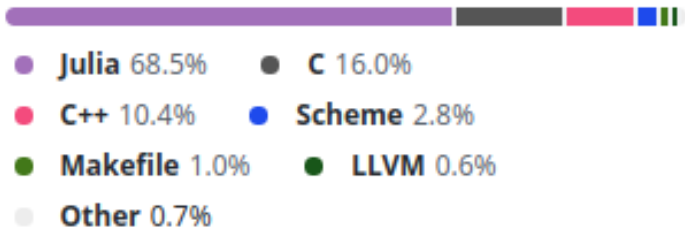
Group of Computational Mathematics
School of Computation, Information and Technology
Technical University of Munich

Summer 2025

Fact sheet of the Julia language

- Started at **MIT** in 2009 to develop a **fast open source** and **free high-level** language
- Features
  - **Dynamically typed** (also enables **static types for better performance**)
  - Just-in-time (JIT) compiled (i.e., **compiled at runtime**)
  - Provided with full-featured interactive command-line **REPL** (read-eval-print loop)
  - Designed for **parallelism** and **distributed computing** (part of the standard library)
  - **No need to vectorize code for performance**
  - Supports notebooks
- Version 1.0 released in 2018
- Current release is **1.11.5**
- Used at MIT, Stanford, UC Berkeley, Amazon, Apple, Google, IBM, Intel, Microsoft, ...
- Over 45 millions downloads as of January 2023

What is Julia made of?



- Most of the Julia **standard library** is written in **Julia**
- Julia makes use of **pre-existing libraries** (mostly in C/C++) for:
  - **BLAS/LAPACK**, however, native Julia versions exist for most functionalities. Optimized native Julia BLAS can match the performances of Intel MKL and OpenBLAS.
  - **Regular expressions** (PCRE)
  - **Downloading files** (libcurl)
  - **Low-level asynchronous IO** (libuv)
  - **Compilation** (LLVM)
  - **Extended precision arithmetic** (GMP, MPFR), but native Julia solutions also exist.

RBGS: An algorithm for comparison purposes

- Randomized block Gram-Schmidt (RBGS) procedure by Balabanov and Grigori (2021):

$$\mathrm{RBGS} : (X, \Theta) \in \mathbb{R}^{n\times m} \times \mathbb{R}^{k\times m} \mapsto (Q, R) \in \mathbb{R}^{n\times m} \times \mathbb{R}^{m\times m}$$

such that $X = QR$ where $m < k \ll n$, $(\Theta Q)^T \Theta Q = I_m$ and $\mathrm{Ran}(\Theta Q) = \mathrm{Ran}(\Theta X)$. We exploit the following structure of p blocks of size $n \times s$:

$$X = [X_{:,1:s}, X_{:,s+1:2s}, \ldots, X_{:,(p-1)s+1:ps}]$$
$$Q = [Q_{:,1:s}, Q_{:,s+1:2s}, \ldots, Q_{:,(p-1)s+1:ps}]$$

where $m = ps$.

- Exploiting those block structures, we have $(\Theta Q)^T \Theta Q = I_m \implies$

$$R_{(i-1)s+1:is,(j-1)s+1:js} = (\Theta Q_{:,(i-1)s+1:is})^T \Theta X_{;,(j-1)s+1:js},\ (i,j) \in [1,p]^2$$

RBGS: An algorithm for comparison purposes

- There are different possible implementations of RBGS algorithm. Let us consider the following:

Algorithm 1 $\mathrm{RBGS} : (X, \Theta) \mapsto (Q, R)$

1: $\mathrm{RGS}(X_{:,1:s}) \mapsto (Q_{:,1:s}, R_{1:s,1:s}, S_{:,1:s})$ ▷ $S := \Theta Q$
2: **for** $i = 2, \ldots, p$ **do**
3: $\quad P := \Theta X_{:,(i-1)s+1:is}$ ▷ Sketching
4: $\quad R_{1:(i-1)s,(i-1)s+1:is} := S^{\dagger}_{:,1:(i-1)s} P$ ▷ Block least-squares problem
5: $\quad Q_{:,(i-1)s+1:is} := X_{:,(i-1)s+1:is} - Q_{:,1:(i-1)s} R_{1:(i-1)s,(i-1)s+1:is}$ ▷ BLAS-3
6: $\quad \mathrm{RGS}(Q_{:,(i-1)s+1:is}) \mapsto (Q_{:,(i-1)s+1:is}, R_{(i-1)s+1:is,(i-1)s+1:is}, S_{:,(i-1)s+1:is})$

where RGS corresponds to RBGS with $s = 1$.

- In what follows, line 3 will be done more efficiently using a matrix-free fast transform.

Julia is close to math

```
function RBGS(X::Array{Float64,2}, p::Int, k::Int)
  n, m = size(X)
  s = Int(m / p)
  P = Array{Float64}(undef, k, s)
  Q = Array{Float64,2}(undef, n, m)
  R = zeros(Float64, m, m)
  S = Array{Float64,2}(undef, k, m)
  srht = set_srht(n, k)
  Q[:, 1:s], R[1:s, 1:s], S[:, 1:s] = RGS(X[:, 1:s], srht)
  for i in 2:p
    P .= MatrixFreeTheta(X[:, (i-1)*s+1:i*s], srht)
    R[1:(i-1)*s, (i-1)*s+1:i*s] = S[:, 1:(i-1)*s] \ P
    Q[:, (i-1)*s+1:i*s] = X[:, (i-1)*s+1:i*s]
                          .- Q[:, 1:(i-1)*s] * R[1:(i-1)*s, (i-1)*s+1:i*s]
    Q[:, (i-1)*s+1:i*s],
    R[(i-1)*s+1:i*s, (i-1)*s+1:i*s],
    S[:, (i-1)*s+1:i*s] = RGS(Q[:, (i-1)*s+1:i*s], srht)
  end
  return Q, R, S
end
```

So is Python

```
def RBGS(X, p, k):
  n, m = X.shape
  s = int(m / p)
  P = np.zeros((k, s))
  Q = np.zeros((n, m))
  R = np.zeros((m ,m))
  S = np.zeros((k, m))
  srht = set_srht(n, k)
  Q[:, :s], R[:s, :s], S[:, :s] = RGS(X[:, :s], srht)
  for i in range(1, p):
    P[:, :] = MatrixFreeTheta(X[:, i*s:(i+1)*s], srht)
    R[:i*s, i*s:(i+1)*s] = np.linalg.lstsq(S[:, :i*s], P)[0]
    Q[:, i*s:(i+1)*s] = X[:, i*s:(i+1)*s]
                        - np.matmul(Q[:, :i*s], R[:i*s, i*s:(i+1)*s])
    Q[:, i*s:(i+1)*s],
    R[i*s:(i+1)*s, i*s:(i+1)*s],
    S[:, i*s:(i+1)*s] = RGS(Q[:, i*s:(i+1)*s], srht)
  return Q, R, S
```

But not C

```
void RBGS(int n, int m, int p, int k, double *X, struct SRHT srht, double *Q, double *R, double *S) {
  int s = m / p;
  double *P = (double*)malloc(k * s * sizeof(double));
  double *Rtmp = (double*)malloc(m * s * sizeof(double));
  double *StS = (double*)malloc(m * m * sizeof(double));
  lapack_int *ipiv = (lapack_int*)malloc(m * sizeof(lapack_int));
  RGS(n, s, k, &X[0], srht, &Q[0], Rtmp, &S[0]);
  for (int v=0; v<s; v++)
    for (int u=0; u<s; u++)
      R[v * m + u] = Rtmp[v * s + u];
  for (int v=0; v<s; v++)
    for (int u=s; u<m; u++)
      R[v * m + u] = 0.;
  for (int i=1; i<p; i++) {
    BlockMatrixFreeTheta(&X[i * s * n], srht, s, P);
    cblas_dgemm(CblasColMajor, CblasTrans, CblasNoTrans, i * s, i * s, k, 1., S, k, S, k, 0., StS, i * s);
    cblas_dgemm(CblasColMajor, CblasTrans, CblasNoTrans, i * s, s, k, 1., S, k, P, k, 0., Rtmp, i * s);
    LAPACKE_dsysv(LAPACK_COL_MAJOR, 'U', i * s, s , StS, i * s, ipiv, Rtmp, i * s);
    for (int v=0; v<s; v++)
      for (int u=0; u<i*s; u++)
        R[i * s * m + v * m + u] = Rtmp[v * i * s + u];
    cblas_dcopy(n * s, &X[i * s * n], 1, &Q[i * s * n], 1);
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, n, s, i * s, -1., Q, n, Rtmp, i * s, 1., &Q[i * s * n]
    RGS(n, s, k, &Q[i * s * n], srht, &Q[i * s * n], Rtmp, &S[i * s * k]);
    for (int v=0; v<s; v++)
      for (int u=0; u<s; u++)
        R[i * s * m + v * m + i * s + u] = Rtmp[v * s + u];
    for (int v=0; v<s; v++)
      for (int u=(i+1)*s; u<m; u++)
        R[i * s * m + v * m + u] = 0.;
  }
  free(P);
  free(Rtmp);
  free(StS);
```

Stepping away from matrix computation

- SRHT refers to subsampled randomized Walsh-Hadamard transform.
- MatrixFreeTheta: $X \to \Theta X$, where Θ is a SRHT matrix given by:

$$\Theta := \sqrt{n/k} RHD$$

$R \in \mathbb{R}^{k \times n}$: Random restriction, i.e., each row is a row from I_n.
$H \in \mathbb{R}^{n \times n}$: Normalized Walsh-Hadamard transform matrix.
$D \in \mathbb{R}^{n \times n}$: Random sign flip, i.e., diagonal array with ± 1 components.

- We have $H = 1/\sqrt{n} H_n$, in which the non-normalized Walsh-Hadamard transform H_n is defined by the following recursion:

$$H_1 := \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad H_q := \begin{bmatrix} H_{q/2} & H_{q/2} \\ H_{q/2} & -H_{q/2} \end{bmatrix}, \quad q = 2, 4, \ldots, n/2, n.$$

- The recurrence of the SRHT lends itself to **divide and conquer**, which yields a **non-vectorized fast algorithm**.

Algorithm for the fast Walsh-Hadamard transform (FWHT)

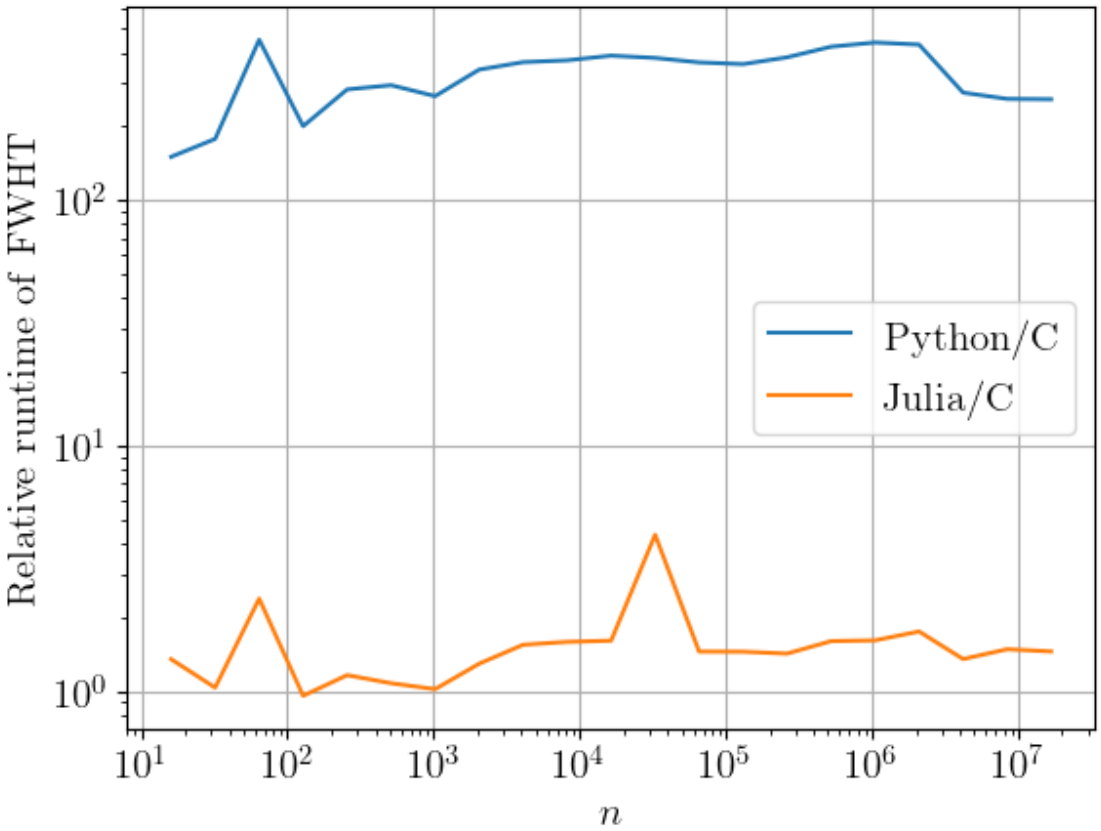
- Pseudocode of the FWHT assuming n is a power of 2:

Algorithm 2 FWHT : $z \mapsto H_n z$

```
h := 1
while h < n do
    for i = 1, 1 + 2h, ..., n − 2h, n do
        for j = i, ..., i + h − 1 do
            x := z_j
            y := z_{j+h}
            z_j := x + y
            z_{j+h} := x − y
    h := 2h
return z
```

- If n is not a power of 2, zero-pad up to the smallest power of 2 greater than n.
- Then, define MatrixFreeTheta by making use of FWHT.

Runtime of the fast Walsh-Hadamard transform (FWHT)



C is approximately
1.5x faster than **Julia**
300x faster than **Python**

For **fast Python**:
code in **C** w/ **pybind11**
code in **Fortran** w/ **f2py**

Julia is **fast**
and **high level**

Other benchmarks

- See `julialang.org/benchmarks/`

Installing Julia

- Install `juliaup`:
  - On Linux and MacOS:

    ```
    $ curl -fsSL https://install.julialang.org | sh
    ```
  - On Windows:

    ```
    > winget install --name Julia --id 9NJNWW8PVKMN -e -s msstore
    ```

  This will install the latest version of Julia.
- `juliaup` is also used to update Julia to the latest version:

  ```
  $ juliaup update
  ```
- You can use `juliaup` to install arbitrary releases, e.g.:

  ```
  $ juliaup add 1.9.3
  ```
- Start a new terminal and access the REPL as follows:

  ```
  $ julia
  julia>
  ```
- To run a specific release, e.g.,

  ```
  $ julia +1.9.3
  ```
- To see the installed versions: `$ juliaup status`

Package management

- The default project is defined by two files
  - ~/.julia/environments/v1.11/**Project.toml**: contains names of packages.
  - ~/.julia/environments/v1.11/**Manifest.toml**: contains version numbers and dependencies.
- To **clone your environment** on a new machine, only **copy** the **Project.toml** in the new default folder.
- In REPL, hit the key **]** to get in **Pkg mode**. You then get the following prompt:

```
(@1.11) pkg>
```

- To **initialize** a **new environment** in ~/MyEnvironment/, **activate** the path and **add** a **package**:

```
(@1.11) pkg> activate ~/MyEnvironment/
(@1.11) pkg> add NPZ
```

this will automatically create the Project.toml and Manifest.toml files in ~/MyEnvironment/.

Package management, cont'd

- **Load** an **existing environment** using the **activate** command as follows:

```
(@1.11) pkg> activate ~/MyEnvironment/
Activating project at '~/MyEnvironment'
```

- Check the status of an environment with the **st** command:

```
(@1.11) pkg> st
Status '~/MyEnvironment/Project.toml'
[15e1cf62] NPZ v0.4.2
```

- When loaded for the first time, use the command **instantiate** to **install** all the **packages** from **Project.toml**:

```
(@1.11) pkg> instantiate
```

- All these operations defined in the Pkg mode can be done with the Pkg package inside a script, e.g.:

```
using Pkg
activate("~/MyEnvironment/")
```

Overview of available packages

- Scientific computing
  - **LinearAlgebra.jl**: Basic linear algebra subroutines, multithreaded BLAS and LAPACK
  - **SparseArrays.jl**: Support for sparse vectors and matrices
  - **Distributed.jl**: Methods for distributed computing
  - **DistributedArrays.jl**
  - **MPI.jl**: Wrapper for the message passing interface
  - **CUDA.jl**: Main entrypoint for programming NVIDIA GPUs
  - **AlgebraicMultigrid.jl**: GPU-based implementation of AMG solvers and preconditioners
  - **Metis.jl**: Wrapper for the Metis library
  - **FFTW.jl**: Bindings to the FFTW library for fast Fourier transforms
  - **SuiteSparse.jl**: Wrapper for the SuiteSparse library
  - **Arpack.jl**: Wrapper for the Arpack library to solve large-scale eigenvalue problems
  - **BenchmarkTools.jl**: Methods for performance tracking

Overview of available packages, cont'd

- Scientific computing (cont'd)
  - **IterativeSolvers.jl**: Iterative algorithms to solve large linear systems
  - **KrylovKit.jl**: Matrix-free Krylov-based algorithms for linear, singular value and eigenvalue problems
  - **TriangleMesh.jl**: Generate and refine 2D unstructured triangular meshes
  - **Gridap.jl**: Finite elements for partial differential equations in arbitrary dimensions
  - **DifferentialEquations.jl**: Suite for numerically solving differential equations (including DAEs)
- Machine learning
  - **Flux.jl**: Go-to library for neural networks and machine learning
  - **Zygote.jl**: Automatic differentiation package
  - **Knet.jl**: Deep learning framework developed at Koç University
  - **TensorFlow.jl**: Wrapper for TensorFlow
  - **ScikitLearn.jl**: Implementation of the scikit-learn API

Interoperability with Python

- Calling Python from Julia
  - Set-up Python installation as follows using **PyCall.jl**:

```
julia> ENV["PYTHON"] = "/usr/bin/python3"
(@1.11) pkg> build PyCall
julia> using PyCall
```

  - Import packages using **pyimport** from PyCall.jl:

```
julia> np = pyimport("numpy");
julia> pushfirst!(pyimport("sys")."path", "");
julia> GS = pyimport("GramSchmidt");
```

  - Proceed seamlessly in Julia as in Python:

```
julia> x = np.random.rand(2^24);
julia> z = GS.fwht(x);
```

  - Other packages:
    - **PyPlot.jl**: Enables Matplotlib in Julia
    - **NPZ.jl**: Enables saving and loading NumPy binary data files
    - **Conda.jl**: Provides access to the Conda package manager

Interoperability with Python, cont'd

- Calling Julia from Python
  - Install **PyJulia**:

    ```
    $ pip install julia
    ```
  - The **default environment** of Julia is then **available** from Python. For example, we can do

    ```
    >>> from julia import NPZ
    ```
  - The **global namespace** of Julia's interpreter can be accessed via the module **julia.Main**:

    ```
    >>> from julia import Main
    ```
  - You can set a variable's name in the julia.Main module to send data from Python to Julia:

    ```
    >>> import numpy as np
    >>> Main.x = np.random.rand(2**24)
    ```
  - Use the **eval** function from **julia.Main** to run Julia code:

    ```
    >>> Main.eval('push!(LOAD_PATH, ".")')      # add current
    >>> Main.eval('using MyGramSchmidt: fwht') # folder to path
    >>> z = Main.eval('fwht(x)')
    ```

Interoperability with C

- Calling C libraries from Julia:
  (see docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/)
  - The C code must be available as a **shared library**
  - No additional overhead for calling from Julia compared to calling from C
  - The function **ccall** is used to call a C function with the following arguments:
    1. A (:function, "path/to/library") pair
    2. The function's return type
    3. A tuple of input types corresponding to the function's signature
    4. Argument values to be passed to the function
  - Example for `double *fwht(double *a, int n)` in srht.so:

```
julia> z_ptr = ccall((:fwht, "./srht.so"), Ptr{Cdouble},
                     (Ptr{Cdouble}, Cint), rand(2^24),
                     2^24)
julia> z = unsafe_wrap(Vector{Float64}, z_ptr, 2^24)
```

Interoperability with C, cont'd

- Calling Julia code from C:
  (see docs.julialang.org/en/v1/manual/embedding/)
  - A header file **julia.h** is made available in the Julia folder
  - Example of C code (main.c) calling Julia code:

```
#include <julia.h>
JULIA_DEFINE_FAST_TLS
int main(int argc, char *argv[]) {
  jl_init();
  jl_eval_string("push!(LOAD_PATH, \".\")");
  jl_eval_string("using MyGramSchmidt: fwht");
  jl_array_t *z = (jl_array_t*)jl_eval_string("fwht(rand(2^24))"
  double *z_data = (double*)jl_array_data(z);
  jl_atexit_hook(0);
  return 0;}
```

  - Compile as follows:

```
$ gcc -o main -fPIC
  -I/home/venkovic/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/include/julia
  -L/home/venkovic/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/lib
  -Wl,-rpath,/home/venkovic/.julia/juliaup/julia-1.11.5+0.x64.linux.gnu/lib
  main.c -ljulia
```

Shared memory multithreading

- The number of threads is set through an environment variable:

```
$ export JULIA_NUM_THREADS=12
```

- Multithreaded for loop:

```
julia> Z = Array{Float64,2}(undef, 1_024, Threads.nthreads())
julia> Threads.@threads for i in 1:Threads.nthreads()
         Z[:, i] = fwht(rand(1_024));
       end
```

- Parallel task launching:

```
julia> a = Threads.@spawn fwht(rand(1_024));
julia> b = Threads.@spawn fwht(rand(1_024));
julia> z = fetch(a) .+ fetch(b)
```

- Multithreaded BLAS:

```
julia> using LinearAlgebra.BLAS
julia> BLAS.set_num_threads(Threads.nthreads())
julia> BLAS.dot(10_000_000, rand(10_000_000), 1,
                rand(10_000_000), 1);
```

Distributed computing

- Add aliases of your machines to **/etc/hosts**:

```
192.168.1.74 hector0 ... 192.168.1.23 hector3
```

- Set-up **password-less ssh** connection between machines

hector
4x Intel Core i7

- Import the **Distributed.jl** package and **add** the **machines**:

```
using Distributed
machines = ["hector$i" for i in 0:3];
for machine in machines
  addprocs([("venkovic@$machine", Sys.CPU_THREADS)],
           tunnel=true)
end
```

- There are **processes** and **workers**. The **master process** is **not a worker**:

```
julia> println(procs(), workers())
       [1,2,3,4,5,6,7,8,9,10,11,12][2,3,4,5,6,7,8,9,10,11,12]
```

Distributed computing, cont'd$_1$

- Load code on **all processes** making use of the **@everywhere** macro:

```
@everywhere push!(LOAD_PATH, ".")
@everywhere using MyGramSchmidt: fwht
```

- Define a **shared array** as follows:

```
@everywhere using SharedArrays
Z = SharedArray(Array{Float64,2}(undef, 1_024, nworkers()))
```

- Do a **distributed for loop** as follows. The loop is distributed over **workers**:

```
@distributed for p in 1:nworkers()
  Z[:, p] = fwht(rand(1_024))
end
```

- Use **pmap** as follows to divide the work among **workers**:

```
Z = pmap(i->fwht(rand(1_024), 1:nworkers()))
```

- Do a **reduction** through a **distributed for loop** as follows:

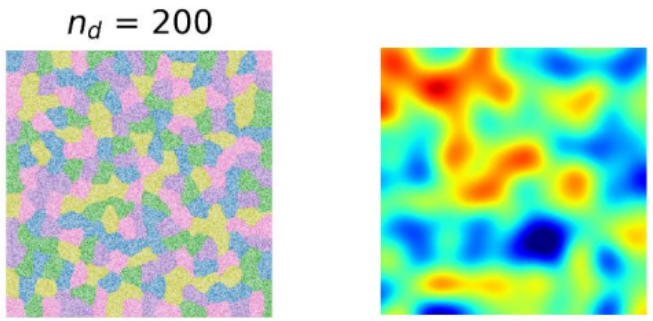
```
z = Array{Float64,1}(undef, 1_024)
z .= @distributed (.+) for _ in 1_nworkers()
  fwht(rand(1_024))
end
```

Distributed computing, cont'd$_2$

- **Dynamic mapreduce** routine for large parallel unbalanced working loads: github.com/venkovic/julia-phd-krylov-spdes/blob/master/Utils/PllUtils.jl

```
"""
    function dynamic_mapreduce!(func::Function,
                                redop::Function,
                                coll::Array{Int,1},
                                K::Array{Float64,2};
                                verbose=true,
                                Δt=2.)
Does parallel mapreduce of arrays with dynamic scheduling. This is an alternative to
K .= @distributed (redop) for c in coll
  func(c)
end
which does static scheduling and tends to crash for time consuming and unbalanced work
loads.
Another approach is given by reduce(redop, Distributed.pmap(func, K)), which requires to
allocate enough memory to store Distributed.pmap(func, K). This becomes a problem when the
number of workers and the dimensions of K are increased.
```

- Used for **parallel Karhunen-Loève decompositions on unstructured meshes**:



Venkovic (2023)

Distributed computing, cont'd$_3$

- An alternative to reduction for loops is the **mapreduce** function:

```
z = mapreduce(x->fwht(rand(1_024)), .+, 1:nworkers())
```

- Launch a **task** on **any available worker**:

```
z = fetch(@spawn fwht(rand(1_024)))
```

- Launch a **task** on a **specific process**, say the 4th process:

```
z = fetch(@spawnat 4 fwht(rand(1_024)))
```

Performance tips

- **Access arrays** in memory order, i.e., **along columns**
- **Pre-allocate** returned variables

Instead of this:

```
for i in 1:10
  x = fwht(rand(1_024))
  println(x[1:3])
end
```

Use this:

```
x = Vector{Float64}(undef, 1_024)
for i in 1:10
  x[:] = fwht(rand(1_024))
  println(x[1:3])
end
```

- **Avoid changing** the **type** of a variable

Avoid this:

```
x = 1
for i in 1:10
  x *= rand()
end
```

Performance tips, cont'd

- Write **type-stable** functions

  Instead of this: `pos(x) = x < 0 ? 0 : x`

  Use this: `pos(x) = x < 0 ? zero(x) : x`

- Use **broadcast** operators for vectorized operations

  Instead of this: `f(x::Vector{Float64}) = 3 * x.^2 + x`

  Use this: `f(x::Vector{Float64}) = 3 .* x.^2 .+ x`

Useful macros

- Macros provide a mechanism to include generated code in the final body program. We've seen @distributed, but there are other examples:
  - Use @time to time a command and get allocations info:

```
julia> @time fwht(rand(2^24));
0.622758 seconds (4 allocations: 256.000 MiB, 1.34% gc time
```

  - Use @elapsed to store time elapsed during command execution:

```
julia> dt = @elapsed fwht(rand(2^24));
julia> println("$dt seconds have passed.")
0.624587781 seconds have passed.
```

  - Use @which to identify the method invoked along with its signature and location in file:

```
julia> @which fwht(rand(1_024))
fwht(a::Vector{Float64}) in MyGramSchmidt at ~/julia-gram-s
```

  - Use @code_llvm to view the LLVM code used by the compiler:

```
julia> @code_llvm fwht(1_024)
; @ ~/julia-gram-schmidt/GramSchmidt.jl:85 within 'fwht'
define nonnull {}* @julia_fwht_819({}* nonnull align 16 der
top:
```

Useful macros, cont'd

- Use @code_native to view the native assembly code generated by the compiler:

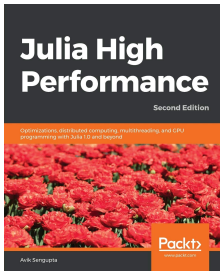
```
julia> @code_native fwht(rand(1_024));
.text
.file "fwht"
.section    .rodata.cst8,"aM",@progbits,8
.p2align    3     # -- Begin function julia_fwht_587
.LCPI0_0
  ...
```

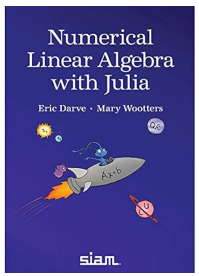
- Use @code_warntype to investigate type stability:

```
julia> @code_warntype fwht(rand(1_024));
MethodInstance for fwht(::Vector{Float64})
  from fwht(a::Vector{Float64}) in Main at REPL[16]:1
Arguments
  #self#::Core.Const(fwht)
  a::Vector{Float64}
Locals
```

Ressources

- **Documentation**: docs.julialang.org
- **Discourse board**: discourse.julialang.org
  Responsive community. Ideal for questions.
- **Slack**: julialang.slack.com
  Good for package development, and questions.
- **YouTube**: www.youtube.com/c/TheJuliaLanguage
- Sengupta, Avik. ***Julia High Performance: Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond***. Packt Publishing Ltd, 2019.

My favorite 



Used at Stanford

- Darve, Eric, and Mary Wootters. ***Numerical Linear Algebra with Julia***. Vol. 172. SIAM, 2021.
- **JuliaCon 2025** @ UPitt and CMU, Pittsburgh, PA. July 21–26, 2025.

References

- Balabanov, Oleg, and Laura Grigori. "Randomized block Gram-Schmidt process for solution of linear systems and eigenvalue problems." arXiv preprint arXiv:2111.14641 (2021).
- Balabanov, Oleg, and Laura Grigori. "Randomized Gram–Schmidt Process with Application to GMRES." SIAM Journal on Scientific Computing 44.3 (2022): A1450-A1474.

Homework

Homework

- Read chapter 10 - Julia Essentials in Darve and Wootters (2021)
- Watch Julia Lightning Round on YouTube